Team notebook

CodeRats al Fallo

March 12, 2025

Contents

1 Codes

1.1 Algebra

1.1.1 Determinant

```
const double EPS = 1E-9;
int n;
vector < vector<double> > a (n,
    vector<double> (n));

double det = 1;
for (int i=0; i<n; ++i) {
    int k = i;
    for (int j=i+1; j<n; ++j)
        if (abs (a[j][i]) > abs (a[k][i]))
            k = j;
    if (abs (a[k][i]) < EPS) {
        det = 0;
        break;
    }
    swap (a[i], a[k]);
    if (i != k)
        det = -det;
    det *= a[i][i];
    for (int j=i+1; j<n; ++j)
        a[i][j] /= a[i][i];
    for (int j=0; j<n; ++j)
        if (j != i && abs (a[j][i]) > EPS)
            for (int k=i+1; k<n; ++k)
                a[j][k] -= a[i][k] *
                    a[j][i];
}

cout << det;
```

1.1.2 Diophantic

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

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

    x0 *= c / g;
    y0 *= c / g;
    if (a < 0) x0 = -x0;
```

```
    if (b < 0) y0 = -y0;
    return true;
}
```

1.1.3 FFT

```
// Multiplicacion de polinomios en O(n
    log n)

const double PI = acos(-1.0);

namespace fft {
    struct pt {
        double r, i;
        pt(double r = 0.0, double i =
            0.0) : r(r), i(i) {}
        pt operator + (const pt &b) {
            return pt(r+b.r, i+b.i); }
        pt operator - (const pt &b) {
            return pt(r-b.r, i-b.i); }
        pt operator * (const pt &b) {
            return pt(r*b.r - i*b.i,
            r*b.i + i*b.r); }
    };
    vector<int> rev;

    void fft(vector<pt> &y, int on) {
        int n = y.size();
        for (int i = 1; i < n; i++)
            if (i < rev[i]) swap(y[i],
                y[rev[i]]);
        for (int m = 2; m <= n; m <<= 1) {
            double ang = -on * 2 * PI / m;
            pt wm(cos(ang), sin(ang));
            for (int k = 0; k < n; k +=
                m) {
                pt w(1, 0);
                for (int j = 0; j < m / 2;
                    j++) {
                    pt u = y[k + j];
                    pt t = w * y[k + j + m
                        / 2];
                    y[k + j] = u + t;
                    y[k + j + m / 2] = u -
                        t;
                    w = w * wm;
                }
            }
        }
        if (on == -1) for (int i = 0; i <
            n; i++) y[i].r /= n;
    }

    vector<ll> mul(vector<ll> &a,
        vector<ll> &b) {
        int n = 1, t = 0, la = a.size(),
            lb = b.size();
        for (; n <= (la+lb+1); n <<= 1,
            t++); t = 1<<(t-1);
        vector<pt> x1(n), x2(n);
        rev.assign(n, 0);
        for (int i = 0; i < n; i++)
            rev[i] = rev[i >> 1] >> 1 |
            (i & 1 ? t : 0);
        for (int i = 0; i < la; i++)
            x1[i] = pt(a[i], 0);
        for (int i = 0; i < lb; i++)
            x2[i] = pt(b[i], 0);
        fft(x1, 1); fft(x2, 1);
        for (int i = 0; i < n; i++) x1[i]
            = x1[i] * x2[i];
        fft(x1, -1);
        vector<ll> ans(n);
        for (int i = 0; i < n; i++)
            ans[i] = x1[i].r + 0.5;
        return ans;
    }
}
```

1.1.4 Gauss

```
const double EPS = 1e-9;
const int INF = 2; // it doesn't
    actually have to be infinity or a big
    number

int gauss (vector < vector<double> > a,
    vector<double> & ans) {
    int n = (int) a.size();
    int m = (int) a[0].size() - 1;

    vector<int> where (m, -1);
    for (int col=0, row=0; col<m &&
        row<n; ++col) {
        int sel = row;
        for (int i=row; i<n; ++i)
            if (abs (a[i][col]) > abs
                (a[sel][col]))
                sel = i;
        if (abs (a[sel][col]) < EPS)
            continue;
        for (int i=col; i<=m; ++i)
            swap (a[sel][i], a[row][i]);
```

```
        where[col] = row;

        for (int i=0; i<n; ++i)
            if (i != row) {
                double c = a[i][col] /
                    a[row][col];
                for (int j=col; j<=m; ++j)
                    a[i][j] -= a[row][j] *
                        c;
            }
        ++row;
    }

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

    for (int i=0; i<m; ++i)
        if (where[i] == -1)
            return INF;
    return 1;
}
```

1.1.5 Linear Recurrence

```
/*
Calcula el n-esimo termino de una
    recurrencia lineal (que depende de
    los k terminos anteriores).
* Llamar init(k) en el main una unica
    vez si no es necesario inicializar
    las matrices multiples veces.
Este ejemplo calcula el fibonacci de n
    como la suma de los k terminos
    anteriores de la secuencia (En la
    secuencia comun k es 2).
Agregar Matrix Multiplication con un
    construcctor vacio.
*/
matrix F, T;

void init(int k) {
    F = {k, 1}; // primeros k terminos
    F[k-1][0] = 1;
    T = {k, k}; // fila k-1 =
        coeficientes: [c_k, c_k-1, ...,
        c_1]
    for (int i = 0; i < k-1; i++)
        T[i][i+1] = 1;
    for (int i = 0; i < k; i++)
        T[k-1][i] = 1;
}
/// O(k^3 log(n))
int fib(ll n, int k = 2) {
    init(k);
    matrix ans = pow(T, n+k-1) * F;
    return ans[0][0];
}
```

1.1.6 Matrix Multiplication

```
// Estructura para realizar operaciones
    de multiplicacion y exponenciacion
    modular sobre matrices.

const int mod = 1e9+7;

struct matrix {
    vector<vector<int>> v;
    int n, m;

    matrix(int n, int m, bool o = false)
        : n(n), m(m), v(n,
        vector<int>(m)) {
        if (o) while (n--) v[n][n] = 1;
    }

    matrix operator * (const matrix &o) {
        matrix ans(n, o.m);
        for (int i = 0; i < n; i++)
            for (int k = 0; k < m; k++)
                if (v[i][k])
                for (int j = 0; j < o.m;
                    j++)
                    ans[i][j] =
                        (1ll*v[i][k]*o.v[k][j]
                        + ans[i][j]) % mod;
        return ans;
    }

    vector<int>& operator[] (int i) {
        return v[i]; }
};
```

```
matrix pow(matrix b, ll e) {
    matrix ans(b.n, b.m, true);
    while (e) {
        if (e&1) ans = ans*b;
        b = b*b;
        e /= 2;
    }
    return ans;
}
```

1.1.7 Mobius

```
const int MOBSZ = 1000000 + 10;
struct Mobius {
    bool prime[MOBSZ];
    int mu[MOBSZ];
    void init() {
        for(int i = 0; i < MOBSZ; i++)
            prime[i] = mu[i] = 1;
        for(int i = 2; i < MOBSZ; i++)
            if(prime[i]) {
            for(int j = i; j < MOBSZ;
                j+=i) {
                if(j > i) prime[j] = false;
                if(j % (1LL * i * i) == 0)
                    mu[j] = 0;
                mu[j] = -mu[j];
            }
        }
    }
} mobius;
```

1.1.8 Rank

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

int compute_rank(vector<vector<double>>
    A) {
    int n = A.size();
    int m = A[0].size();

    int rank = 0;
    vector<bool> row_selected(n, false);
    for (int i = 0; i < m; ++i) {
        int j;
        for (j = 0; j < n; ++j) {
            if (!row_selected[j] &&
                abs(A[j][i]) > EPS)
                break;
        }

        if (j != n) {
            ++rank;
            row_selected[j] = true;
            for (int p = i + 1; p < m;
                ++p)
                A[j][p] /= A[j][i];
            for (int k = 0; k < n; ++k) {
                if (k != j && abs(A[k][i])
                    > EPS) {
                    for (int p = i + 1; p
                        < m; ++p)
                        A[k][p] -= A[j][p]
                            * A[k][i];
                }
            }
        }
    }
    return rank;
}
```

1.1.9 Submasks

```
// enumera todas las submasks en O(3^n)
for (int m=0; m<(1<<n); ++m)
    for (int s=m; s; s=(s-1)&m)
 ... s and m ... /// procesar submask
```

1.1.10 XOR Basis

```
struct XorBasis {
    int K;
    vi basis;
    XorBasis(int K_) : K(K_) {
        basis.assign(K, 0); }
    int reduce(int x) {
        for(int i = K - 1; i >= 0; i--) {
            if(x & (1 << i)) x ^=
                basis[i];
        }
        return x;
    }

    bool add(int x) {
        x = reduce(x);
        if(x != 0) {
            for(int i = K - 1; i >= 0;
                i--) {
```

```
            if( x & (1 << i)) {
                basis[i] = x;
                return true;
            }
        }
    }
    return false;
}

bool check(int x) { return reduce(x)
    == 0; }
};
```

1.2 Data Structures

1.2.1 Fenwick Tree

```
int lso(int n) {return (n & (-n));}

// las consultas estn indexadas en 1
struct FenwickTree{
    vector<int> ft;
    FenwickTree(int m) {ft.assign(m + 1,
        0);};
    int rsq(int j) {
        int sum = 0;
        for(; j; j -= lso(j)) sum +=
            ft[j];
        return sum;
    }
    void upd(int i, int v) {
        for(; i < ft.size(); i += lso(i))
            ft[i] += v;
    }
};
```

1.2.2 HLD

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

const int N = 1e5 + 9, LG = 18, inf =
    1e9 + 9;

struct ST {
#define lc (n << 1)
#define rc ((n << 1) | 1)
  int t[4 * N], lazy[4 * N];
  ST() {
    fill(t, t + 4 * N, -inf);
    fill(lazy, lazy + 4 * N, 0);
  }
  inline void push(int n, int b, int e) {
    if(lazy[n] == 0) return;
    t[n] = t[n] + lazy[n];
    if(b != e) {
      lazy[lc] = lazy[lc] + lazy[n];
      lazy[rc] = lazy[rc] + lazy[n];
    }
    lazy[n] = 0;
  }
  inline int combine(int a, int b) {
    return max(a, b); //merge left and
        right queries
  }
  inline void pull(int n) {
    t[n] = max(t[lc], t[rc]); //merge
        lower nodes of the tree to get
        the parent node
  }
  void build(int n, int b, int e) {
    if(b == e) {
      t[n] = 0;
      return;
    }
    int mid = (b + e) >> 1;
    build(lc, b, mid);
    build(rc, mid + 1, e);
    pull(n);
  }
  void upd(int n, int b, int e, int i,
      int j, int v) {
    push(n, b, e);
    if(j < b || e < i) return;
    if(i <= b && e <= j) {
      lazy[n] += v;
      push(n, b, e);
      return;
    }
    int mid = (b + e) >> 1;
    upd(lc, b, mid, i, j, v);
    upd(rc, mid + 1, e, i, j, v);
    pull(n);
  }
  int query(int n, int b, int e, int i,
      int j) {
    push(n, b, e);
    if(i > e || b > j) return -inf;
    if(i <= b && e <= j) return t[n];
    int mid = (b + e) >> 1;
```

```
    return combine(query(lc, b, mid, i,
        j), query(rc, mid + 1, e, i, j));
  }
} t;

vector<int> g[N];
int par[N][LG + 1], dep[N], sz[N];
void dfs(int u, int p = 0) {
  par[u][0] = p;
  dep[u] = dep[p] + 1;
  sz[u] = 1;
  for (int i = 1; i <= LG; i++)
      par[u][i] = par[par[u][i - 1]][i -
      1];
  if (p) g[u].erase(find(g[u].begin(),
      g[u].end(), p));
  for (auto &v : g[u]) if (v != p) {
      dfs(v, u);
      sz[u] += sz[v];
      if(sz[v] > sz[g[u][0]]) swap(v,
          g[u][0]);
    }
}
int lca(int u, int v) {
  if (dep[u] < dep[v]) swap(u, v);
  for (int k = LG; k >= 0; k--) if
      (dep[par[u][k]] >= dep[v]) u =
      par[u][k];
  if (u == v) return u;
  for (int k = LG; k >= 0; k--) if
      (par[u][k] != par[v][k]) u =
      par[u][k], v = par[v][k];
  return par[u][0];
}
int kth(int u, int k) {
  assert(k >= 0);
  for (int i = 0; i <= LG; i++) if (k &
      (1 << i)) u = par[u][i];
  return u;
}
int T, head[N], st[N], en[N];
void dfs_hld(int u) {
  st[u] = ++T;
  for (auto v : g[u]) {
    head[v] = (v == g[u][0] ? head[u] :
        v);
    dfs_hld(v);
  }
  en[u] = T;
}
int n;
int query_up(int u, int v) {
  int ans = -inf;
  while(head[u] != head[v]) {
    ans = max(ans, t.query(1, 1, n,
        st[head[u]], st[u]));
    u = par[head[u]][0];
  }
  ans = max(ans, t.query(1, 1, n, st[v],
      st[u]));
  return ans;
}
int query(int u, int v) {
  int l = lca(u, v);
  int ans = query_up(u, l);
  if (v != l) ans = max(ans, query_up(v,
      kth(v, dep[v] - dep[l] - 1)));
  return ans;
}
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);

  cin >> n;
  for (int i = 1; i < n; i++) {
    int u, v;
    cin >> u >> v;
    g[u].push_back(v);
    g[v].push_back(u);
  }
  dfs(1);
  head[1] = 1;
  dfs_hld(1);
  int q;
  cin >> q;
  t.build(1, 1, n);
  while (q--) {
    string ty;
    int u, v;
    cin >> ty >> u >> v;
    if (ty == "add") {
      t.upd(1, 1, n, st[u], en[u], v);
    } else {
      cout << query(u, v) << '\n';
    }
  }
  return 0;
}
//https://www.hackerrank.com/challenges/subtree
```

1.2.3 Li$_{c}hao_{T}ree$

```
typedef long long ftype;
typedef complex<ftype> point;
```

```cpp
#define x real
#define y imag

ftype dot(point a, point b) {
    return (conj(a) * b).x();
}

ftype f(point a, ftype x) {
    return dot(a, {x, 1});
}

const int maxn = 2e5;

point line[4 * maxn];

void add_line(point nw, int v = 1, int l
    = 0, int r = maxn) {
    int m = (l + r) / 2;
    bool lef = f(nw, l) < f(line[v], l);
    bool mid = f(nw, m) < f(line[v], m);
    if(mid) {
        swap(line[v], nw);
    }
    if(r - l == 1) {
        return;
    } else if(lef != mid) {
        add_line(nw, 2 * v, l, m);
    } else {
        add_line(nw, 2 * v + 1, m, r);
    }
}

ftype get(int x, int v = 1, int l = 0,
    int r = maxn) {
    int m = (l + r) / 2;
    if(r - l == 1) {
        return f(line[v], x);
    } else if(x < m) {
        return min(f(line[v], x), get(x,
            2 * v, l, m));
    } else {
        return min(f(line[v], x), get(x,
            2 * v + 1, m, r));
    }
}
```

1.2.4 Merge Sort Tree

```cpp
const int MAXN = 200000 + 10;

vi tree[4 * MAXN];
vi arr;

void build(int v, int tl, int tr) {
        if(tl == tr) {
                tree[v] = vi(1, arr[tl]);
                return;
        }

        int tm = (tl + tr) / 2;
        build(2 * v, tl, tm);
        build(2 * v + 1, tm + 1, tr);
        merge(tree[v * 2].begin(), tree[v
            * 2].end(),
                        tree[v * 2 +
                            1].begin(),
                            tree[v * 2 +
                            1].end(),
                        back_inserter(tree[v]));
}

int query(int v, int tl, int tr, int l,
    int r, int x) {
        if(l > r) return 0;
        if(l == tl && r == tr) {
                auto pos =
                    upper_bound(tree[v].begin(),
                    tree[v].end(), x);
                return tree[v].end() - pos;
        }

        int tm = (tl + tr) / 2;
        return query(v * 2, tl, tm, l,
            min(r, tm), x)
                        + query(v * 2 + 1,
                            tm + 1, tr,
                            max(l, tm + 1),
                            r, x);
}
```

1.2.5 Mo Sqrt

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

int block_size;

struct Query {
```

```
    int l, r, idx;
    bool operator<(Query other) const
    {
        return make_pair(l / block_size,
            r) <
               make_pair(other.l /
                   block_size, other.r);
    }
};

vector<int> mo_s_algorithm(vector<Query>
    queries) {
    vector<int> answers(queries.size());
    sort(queries.begin(), queries.end());

    // TODO: initialize data structure

    int cur_l = 0;
    int cur_r = -1;
    // invariant: data structure will
        always reflect the range [cur_l,
        cur_r]
    for (Query q : queries) {
        while (cur_l > q.l) {
            cur_l--;
            add(cur_l);
        }
        while (cur_r < q.r) {
            cur_r++;
            add(cur_r);
        }
        while (cur_l < q.l) {
            remove(cur_l);
            cur_l++;
        }
        while (cur_r > q.r) {
            remove(cur_r);
            cur_r--;
        }
        answers[q.idx] = get_answer();
    }
    return answers;
}

/*
bool cmp(pair<int, int> p, pair<int,
    int> q) {
    if (p.first / BLOCK_SIZE != q.first
        / BLOCK_SIZE)
        return p < q;
    return (p.first / BLOCK_SIZE & 1) ?
        (p.second < q.second) : (p.second
        > q.second);
}
*/
```

1.2.6 Ordered Set

```
/*
Estructura de datos basada en politicas.
    Funciona como un set<> pero es
    internamente indexado, cuenta con dos
    funciones adicionales.
.find_by_order(k) -> Retorna un iterador
    al k-esimo elemento, si k >= size()
    retona .end()
.order_of_key(x) -> Retorna cuantos
    elementos hay menores (<) que x
*/

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

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

1.2.7 Persistent$_{segment_Tree}$

```
using ll = long long ;
class PersistentSegtree {
private:
    struct Node {
        ll sum = 0;
        int l = 0, r = 0;
    };

    const int n;
    vector<Node> tree;
    int timer = 1;

    Node join(int l, int r) { return
        Node{tree[l].sum + tree[r].sum,
        l, r}; }

    int build(int tl, int tr, const
        vector<int> &arr) {
        if (tl == tr) {
            tree[timer] = {arr[tl], 0, 0};
            return timer++;
        }
```

```
    int mid = (tl + tr) / 2;
    tree[timer] = join(build(tl, mid,
        arr), build(mid + 1, tr,
        arr));

    return timer++;
}

int set(int v, int pos, int val, int
    tl, int tr) {
    if (tl == tr) {
        tree[timer] = {val, 0, 0};
        return timer++;
    }

    int mid = (tl + tr) / 2;
    if (pos <= mid) {
        tree[timer] =
            join(set(tree[v].l, pos,
            val, tl, mid), tree[v].r);
    } else {
        tree[timer] = join(tree[v].l,
            set(tree[v].r, pos, val,
            mid + 1, tr));
    }

    return timer++;
}

ll range_sum(int v, int ql, int qr,
    int tl, int tr) {
    if (qr < tl || tr < ql) { return
        0ll; }
        if (ql <= tl && tr <= qr) {
            return tree[v].sum; }

        int mid = (tl + tr) / 2;
        return range_sum(tree[v].l, ql,
            qr, tl, mid) +
                range_sum(tree[v].r, ql,
                    qr, mid + 1, tr);
    }

public:
    PersistentSegtree(int n, int
        MX_NODES) : n(n), tree(MX_NODES)
        {}

    int build(const vector<int> &arr) {
        return build(0, n - 1, arr); }

    int set(int root, int pos, int val)
        { return set(root, pos, val, 0, n
        - 1); }

    ll range_sum(int root, int l, int r)
        { return range_sum(root, l, r, 0,
        n - 1); }

    int add_copy(int root) {
        tree[timer] = tree[root];
        return timer++;
    }
};

// uso : PersistentSegTree st(n, mx);
// n = tamanio del arreglo, mx = 2 * n +
    q * (2 + log(n))
// guardar raices en vector
// vi roots = { st.build(A) } ;
    inicializacion
```

1.2.8 Segment Tree (Iterativo)

```
// query = max [l, r)
// update = asignar en punto
// indexa desde 0
struct SegmentTree{
        vi tree;
        int n;
        SegmentTree(int n) : n(n) ,
            tree(2 * n + 5, 0) {}
        void upd(int p, int v){
                p += n;
                for(tree[p] = v; p > 1;
                    p>>=1) tree[p>>1] =
                    max(tree[p],tree[p^1]);
        }
        int query(int l, int r){ // [l, r)
                int res = 0;
                for(l += n, r += n; l < r;
                    l >>= 1, r >>= 1){
                        if(l & 1) res =
                            max(res,
                            tree[l++]);
                        if(r & 1) res =
                            max(res,
                            tree[--r]);
                }
                return res;
        }
};
```

1.2.9 Segment Tree(Lazy)

```
const int MOD = 1e9 + 7;
// modificar los struct
// operadores * y +
// en node + = merge
// en operation * = merge
// node * operation = como afecta la
    operacion al nodo
struct operation{
        bool swapped;
        operation() : swapped(false) {}
        operation(bool s) : swapped(s) {}

        operation operator * (const
            operation &rhs) const {
                operation res;
                res.swapped = (this ->
                    swapped) ^ rhs.swapped;
                return res;
        }

        bool id() {
                return !swapped;
        }

        void clear() {
                swapped = false;
        }
};

struct node {
    int imx, imn, mx, mn;
        node() { imx = imn = mx = mn = 0;
            }
    node(int imx, int imn, int mx, int
        mn) : imx(imx), imn(imn), mx(mx),
        mn(mn) { }

        node operator + (const node &rhs)
            const {
                node res;
                res.mx = max(this -> mx,
                    rhs.mx);
        res.mn = min(this -> mn, rhs.mn);
        if(this -> mx >= rhs.mx) res.imx
            = (this -> imx);
        else res.imx = rhs.imx;
        if(this -> mn <= rhs.mn) res.imn
            = (this -> imn);
        else res.imn = rhs.imn;
                return res;
        }

        node operator * (const operation
            &rhs) const {
                node res;
        if(rhs.swapped) {
            res.imx = imn;
            res.imn = imx;
            res.mx = MOD - mn;
            res.mn = MOD - mx;
        }else{
            res.imx = imx;
            res.imn = imn;
            res.mx = mx;
            res.mn = mn;
        }
        return res;
        }
};

const int N = 4e6 + 20;
node tree[N * 4];
operation lazy[N * 4];
pii arr[N];

void build(int v, int tl, int tr) {
        lazy[v].clear();
        if(tl == tr) {
                tree[v] =
                    node(arr[tl].second,
                    arr[tl].second,
                    arr[tl].first,
                    arr[tl].first);
                return;
        }
        int tm = (tl + tr) / 2;
        build(v * 2, tl, tm);
        build(v * 2 + 1, tm + 1, tr);
        tree[v] = tree[v * 2] + tree[v *
            2 + 1];
}

void push(int v) {
        if(lazy[v].id()) return;
        tree[v * 2] = tree[v * 2] *
            lazy[v];
        tree[v * 2 + 1] = tree[v * 2 + 1]
            * lazy[v];
```

```
        lazy[v * 2] = lazy[v] * lazy[v *
            2];
        lazy[v * 2 + 1] = lazy[v] *
            lazy[v * 2 + 1];
        lazy[v].clear();
}

void update(int v, int tl, int tr, int
    l, int r, operation &op) {
        if(l > r) return;
        if(l == tl && r == tr) {
                tree[v] = tree[v] * op;
                lazy[v] = op * lazy[v];
                return;
        }
        push(v);
        int tm = (tl + tr) / 2;
        update(v * 2, tl, tm, l, min(r,
            tm), op);
        update(v * 2 + 1, tm + 1, tr,
            max(l, tm + 1), r, op);
        tree[v] = tree[v * 2] + tree[v *
            2 + 1];
}

node query(int v, int tl, int tr, int l,
    int r) {
        if(l > r) return node();
        if(l == tl && r == tr) return
            tree[v];
        push(v);
        int tm = (tl + tr) / 2;
        return query(v * 2, tl, tm, l,
            min(r, tm))
                + query(v * 2 + 1, tm + 1,
                    tr, max(tm + 1, l), r);
}
```

1.2.10 Segment Tree

```
const int N = 100000 + 5;
const long long inf = 1e18 + 10;

struct node {
        long long sum;
        long long maxi;
        node(){
                sum = 0;
                maxi = -inf;
        }

        node(long long x) {
                sum = maxi = x;
        }

        node operator + (const node &rhs)
            const {
                node q;
                q.sum = sum + rhs.sum;
                q.maxi = max(maxi,
                    rhs.maxi);
                return q;
        }
};

int n;
int q;
node NIL;
long long a[N];
node st[4 * N];

void build(int pos = 1, int l = 1, int r
    = n) {
        if(l == r) {
                st[pos] = node(a[l]);
                return;
        }
        int mi = (l + r) / 2;
        build(2 * pos, l, mi);
        build(2 * pos + 1, mi + 1, r);
        st[pos] = st[2 * pos] + st[2 *
            pos + 1];
}

void update(int x, int y, int pos = 1,
    int l = 1, int r = n) {
        if(st[pos].maxi <= 1) return; //
            Funcion Potencial sqrt(1) = 1
        if(y < l or r < x) return;
        if(l == r) {
        // to change
                st[pos].sum =
                    sqrt(st[pos].sum);
                st[pos].maxi = st[pos].sum;
                return;
        }
        int mi = (l + r) / 2;
        update(x, y, 2 * pos, l, mi);
        update(x, y, 2 * pos + 1, mi + 1,
            r);
        st[pos] = st[2 * pos] + st[2 *
            pos + 1];
}
```

```
node query(int x, int y, int pos = 1,
    int l = 1, int r = n) {
        if(y < l or r < x) return NIL;
        if(x <= l and r <= y) return
            st[pos];
        int mi = (l + r) / 2;
        return query(x, y, 2 * pos, l,
            mi) + query(x, y, 2 * pos +
            1, mi + 1, r);
}

int main() {
        build();
    update(l, r);
    query(l, r).sum;
    query(l, r).maxi;
}
```

1.2.11 Segment$_{Tree_2D}$

```
struct segtree {
    int n, m;
    T neutro = {1, 0, 0, true};
    vector<vector<T>> st;

    segtree(int &n, int &m,
        vector<vector<T>> &a) : n(n),
        m(m) {
        st = vector<vector<T>>(2*n,
            vector<T>(2*m, neutro));
        build(n, m, a);
    }

    T get(T a, T b) {
        return max(a, b);
    }

    void build(int &n, int &m,
        vector<vector<T>> &a) {
        for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
        st[i + n][j + m] = a[i][j];

        for (int i = 0; i < n; i++)
        for (int j = m - 1; j; j--)
        st[i + n][j] = get(st[i + n][j <<
            1], st[i + n][j << 1 | 1]);

        for (int i = n - 1; i; i--)
        for (int j = 0; j < 2*m; j++)
        st[i][j] = get(st[i << 1][j],
            st[i << 1 | 1][j]);
    }

    T query(int x1, int y1, int x2, int
        y2) {
        T ans = neutro;
        vector<int> pos(2, 0);
        int node;
        for (x1 += n, x2 += n + 1; x1 <
            x2; x1 >>= 1, x2 >>= 1) { //
            rows
            node = 0;
            if (x1&1) pos[node++] = x1++;
            if (x2&1) pos[node++] = --x2;
            for (int it = 0; it < node;
                it++) {
                for (int l = y1 + m, r =
                    y2 + m + 1; l < r; l
                    >>= 1, r >>= 1) { //
                    cols
                    if (l&1) ans =
                        get(ans,
                        st[pos[it]][l++]);
                    if (r&1) ans =
                        get(ans,
                        st[pos[it]][--r]);
                }
            }
        }
        return ans;
    }

    void upd(int l, int r, T val) {
        st[l + n][r + m] = val;
        for (int j = r + m; j; j >>= 1)
            st[l][j >> 1] = get(st[l][j],
                st[l][j + 1]);

        for (int i = l + n; i; i >>= 1)
            for (int j = r + m; j; j >>=
                1)
                st[i >> 1][j] =
                    get(st[i][j], st[i +
                    1][j]);
    }
};
```

1.2.12 Segment$_{Tree_Implicit}$

```
struct Vertex {
```

```
int left, right;
int sum = 0;
Vertex *left_child = nullptr,
    *right_child = nullptr;

Vertex(int lb, int rb) {
    left = lb;
    right = rb;
}

void extend() {
    if (!left_child && left + 1 <
        right) {
        int t = (left + right) / 2;
        left_child = new Vertex(left,
            t);
        right_child = new Vertex(t,
            right);
    }
}

void add(int k, int x) {
    extend();
    sum += x;
    if (left_child) {
        if (k < left_child->right)
            left_child->add(k, x);
        else
            right_child->add(k, x);
    }
}

int get_sum(int lq, int rq) {
    if (lq <= left && right <= rq)
        return sum;
        if (max(left, lq) >= min(right,
            rq))
            return 0;
        extend();
        return left_child->get_sum(lq,
            rq) +
            right_child->get_sum(lq, rq);
    }
};
```

1.2.13 Sparce$_{T}able_{2}D$

```
const int MAX_N = 100;
const int MAX_M = 100;
const int KN = log2(MAX_N)+1;
const int KM = log2(MAX_M)+1;
int table[KN][MAX_N][KM][MAX_M];
int _log2N[MAX_N+1];
int _log2M[MAX_M+1];

int MAT[MAX_N][MAX_M];
int n, m, ic, ir, jc, jr;

void calc_log2() {
    _log2N[1] = 0;
    _log2M[1] = 0;
    for (int i = 2; i <= MAX_N; i++)
        _log2N[i] = _log2N[i/2] + 1;
    for (int i = 2; i <= MAX_M; i++)
        _log2M[i] = _log2M[i/2] + 1;
}

void build() {
    for (ir = 0; ir < n; ir++) {
        for (ic = 0; ic < m; ic++)
            table[0][ir][0][ic] =
                MAT[ir][ic];
        for (jc = 1; jc < KM; jc++)
            for (ic = 0; ic + (1
                <<(jc-1)) < m; ic++)
                table[0][ir][jc][ic] =
                    min(table[0][ir][jc-1][ic],
                    table[0][ir][jc-1][ic
                    + (1 << (jc-1))]);
    }

    for (jr = 1; jr < KN; jr++)
        for (ir = 0; ir < n; ir++)
            for (jc = 0; jc < KM; jc++)
                for (ic = 0; ic < m; ic++)
                    table[jr][ir][jc][ic]
                        =
                        min(table[jr-1][ir][jc][i
                        table[jr-1][ir+(1<<(jr-1)
}

int rmq(int x1, int y1, int x2, int y2) {
    int lenx = x2-x1+1;
    int kx = _log2N[lenx];
    int leny = y2-y1+1;
    int ky = _log2M[leny];

    int min_R1 =
        min(table[kx][x1][ky][y1],
        table[kx][x1][ky][y2 + 1 -
        (1<<ky)]);
    int min_R2 =
        min(table[kx][x2+1-(1<<kx)][ky][y1],
        table[kx][x2+1- (1<<kx)][ky][y2 +
```

```
        1 - (1<<ky)]);
    return min(min_R1, min_R2);
}
```

1.2.14 Sparse Table

Estructura de datos que permite procesar consultas por rangos.

```
const int MX = 1e5+5;
const int LG = log2(MX)+1;
int spt[LG][MX];
int arr[MX];
int n;

void build() {
    for (int i = 0; i < n; i++)
        spt[0][i] = arr[i];
    for (int j = 0; j < LG-1; j++)
        for (int i = 0; i+(1<<(j+1)) <=
            n; i++)
            spt[j+1][i] = min(spt[j][i],
                spt[j][i+(1<<j)]);
}

int rmq(int i, int j) {
    int k = 31-__builtin_clz(j-i+1);
    return min(spt[k][i],
        spt[k][j-(1<<k)+1]);
}
```

1.2.15 Treap

```
// treap
// O(logN)
struct Node {
        Node *l = 0, *r = 0;
        int val, y, c = 1;
        Node(int val) : val(val),
            y(rand()) {}
        void recalc();
};

int cnt(Node* n) { return n ? n -> c :
    0; }
void Node::recalc() { c = cnt(l) +
    cnt(r) + 1; }

template<class F> void each(Node* n, F
    f) {
        if (n) { each(n -> l, f); f(n ->
            val); each(n -> r, f); }
}

pair<Node*, Node*> split(Node* n, int k)
    {
        if (!n) return {};
        if (cnt(n -> l) >= k) { //
            "n->val >= k" for
            lower_bound(k)
                auto pa = split(n->l, k);
                n -> l = pa.second;
                n -> recalc();
                return {pa.first, n};
        } else {
                auto pa = split(n -> r, k
                    - cnt(n -> l) - 1); //
                    and just "k"
                n -> r = pa.first;
                n -> recalc();
                return {n, pa.second};
        }
}

Node* merge(Node* l, Node* r) {
        if (!l) return r;
        if (!r) return l;
        if (l -> y > r -> y) {
                l -> r = merge(l -> r, r);
                l -> recalc();
                return l;
        } else {
                r -> l = merge(l, r -> l);
                r -> recalc();
                return r;
        }
}

Node* ins(Node* t, Node* n, int pos) {
        auto pa = split(t, pos);
        return merge(merge(pa.first, n),
            pa.second);
}


// Example application: move the range
    [l, r) to index k
void move(Node*& t, int l, int r, int k)
    {
        Node *a, *b, *c;
```

```
        tie(a,b) = split(t, l); tie(b,c)
            = split(b, r - l);
        if (k <= l) t = merge(ins(a, b,
            k), c);
        else t = merge(a, ins(c, b, k -
            r));
}
```

1.3 DP

1.3.1 1D1D

```
/*Let w(i,j) be the cost function.
The required condition is the quadrangle
    inequality: for all i<=j,
    w(i,j)+w(i+1,j+1)<=w(i+1,j)+w(i,j+1)
From a practical point of view, we don't
    need to prove that w satisfies such
    an inequality,
it's easier to just list out a few
    values of w and check.
The normal form of DP is:
    dp[x]=min(dp[i]+w(i,x)) for all i
    from 0 to x-1.
Let k(x) be the optimal index for dp(x)
    i.e. dp(x) is minimum for index k(x)
    i.e. dp(x)=dp(k(x))+w(k(x),x)
Then for all i<=j, k(i)<=k(j) because of
    the quadrangle inequality
If we can't use CHT for this kind of DP
    optimization, GENERALLY they are
    solvable using this trick.
Complexity: O(nlogn)*/
int n, s, t[N], f[N];
///the cost function, (i, x]
int w(int i, int x) {
  if (i >= x) return 1e9; //inf > w(0, n)
  return s * (f[n] - f[i]) + t[x] *
      (f[x] - f[i]);
}
int dp[N];
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  cin >> n >> s;
  t[0] = 0;
  f[0] = 0;
  for (int i = 1; i <= n; i++) {
    cin >> t[i] >> f[i];
    t[i] += t[i - 1];
    f[i] += f[i - 1];
  }
  dp[0] = 0;
  vector<pair<int, int> > v; // (start
      pos, best k)
  v.push_back(make_pair(0, 0));
  for (int x = 1; x <= n; x++) {
    int k = (--lower_bound(v.begin(),
        v.end(), make_pair(x + 1,
        0)))->second;
    dp[x] = dp[k] + w(k, x);
    for (int i = (int)v.size() - 1; i >=
        0; i--) {
      int y = v[i].first, oldk =
          v[i].second;
      if (y > x && dp[x] + w(x, y) <
          dp[oldk] + w(oldk, y))
          v.pop_back();
      else {
        int l = y + 1, r = n + 1;
        while (l < r) {
          int mid = (l + r) / 2;
          if (dp[x] + w(x, mid) <
              dp[oldk] + w(oldk, mid)) r =
              mid;
          else l = mid + 1;
        }
        if (r != n + 1)
          v.push_back(make_pair(r, x));
        break;
      }
    }
    if (v.size() == 0)
        v.push_back(make_pair(0, x));
  }
  cout << dp[n] << '\n';
  return 0;
}
```

1.3.2 digitDP

```
vi get(ll n) {
    vi dig(19, 0);
    for (int i = 18; i >= 0; i--) {
        dig[i] = n % 10;
        n /= 10;
    }
    return dig;
}

vi high;
ll dp[19][50][2];
```

```
vi cnt(10);

void init() { memset(dp, -1,
    sizeof(dp)); }

ll dfs(int pos = 0, bool smaller = 0,
    int inv = 0, bool positive = 0) {
    if (pos == 19) {
        return inv;
    }
    if (smaller &&
        dp[pos][inv][positive] != -1)
        return dp[pos][inv][positive];

    ll res = 0;
    int mx_digit = smaller ? 9 :
        high[pos];

    for (int dig = 0; dig <= mx_digit;
        dig++) {
        int add = 0;
        for (int i = dig - 1; i >= 0;
            i--) add += cnt[i];
        if(dig || positive) cnt[dig]++;
        res += dfs(
            pos + 1,
            smaller || (dig < mx_digit),
            inv + add,
            positive || dig
        );
        if(dig || positive) cnt[dig]--;
    }

    return smaller ?
        dp[pos][inv][positive] = res :
            res;
}

void solve(){
    ll a, b; cin >> a >> b;

    init();
    high = get(a - 1);
    ll dpa = dfs();

    init();
    high = get(b);
    ll dpb = dfs();
    cout << dpb - dpa << endl;
}
```

1.3.3 DivideAndConquer

```
int n, m;
vector<long long> a, prefix, dp_before,
    dp_cur;

// Segment Cost [i, j] = (sm_{i..j})^2
long long C(int i, int j) {
    long long sum = prefix[j + 1] -
        prefix[i];
    return sum * sum;
}

// compute dp_cur[l], ... dp_cur[r]
    (inclusive)
void compute(int l, int r, int optl, int
    optr) {
    if (l > r) return;

    int mid = (l + r) >> 1;
    pair<long long, int> best =
        {LLONG_MAX, -1};

    for (int k = optl; k <= min(mid,
        optr); ++k) {
        long long cost = (k ? dp_before[k
            - 1] : 0);
        if (cost != LLONG_MAX)
            best = min(best, {cost + C(k,
                mid), k});
    }

    dp_cur[mid] = best.first;
    int opt = best.second;

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

long long solve() {
    dp_before.assign(n, LLONG_MAX);
    dp_cur.assign(n, 0);

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

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

    return dp_before[n - 1];
}
```

1.3.4 Knuth

```
int solve() {
    sort(cuts.begin(), cuts.end());

    int N = n + 2;
    int dp[N][N], opt[N][N];

    auto C = [&](int i, int j) {
        return cuts[j] - cuts[i];
    };

    // Inicializacin base
    for (int i = 0; i < N; ++i) {
        dp[i][i] = 0;
        if (i + 1 < N) {
            dp[i][i + 1] = 0;
            opt[i][i + 1] = i + 1;
        }
    }

    // DP con Knuth Optimization
    for (int i = N - 2; i >= 0; i--) {
        for (int j = i + 2; j < N; j++) {
            int mn = INT_MAX;
            int cost = C(i, j);
            for (int k = opt[i][j - 1]; k
                <= min(j - 1, opt[i +
                1][j]); k++) {
                int val = dp[i][k] +
                    dp[k][j] + cost;
                if (val < mn) {
                    mn = val;
                    opt[i][j] = k;
                }
            }
            dp[i][j] = mn;
        }
    }

    return dp[0][N-1];
}
```

1.3.5 RangeDP

```
vector<int> cost(n + 2);
for(int i = 1; i <= n; i++) cin >>
    cost[i];

vector<vector<int>> seg(n + 2);
for(int i = 0; i < m; i++){
    int l, r; cin >> l >> r;
    seg[r].emplace_back(l);
}

vector<int> last(n + 2);
int curr = 0;

for(int i = 1; i <= n + 1; i++){
    for(auto l : seg[i - 1]) curr =
        max(l, curr);
    last[i] = curr;
}


SegmentTree dp(n + 2);
for(int r = 1; r <= n + 1; r++){
    long long now = 1LL * cost[r] +
        dp.query(last[r], r);
    dp.upd(r, now);
}

cout << dp.query(n + 1, n + 2) << "\n";
```

1.4 Flows

1.4.1 Blossom

```
// Halla el mximo match en un grafo
    general O(E * v ^2)

struct network {
    struct struct_edge {
        int v; struct_edge * n;
    };

    typedef struct_edge* edge;

    int n;
    struct_edge pool[MAXE]; ///2*n*n;
    edge top;
    vector<edge> adj;
    queue<int> q;
    vector<int> f, base, inq, inb, inp,
        match;
    vector<vector<int>> ed;

    network(int n) : n(n), match(n, -1),
        adj(n), top(pool), f(n), base(n),
                inq(n), inb(n),
                    inp(n), ed(n,
                    vector<int>(n)) {}
```

```
void add_edge(int u, int v) {
    if(ed[u][v]) return;
    ed[u][v] = 1;
    top->v = v, top->n = adj[u],
        adj[u] = top++;
    top->v = u, top->n = adj[v],
        adj[v] = top++;
}

int get_lca(int root, int u, int v) {
    fill(inp.begin(), inp.end(), 0);
    while(1) {
        inp[u = base[u]] = 1;
        if(u == root) break;
        u = f[ match[u] ];
    }
    while(1) {
        if(inp[v = base[v]]) return v;
        else v = f[ match[v] ];
    }
}

void mark(int lca, int u) {
    while(base[u] != lca) {
        int v = match[u];
        inb[ base[u ]] = 1;
        inb[ base[v] ] = 1;
        u = f[v];
        if(base[u] != lca) f[u] = v;
    }
}

void blossom_contraction(int s, int
    u, int v) {
    int lca = get_lca(s, u, v);
    fill(inb.begin(), inb.end(), 0);
    mark(lca, u); mark(lca, v);
    if(base[u] != lca) f[u] = v;
    if(base[v] != lca) f[v] = u;
    for(int u = 0; u < n; u++){
        if(inb[base[u]]) {
            base[u] = lca;
            if(!inq[u]) {
                inq[u] = 1;
                q.push(u);
            }
        }
    }
}

int bfs(int s) {
    fill(inq.begin(), inq.end(), 0);
    fill(f.begin(), f.end(), -1);
    for(int i = 0; i < n; i++)
        base[i] = i;
    q = queue<int>();
    q.push(s);
    inq[s] = 1;
    while(q.size()) {
        int u = q.front(); q.pop();
        for(edge e = adj[u]; e; e =
            e->n) {
            int v = e->v;
            if(base[u] != base[v] &&
                match[u] != v) {
                if((v == s) ||
                    (match[v] != -1 &&
                    f[match[v]] != -1)){
                    blossom_contraction(s,
                        u, v);
                }else if(f[v] == -1) {
                    f[v] = u;
                    if(match[v] == -1)
                        return v;
                    else
                        if(!inq[match[v]])
                        {
                        inq[match[v]] =
                            1;
                        q.push(match[v]);
                    }
                }
            }
        }
    }
    return -1;
}

int doit(int u) {
    if(u == -1) return 0;
    int v = f[u];
    doit(match[v]);
    match[v] = u; match[u] = v;
    return u != -1;
}

/// (i < net.match[i]) => means match
int maximum_matching() {
    int ans = 0;
    for(int u = 0; u < n; u++)
        ans += (match[u] == -1) &&
            doit(bfs(u));
    return ans;
}
};
```

1.4.2 Dinic

```
// O (V^2 E) pero es veloz en prctica
// para obtener los valores de flujo:
// revisar aristas con capacidad > 0
// las aristas con capacidad = 0 son
    residuales
struct Edge {
    int u, v;
    ll cap, flow;
    Edge() {};
    Edge(int u, int v, ll cap) : u(u),
        v(v), cap(cap), flow(0) {}
};
struct Dinic {
    int n;
    vector<Edge> E;
    vvi g;
    vi d, pt;
    Dinic(int n): n(n), E(0), g(n),
        d(n), pt(n) {}

    void add_edge(int u, int v, ll cap) {
        E.emplace_back(u, v, cap);
        g[u].emplace_back(int(E.size()) -
            1);
        E.emplace_back(v, u, 0);
        g[v].emplace_back(int(E.size()) -
            1);
    }

    bool BFS(int S, int T) {
        queue<int> q; q.push(S);
        fill(d.begin(), d.end(), n + 1);
        d[S] = 0;
        while(!q.empty()) {
            int u = q.front(); q.pop();
            if(u == T) break;
            for(int k : g[u]) {
                Edge &e = E[k];
                if(e.flow < e.cap &&
                    d[e.v] > d[e.u] + 1) {
                    d[e.v] = d[e.u] + 1;
                    q.emplace(e.v);
                }
            }
        }
        return d[T] != n + 1;
    }

    ll DFS(int u, int T, ll flow = -1) {
        if(u == T || flow == 0) return
            flow;
        for(int &i = pt[u]; i <
            int(g[u].size()); i++) {
            Edge &e = E[g[u][i]];
            Edge &oe = E[g[u][i] ^ 1];
            if(d[e.v] == d[e.u] + 1) {
                ll amt = e.cap - e.flow;
                if(flow != -1 && amt >
                    flow) amt = flow;
                if(ll pushed = DFS(e.v, T,
                    amt)) {
                    e.flow += pushed;
                    oe.flow -= pushed;
                    return pushed;
                }
            }
        }
        return 0;
    }

    ll max_flow(int S, int T) {
        ll total = 0;
        while(BFS(S, T)) {
            fill(pt.begin(), pt.end(), 0);
            while(ll flow = DFS(S, T))
                total += flow;
        }
        return total;
    }

};
```

1.4.3 Hungarian

```
/*
 * returns (min cost, match), where L[i]
     is matched with
 * R[match[i]]. Negate costs for max
     cost. Requires N <= M.
 * O(N^2M)
*/
pair<int, vi> hungarian(const vvi &a) {
        if (a.empty()) return {0, {}};
        int n = (int)a.size() + 1, m =
            (int)a[0].size() + 1;
        vi u(n), v(m), p(m), ans(n - 1);
        for(int i = 1; i < n; i++) {
                p[0] = i;
                int j0 = 0; // add "dummy"
                    worker 0
                vi dist(m, INT_MAX),
                    pre(m, -1);
```

```
        vector<bool> done(m + 1);
        do { // dijkstra
            done[j0] = true;
            int i0 = p[j0],
                j1, delta =
                INT_MAX;
            for(int j = 1; j <
                m; j++) if
                (!done[j]) {
                    auto cur =
                        a[i0 -
                        1][j -
                        1] -
                        u[i0] -
                        v[j];
                    if (cur <
                        dist[j])
                        dist[j]
                        = cur,
                        pre[j] =
                        j0;
                    if (dist[j]
                        < delta)
                        delta =
                        dist[j],
                        j1 = j;
            }
            for(int j = 0; j <
                m; j++) {
                    if
                        (done[j])
                        u[p[j]]
                        +=
                        delta,
                        v[j] -=
                        delta;
                    else
                        dist[j]
                        -= delta;
            }
            j0 = j1;
        } while (p[j0]);
        while (j0) { // update
            alternating path
            int j1 = pre[j0];
            p[j0] = p[j1], j0
                = j1;
        }
    }
    for(int j = 1; j < m; j++) if
        (p[j]) ans[p[j] - 1] = j - 1;
    return {-v[0], ans}; // min cost
}
```

1.4.4 Maximum Bipartite Matching

```
// Halla el maximo match en un grafo
    bipartito O(|E|*|V|)

struct mbm {
    int l, r;
    vector<vector<int>> g;
    vector<int> match, vis;

    mbm(int l, int r) : l(l), r(r), g(l)
        {}

    void add_edge(int l, int r) {
        g[l].push_back(r);
    }

    bool dfs(int u) {
        for (auto &v : g[u]) {
            if (vis[v]++) continue;
            if (match[v] == -1 ||
                dfs(match[v])) {
                match[v] = u;
                return true;
            }
        }
        return false;
    }

    int max_matching() {
        int ans = 0;
        match.assign(r, -1);
        for (int u = 0; u < l; ++u) {
            vis.assign(r, 0);
            ans += dfs(u);
        }
        return ans;
    }
};

// Hopcroft Karp: O(E * sqrt(V))

const int INF = INT_MAX;

struct mbm {
    vector<vector<int>> g;
    vector<int> d, match;
    int nil, l, r;
    /// u -> 0 to l, v -> 0 to r
```

```
    mbm(int l, int r) : l(l), r(r),
        nil(l+r), g(l+r),
                        d(1+l+r, INF),
                            match(l+r, l+r)
                            {}

    void add_edge(int a, int b) {
        g[a].push_back(l+b);
        g[l+b].push_back(a);
    }

    bool bfs() {
        queue<int> q;
        for(int u = 0; u < l; u++) {
            if(match[u] == nil) {
                d[u] = 0;
                q.push(u);
            } else {
                d[u] = INF;
            }
        }
        d[nil] = INF;
        while(q.size()) {
            int u = q.front(); q.pop();
            if(u == nil) continue;
            for(auto v : g[u]) {
                if(d[ match[v] ] == INF) {
                    d[ match[v] ] = d[u]+1;
                    q.push(match[v]);
                }
            }
        }
        return d[nil] != INF;
    }

    bool dfs(int u) {
        if(u == nil) return true;
        for(int v : g[u]) {
            if(d[ match[v] ] == d[u]+1 &&
                dfs(match[v])) {
                match[v] = u; match[u] = v;
                return true;
            }
        }
        d[u] = INF;
        return false;
    }

    int max_matching() {
        int ans = 0;
        while(bfs()) {
            for(int u = 0; u < l; u++) {
                ans += (match[u] == nil &&
                    dfs(u));
            }
        }
        return ans;
    }
};
```

1.4.5 MCMF Racso

```
//Works for both directed, undirected
    and with negative cost too
//doesn't work for negative cycles
//for undirected edges just make the
    directed flag false
//Complexity: O(min(E^2 *V log V, E logV
    * flow))
using T = int;
const T inf = 1e9;
struct MCMF {
  struct edge {
    int u, v;
    T cap, cost;
    //int id;
    edge(int _u, int _v, T _cap, T
        _cost, int _id) {
      u = _u;
      v = _v;
      cap = _cap;
      cost = _cost;
      //id = _id;
    }
  };
  int n, s, t, mxid;
  T flow, cost;
  vector<vector<int>> g;
  vector<edge> e;
  vector<T> d, potential, flow_through;
  vector<int> par;
  bool neg;
  MCMF() {}
  MCMF(int _n) { // 0-based indexing
    n = _n + 10;
    g.assign(n, vector<int> ());
    neg = false;
    mxid = 0;
  }
  void add_edge(int u, int v, T cap, T
      cost, int id = -1, bool directed =
      true) {
    if(cost < 0) neg = true;
    g[u].push_back(e.size());
```

```
    e.push_back(edge(u, v, cap, cost,
        id));
    g[v].push_back(e.size());
    e.push_back(edge(v, u, 0, -cost,
        -1));
    mxid = max(mxid, id);
    if(!directed) add_edge(v, u, cap,
        cost, -1, true);
  }
  bool dijkstra() {
    par.assign(n, -1);
    d.assign(n, inf);
    priority_queue<pair<T, T>,
        vector<pair<T, T>>,
        greater<pair<T, T>> > q;
    d[s] = 0;
    q.push(pair<T, T>(0, s));
    while (!q.empty()) {
      int u = q.top().second;
      T nw = q.top().first;
      q.pop();
      if(nw != d[u]) continue;
      for (int i = 0; i <
          (int)g[u].size(); i++) {
        int id = g[u][i];
        int v = e[id].v;
        T cap = e[id].cap;
        T w = e[id].cost + potential[u] -
            potential[v];
        if (d[u] + w < d[v] && cap > 0) {
          d[v] = d[u] + w;
          par[v] = id;
          q.push(pair<T, T>(d[v], v));
        }
      }
    }
    for (int i = 0; i < n; i++) {
      if (d[i] < inf) d[i] +=
          (potential[i] - potential[s]);
    }
    for (int i = 0; i < n; i++) {
      if (d[i] < inf) potential[i] = d[i];
    }
    return d[t] != inf; // for max flow
        min cost
    // return d[t] <= 0; // for min cost
        flow
  }
  T send_flow(int v, T cur) {
    if(par[v] == -1) return cur;
    int id = par[v];
    int u = e[id].u;
    T w = e[id].cost;
    T f = send_flow(u, min(cur,
        e[id].cap));
    cost += f * w;
    e[id].cap -= f;
    e[id ^ 1].cap += f;
    return f;
  }
  //returns {maxflow, mincost}
  pair<T, T> solve(int _s, int _t, T
      goal = inf) {
    s = _s;
    t = _t;
    flow = 0, cost = 0;
    potential.assign(n, 0);
    if (neg) {
      // Run Bellman-Ford to find
          starting potential on the
          starting graph
      // If the starting graph (before
          pushing flow in the residual
          graph) is a DAG,
      // then this can be calculated in
          O(V + E) using DP:
      // potential(v) = min({potential[u]
          + cost[u][v]}) for each u -> v
          and potential[s] = 0
      d.assign(n, inf);
      d[s] = 0;
      bool relax = true;
      for (int i = 0; i < n && relax;
          i++) {
        relax = false;
        for (int u = 0; u < n; u++) {
          for (int k = 0; k <
              (int)g[u].size(); k++) {
            int id = g[u][k];
            int v = e[id].v;
            T cap = e[id].cap, w =
                e[id].cost;
            if (d[v] > d[u] + w && cap >
                0) {
              d[v] = d[u] + w;
              relax = true;
            }
          }
        }
      }
      for(int i = 0; i < n; i++) if(d[i]
          < inf) potential[i] = d[i];
    }
    while (flow < goal && dijkstra())
      flow += send_flow(t, goal - flow);
```

```
    //flow_through.assign(mxid + 10, 0);
    //for (int u = 0; u < n; u++) {
    //  for (auto v : g[u]) {
    //    if (e[v].id >= 0)
        flow_through[e[v].id] = e[v ^
        1].cap;
    //  }
    //}
    return make_pair(flow, cost);
  }
};
```

1.4.6 MinCost MaxFlow

```
Dado un grafo, halla el flujo maximo y
    el costo minimo entre el source s y
    el sink t.

struct edge {
    int u, v, cap, flow, cost;
    int rem() { return cap - flow; }
};

const int inf = 1e9;
const int MX = 405; //Cantidad maxima
    TOTAL de nodos
vector<int> g[MX]; //Lista de adyacencia
vector<edge> e; //Lista de aristas
vector<bool> in_queue; //Marca los nodos
    que estan en cola
vector<int> pre, dist, cap; //Almacena
    el nodo anterior, la distancia y el
    flujo de cada nodo
int mxflow, mncost; //Flujo maximo y
    costo minimo
int N; //Cantidad TOTAL de nodos

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

void flow(int s, int t) {
    mxflow = mncost = 0;
    in_queue.assign(N, false);
    while (true) {
        dist.assign(N, inf); dist[s] = 0;
        cap.assign(N, 0); cap[s] = inf;
        pre.assign(N, -1); pre[s] = 0;
        queue<int> q; q.push(s);
        in_queue[s] = true;

        while (q.size()) {
            int u = q.front(); q.pop();
            in_queue[u] = false;
            for (int &id : g[u]) {
                edge &ed = e[id];
                int v = ed.v;
                if (ed.rem() && dist[v] >
                    dist[u]+ed.cost) {
                    dist[v] =
                        dist[u]+ed.cost;
                    cap[v] = min(cap[u],
                        ed.rem());
                    pre[v] = id;
                    if (!in_queue[v]) {
                        q.push(v);
                        in_queue[v] = true;
                    }
                }
            }
        }
        if (pre[t] == -1) break;
        mxflow += cap[t];
        mncost += cap[t] * dist[t];
        for (int v = t; v != s; v =
            e[pre[v]].u) {
            e[pre[v]].flow += cap[t];
            e[pre[v]^1].flow -= cap[t];
        }
    }
}

void init() {
    e.clear();
    for (int i = 0; i <= N; i++) {
        g[i].clear();
    }
}

// O(V * E * 2 * log(E))
template <class type>
struct mcmf {
    struct edge { int u, v, cap, flow;
        type cost; };

    int n;
    vector<edge> ed;
    vector<vector<int>> g;
    vector<int> p;
```

```
    vector<type> d, phi;

    mcmf(int n) : n(n), g(n), p(n),
        d(n), phi(n) {}

    void add_edge(int u, int v, int cap,
        type cost) {
        g[u].push_back(ed.size());
        ed.push_back({u, v, cap, 0,
            cost});
        g[v].push_back(ed.size());
        ed.push_back({v, u, 0, 0, -cost});
    }

    bool dijkstra(int s, int t) {
        fill(d.begin(), d.end(), INF);
        fill(p.begin(), p.end(), -1);
        set<pair<type, int>> q;
        d[s] = 0;
        for(q.insert({d[s], s});
            q.size();) {
            int u = (*q.begin()).second;
                q.erase(q.begin());
            for(auto v : g[u]) {
                auto &e = ed[v];
                type nd =
                    d[e.u]+e.cost+phi[e.u]-phi[e.v];
                if(0 < (e.cap-e.flow) &&
                    nd < d[e.v]) {
                    q.erase({d[e.v], e.v});
                    d[e.v] = nd; p[e.v] =
                        v;
                    q.insert({d[e.v],
                        e.v});
                }
            }
        }
        for(int i = 0; i < n; i++) phi[i]
            = min(INF, phi[i]+d[i]);
        return d[t] != INF;
    }
    pair<int, type> max_flow(int s, int
        t) {
        type mc = 0;
        int mf = 0;
        fill(phi.begin(), phi.end(), 0);
        while(dijkstra(s, t)) {
            int flow = INF;
            for(int v = p[t]; v != -1; v
                = p[ ed[v].u ])
                flow = min(flow,
                    ed[v].cap-ed[v].flow);
            for(int v = p[t]; v != -1; v
                = p[ ed[v].u ]) {
                edge &e1 = ed[v];
                edge &e2 = ed[v^1];
                mc += e1.cost*flow;
                e1.flow += flow;
                e2.flow -= flow;
            }
            mf += flow;
        }
        return {mf, mc};
    }
};
```

1.4.7 Weighted Matching

```
Halla el mximo match con pesos O(V ^ 3)
```

```
typedef int type;
struct matching_weighted {
    int l, r;
    vector<vector<type>> c;
    matching_weighted(int l, int r) :
        l(l), r(r), c(l, vector<type>(r))
        {
        assert(l <= r);
    }

    void add_edge(int a, int b, type
        cost) { c[a][b] = cost; }

    type matching() {
        vector<type> v(r), d(r); // v:
            potential
        vector<int> ml(l, -1), mr(r, -1);
            // matching pairs
        vector<int> idx(r), prev(r);
        iota(idx.begin(), idx.end(), 0);
        auto residue = [&](int i, int j)
            { return c[i][j]-v[j]; };
        for(int f = 0; f < l; ++f) {
            for(int j = 0; j < r; ++j) {
                d[j] = residue(f, j);
                prev[j] = f;
            }
            type w;
            int j, l;
            for (int s = 0, t = 0;;) {
                if(s == t) {
                    l = s;
                    w = d[ idx[t++] ];
```

```
            for(int k = t; k < r;
                ++k) {
                j = idx[k];
                type h = d[j];
                if (h <= w) {
                    if (h < w) t =
                        s, w = h;
                    idx[k] = idx[t];
                    idx[t++] = j;
                }
            }
            for (int k = s; k < t;
                ++k) {
                j = idx[k];
                if (mr[j] < 0)
                    goto aug;
            }
        }
        int q = idx[s++], i =
            mr[q];
        for (int k = t; k < r;
            ++k) {
            j = idx[k];
            type h = residue(i, j)
                - residue(i, q) + w;
            if (h < d[j]) {
                d[j] = h;
                prev[j] = i;
                if(h == w) {
                    if(mr[j] < 0)
                        goto aug;
                    idx[k] = idx[t];
                    idx[t++] = j;
                }
            }
                }
            }
            aug:
            for (int k = 0; k < l; ++k)
                v[ idx[k] ] += d[ idx[k] ]
                    - w;
            int i;
            do {
                mr[j] = i = prev[j];
                swap(j, ml[i]);
            } while (i != f);
        }
        type opt = 0;
        for (int i = 0; i < l; ++i)
            opt += c[i][ml[i]]; // (i,
                ml[i]) is a solution
        return opt;
    }
};
```

1.5 Geometry

1.5.1 Hull

```
vector<P> convex_hull(vector<P> points) {
    int n = points.size();
    sort(points.begin(), points.end(),
        [](P p1, P p2){
        return make_pair(p1.x, p1.y) <
            make_pair(p2.x, p2.y);
    });
    vector<P> hull;
    for(int rep = 0; rep < 2; rep++) {
        // la primera halla el hull
            superior
        int S = hull.size();
        for(int i = 0; i < n; i++) {
            while((int)hull.size() >= S +
                2) {
                P A = hull.end()[-2];
                P B = hull.end()[-1];
                // el <= incluye puntos
                    colineales
                if(A.triangle(B,
                    points[i]) <= 0) break;
                hull.pop_back();
            }
            hull.push_back(points[i]);
        }
        hull.pop_back(); // derecha e
            izquierda se repiten
        reverse(points.begin(),
            points.end());
    }
    return hull;
}
```

1.5.2 Misc

```
bool intersects(P p1, P p2, P p3, P p4) {
    if((p2 - p1) * (p4 - p3) == 0) { //
        paralelos
        if(p1.triangle(p2, p3) != 0)
            return false; // no colineales
        // ahora son colineales
        // bounding boxes
        for(int it = 0; it < 2; it++) {
```

```
        if(max(p1.x, p2.x) <
           min(p3.x, p4.x)
           || max(p1.y, p2.y) <
              min(p3.y, p4.y))
              return false;
        swap(p1, p3);
        swap(p2, p4);
      }
      return true;
    }

    for(int it = 0; it < 2; it++) {
      ll t1 = p1.triangle(p2, p3), t2 =
         p1.triangle(p2, p4);
      if((t1 < 0 && t2 < 0) || (t1 > 0
         && t2 > 0)) return false;
      swap(p1, p3);
      swap(p2, p4);
    }
    return true;
}

bool on_boundary(P p, P p1, P p2) { // p
    est en el segmento p1p2 ?
    if(p.triangle(p1, p2) != 0) return
       false; // no son colineales
    return min(p1.x, p2.x) <= p.x && p.x
       <= max(p1.x, p2.x)
       && min(p1.y, p2.y) <= p.y && p.y
          <= max(p1.y, p2.y);
}

ll polygon_area(const vector<P>& poly) {
    // dividir entre dos al final
    int n = poly.size();
    ll ans = 0;
    for(int i = 1; i + 1 < n; i++) ans
       += poly[0].triangle(poly[i],
       poly[i + 1]);
    return abs(ans);
}

ll points_inside(const vector<P>& poly)
    { // teorema de pick
    int n = poly.size();
    ll on_boundary = 0;
    for(int i = 0; i < n; i++) {
       // segmento entre p[i] y p[i + 1];
       P p1 = poly[i], p2 = poly[(i + 1)
          % n];
       p2 -= p1;
       on_boundary += gcd(abs(p2.x),
          abs(p2.y));
    }
    // pick: Area = Inside + Boundary /
       2 - 1
    //        2 * Area = 2 * Inside +
       Boundary - 2
    //        Inside = (2 * Area -
       Boundary + 2) / 2
    return (polygon_area(poly) -
       on_boundary + 2) / 2;
}
```

1.5.3 Point2D

```
struct P{
    ll x, y;
    P() : x(0), y(0) {}
    P(ll x_, ll y_) : x(x_), y(y_) {}
    void read() { cin >> x >> y; }


    P operator - (const P& b) const {
       return P(x - b.x, y - b.y);
    }
    void operator-=(const P& b) {
       x -= b.x;
       y -= b.y;
    }

    ll operator *(const P& b) const {
       return 1LL * x * b.y - 1LL * y *
          b.x;
    }
    // si miro a Pb, en que lado queda c
    // positivo = izquierda
    ll triangle(const P& b, const P& c)
       const {
       return (b - *this) * (c - *this);
    }

    friend ostream& operator<<(ostream&
       os, const P& p) {
       os << p.x << " " << p.y; return
          os;
    }

};
```

1.6 Geometry-CPAlgo

1.6.1 Closest Pair of Points

```
struct pt { int x, y, id; };

struct cmp_x {
    bool operator()(const pt & a, const
        pt & b) const {
        return a.x < b.x || (a.x == b.x
            && a.y < b.y);
    }
};

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

int n;
vector<pt> a;
double mindist;
pair<int, int> best_pair;

void upd_ans(const pt & a, const pt & b)
    {
    double dist = sqrt((a.x - b.x)*(a.x
        - b.x) + (a.y - b.y)*(a.y - b.y));
    if (dist < mindist) {
        mindist = dist;
        best_pair = {a.id, b.id};
    }
}

vector<pt> t;

void rec(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(a[i], a[j]);
            }
        }
        sort(a.begin() + l, a.begin() +
            r, cmp_y());
        return;
    }

    int m = (l + r) >> 1;
    int midx = a[m].x;
    rec(l, m);
    rec(m, r);

    merge(a.begin() + l, a.begin() + m,
        a.begin() + m, a.begin() + r,
        t.begin(), cmp_y());
    copy(t.begin(), t.begin() + r - l,
        a.begin() + l);

    int tsz = 0;
    for (int i = l; i < r; ++i) {
        if (abs(a[i].x - midx) < mindist)
            {
            for (int j = tsz - 1; j >= 0
                && a[i].y - t[j].y <
                mindist; --j)
                upd_ans(a[i], t[j]);
            t[tsz++] = a[i];
        }
    }
}
```

1.6.2 Find Segment Intersection

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

struct pt {
    double x, y;
};

struct seg {
    pt p, q;
    int id;

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

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

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

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

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

struct event {
    double x;
    int tp, id;

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

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

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

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

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

pair<int, int> solve(const vector<seg>&
    a) {
    int n = (int)a.size();
    vector<event> e;
    for (int i = 0; i < n; ++i) {
        e.push_back(event(min(a[i].p.x,
            a[i].q.x), +1, i));
        e.push_back(event(max(a[i].p.x,
            a[i].q.x), -1, i));
    }
    sort(e.begin(), e.end());

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

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

```
}
```

1.6.3 Half Plane Intersection

```
// Redefine epsilon and infinity as
   necessary. Be mindful of precision
   errors.
const long double eps = 1e-9, inf = 1e9;
struct Point {
    long double x, y;
    explicit Point(long double x = 0,
        long double y = 0) : x(x), y(y) {}
    friend Point operator + (const
        Point& p, const Point& q) {
        return Point(p.x + q.x, p.y +
            q.y);
    }
    friend Point operator - (const
        Point& p, const Point& q) {
        return Point(p.x - q.x, p.y -
            q.y);
    }
    friend Point operator * (const
        Point& p, const long double& k) {
        return Point(p.x * k, p.y * k);
    }
    friend long double dot(const Point&
        p, const Point& q) {
        return p.x * q.x + p.y * q.y;
    }
    friend long double cross(const
        Point& p, const Point& q) {
        return p.x * q.y - p.y * q.x;
    }
};

// Basic half-plane struct.
struct Halfplane {

    // 'p' is a passing point of the
       line and 'pq' is the direction
       vector of the line.
    Point p, pq;
    long double angle;

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

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

    // Comparator for sorting.
    bool operator < (const Halfplane& e)
        const {
        return angle < e.angle;
    }

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

// Actual algorithm
vector<Point>
    hp_intersect(vector<Halfplane>& H) {

    Point box[4] = { // Bounding box in
        CCW order
        Point(inf, inf),
        Point(-inf, inf),
        Point(-inf, -inf),
        Point(inf, -inf)
    };

    for(int i = 0; i<4; i++) { // Add
        bounding box half-planes.
        Halfplane aux(box[i], box[(i+1) %
            4]);
        H.push_back(aux);
    }

    // Sort by angle and start algorithm
    sort(H.begin(), H.end());
    deque<Halfplane> dq;
    int len = 0;
    for(int i = 0; i < int(H.size());
        i++) {
```

```cpp
        // Remove from the back of the
            deque while last half-plane
            is redundant
        while (len > 1 &&
            H[i].out(inter(dq[len-1],
            dq[len-2]))) {
            dq.pop_back();
            --len;
        }

        // Remove from the front of the
            deque while first half-plane
            is redundant
        while (len > 1 &&
            H[i].out(inter(dq[0],
            dq[1]))) {
            dq.pop_front();
            --len;
        }

        // Special case check: Parallel
            half-planes
        if (len > 0 &&
            fabsl(cross(H[i].pq,
            dq[len-1].pq)) < eps) {
            // Opposite parallel
                half-planes that ended up
                checked against each other.
            if (dot(H[i].pq,
                dq[len-1].pq) < 0.0)
                return vector<Point>();

            // Same direction half-plane:
                keep only the leftmost
                half-plane.
            if (H[i].out(dq[len-1].p)) {
                dq.pop_back();
                --len;
            }
            else continue;
        }

        // Add new half-plane
        dq.push_back(H[i]);
        ++len;
    }

    // Final cleanup: Check half-planes
        at the front against the back and
        vice-versa
    while (len > 2 &&
        dq[0].out(inter(dq[len-1],
        dq[len-2]))) {
        dq.pop_back();
        --len;
    }

    while (len > 2 &&
        dq[len-1].out(inter(dq[0],
        dq[1]))) {
        dq.pop_front();
        --len;
    }

    // Report empty intersection if
        necessary
    if (len < 3) return vector<Point>();

    // Reconstruct the convex polygon
        from the remaining half-planes.
    vector<Point> ret(len);
    for(int i = 0; i+1 < len; i++) {
        ret[i] = inter(dq[i], dq[i+1]);
    }
    ret.back() = inter(dq[len-1], dq[0]);
    return ret;
}
```

1.6.4 Length Union

```cpp
int length_union(const vector<pair<int,
    int>> &a) {
    int n = a.size();
    vector<pair<int, bool>> x(n*2);
    for (int i = 0; i < n; i++) {
        x[i*2] = {a[i].first, false};
        x[i*2+1] = {a[i].second, true};
    }

    sort(x.begin(), x.end());

    int result = 0;
    int c = 0;
    for (int i = 0; i < n * 2; i++) {
        if (i > 0 && x[i].first >
            x[i-1].first && c > 0)
            result += x[i].first -
                x[i-1].first;
        if (x[i].second)
            c--;
        else
            c++;
    }
    return result;
```

```
}
```

1.6.5 Manhattan MST

```
struct point {
    long long x, y;
};

// Returns a list of edges in the format
    (weight, u, v).
// Passing this list to Kruskal
    algorithm will give the Manhattan MST.
vector<tuple<long long, int, int>>
    manhattan_mst_edges(vector<point> ps)
    {
    vector<int> ids(ps.size());
    iota(ids.begin(), ids.end(), 0);
    vector<tuple<long long, int, int>>
        edges;
    for (int rot = 0; rot < 4; rot++) {
        // for every rotation
        sort(ids.begin(), ids.end(),
            [&](int i, int j){
            return (ps[i].x + ps[i].y) <
                (ps[j].x + ps[j].y);
        });
        map<int, int, greater<int>>
            active; // (xs, id)
        for (auto i : ids) {
            for (auto it =
                active.lower_bound(ps[i].x);
                it != active.end();
            active.erase(it++)) {
                int j = it->second;
                if (ps[i].x - ps[i].y >
                    ps[j].x - ps[j].y)
                    break;
                assert(ps[i].x >= ps[j].x
                    && ps[i].y >= ps[j].y);
                edges.push_back({(ps[i].x
                    - ps[j].x) + (ps[i].y
                    - ps[j].y), i, j});
            }
            active[ps[i].x] = i;
        }
        for (auto &p : ps) { // rotate
            if (rot & 1) p.x *= -1;
            else swap(p.x, p.y);
        }
    }
    return edges;
}
```

1.6.6 Minkowski Sum

```
struct pt{
    long long x, y;
    pt operator + (const pt & p) const {
        return pt{x + p.x, y + p.y};
    }
    pt operator - (const pt & p) const {
        return pt{x - p.x, y - p.y};
    }
    long long cross(const pt & p) const {
        return x * p.y - y * p.x;
    }
};
void reorder_polygon(vector<pt> & P){
    size_t pos = 0;
    for(size_t i = 1; i < P.size(); i++){
        if(P[i].y < P[pos].y || (P[i].y
            == P[pos].y && P[i].x <
            P[pos].x))
            pos = i;
    }
    rotate(P.begin(), P.begin() + pos,
        P.end());
}

vector<pt> minkowski(vector<pt> P,
    vector<pt> Q){
    // the first vertex must be the
        lowest
    reorder_polygon(P);
    reorder_polygon(Q);
    // we must ensure cyclic indexing
    P.push_back(P[0]);
    P.push_back(P[1]);
    Q.push_back(Q[0]);
    Q.push_back(Q[1]);
    // main part
    vector<pt> result;
    size_t i = 0, j = 0;
    while(i < P.size() - 2 || j <
        Q.size() - 2){
        result.push_back(P[i] + Q[j]);
        auto cross = (P[i + 1] -
            P[i]).cross(Q[j + 1] - Q[j]);
        if(cross >= 0 && i < P.size() - 2)
            ++i;
        if(cross <= 0 && j < Q.size() - 2)
            ++j;
```

```
    }
    return result;
}
```

1.6.7 Planar Graph Faces

```
struct Point {
    int64_t x, y;

    Point(int64_t x_, int64_t y_):
        x(x_), y(y_) {}

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

    int64_t cross (const Point & p)
        const {
        return x * p.y - y * p.x;
    }

    int64_t cross (const Point & p,
        const Point & q) const {
        return (p - *this).cross(q -
            *this);
    }

    int half () const {
        return int(y < 0 || (y == 0 && x
            < 0));
    }
};
```

```
std::vector<std::vector<size_t>>
    find_faces(std::vector<Point>
    vertices,
    std::vector<std::vector<size_t>> adj)
    {
    size_t n = vertices.size();
    std::vector<std::vector<char>>
        used(n);
    for (size_t i = 0; i < n; i++) {
        used[i].resize(adj[i].size());
        used[i].assign(adj[i].size(), 0);
        auto compare = [&](size_t l,
            size_t r) {
            Point pl = vertices[l] -
                vertices[i];
            Point pr = vertices[r] -
                vertices[i];
            if (pl.half() != pr.half())
                return pl.half() <
                    pr.half();
            return pl.cross(pr) > 0;
        };
        std::sort(adj[i].begin(),
            adj[i].end(), compare);
    }
    std::vector<std::vector<size_t>>
        faces;
    for (size_t i = 0; i < n; i++) {
        for (size_t edge_id = 0; edge_id
            < adj[i].size(); edge_id++) {
            if (used[i][edge_id]) {
                continue;
            }
            std::vector<size_t> face;
            size_t v = i;
```

```
            size_t e = edge_id;
            while (!used[v][e]) {
                used[v][e] = true;
                face.push_back(v);
                size_t u = adj[v][e];
                size_t e1 =
                    std::lower_bound(adj[u].begin
                    adj[u].end(), v,
                    [&](size_t l, size_t
                    r) {
                    Point pl = vertices[l]
                        - vertices[u];
                    Point pr = vertices[r]
                        - vertices[u];
                    if (pl.half() !=
                        pr.half())
                        return pl.half() <
                            pr.half();
                    return pl.cross(pr) >
                        0;
                }) - adj[u].begin() + 1;
                if (e1 == adj[u].size()) {
                    e1 = 0;
                }
                v = u;
                e = e1;
            }
            std::reverse(face.begin(),
                face.end());
            int sign = 0;
            for (size_t j = 0; j <
                face.size(); j++) {
                size_t j1 = (j + 1) %
                    face.size();
```

```
            size_t j2 = (j + 2) %
                face.size();
            int64_t val =
                vertices[face[j]].cross(vertices[face[j1]],
                vertices[face[j2]]);
            if (val > 0) {
                sign = 1;
                break;
            } else if (val < 0) {
                sign = -1;
                break;
            }
        }
        if (sign <= 0) {
            faces.insert(faces.begin(),
                face);
        } else {
            faces.emplace_back(face);
        }
    }
}
return faces;
}
```

1.6.8 Point in Polygon

```
struct pt {
    long long x, y;
    pt() {}
    pt(long long _x, long long _y) :
        x(_x), y(_y) {}
    pt operator+(const pt &p) const {
        return pt(x + p.x, y + p.y); }
    pt operator-(const pt &p) const {
        return pt(x - p.x, y - p.y); }
    long long cross(const pt &p) const {
        return x * p.y - y * p.x; }
    long long dot(const pt &p) const {
        return x * p.x + y * p.y; }
    long long cross(const pt &a, const
        pt &b) const { return (a -
        *this).cross(b - *this); }
    long long dot(const pt &a, const pt
        &b) const { return (a -
        *this).dot(b - *this); }
    long long sqrLen() const { return
        this->dot(*this); }
};

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

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

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

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

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

    n--;
    seq.resize(n);
    for (int i = 0; i < n; i++)
        seq[i] = points[i + 1] -
            points[0];
    translation = points[0];
}

bool pointInConvexPolygon(pt point) {
    point = point - translation;
    if (seq[0].cross(point) != 0 &&
            sgn(seq[0].cross(point)) !=
                sgn(seq[0].cross(seq[n -
                1])))
        return false;
    if (seq[n - 1].cross(point) != 0 &&
            sgn(seq[n - 1].cross(point))
                != sgn(seq[n -
                1].cross(seq[0])))
        return false;
```

```
    if (seq[0].cross(point) == 0)
        return seq[0].sqrLen() >=
            point.sqrLen();

    int l = 0, r = n - 1;
    while (r - l > 1) {
        int mid = (l + r) / 2;
        int pos = mid;
        if (seq[pos].cross(point) >= 0)
            l = mid;
        else
            r = mid;
    }
    int pos = l;
    return pointInTriangle(seq[pos],
        seq[pos + 1], pt(0, 0), point);
}
```

1.6.9 Rotating Callipers

```
vector<pii> all_anti_podal(int n,
    vector<Point> &p) {
    int p1 = 0, p2 = 0; // two "pointers"
    vector<pii> result;

    // parallel edges should't be
        visited twice
    vector<bool> vis(n, false);

    for (;p1<n;p1++) {
        // the edge that we are going to
            consider in this iteration
        // the datatype is Point, but it
            acts as a vector
        Point base = p[nx(p1)] - p[p1];

        // the last condition makes sure
            that the cross products don't
            have the same sign
        while (p2 == p1 || p2 == nx(p1)
            || sign(cross(base, p[nx(p2)]
            - p[p2])) == sign(cross(base,
            p[p2] - p[pv(p2)]))) {
            p2 = nx(p2);
        }

        if (vis[p1]) continue;
        vis[p1] = true;

        result.push_back({p1, p2});
        result.push_back({nx(p1), p2});

        // if both edges from p1 and p2
            are parallel to each other
        if (cross(base, p[nx(p2)] -
            p[p2]) == 0) {
            result.push_back({p1,
                nx(p2)});
            result.push_back({nx(p1),
                nx(p2)});
            vis[p2] = true;
        }
    }

    return result;
}
```

1.7 Graphs

1.7.1 2SAT

```
struct SATSolver {
        // Assumes that nodes are
            0-indexed
        int n;
        int m;
        vector<bool> vis;
        vector<int> comp;
        vector<int> order;
        vector<int> component;
        vector<vector<int>> G;
        vector<vector<int>> Gt;

        SATSolver(int n, int m) : n(n),
            m(m) {
                // X_i = 2i
                // ~X_i = 2i + 1
                comp.resize(2 * n);
                vis.resize(2 * n, false);
                G.resize(2 * n,
                    vector<int>());
                Gt.resize(2 * n,
                    vector<int>());
        }

        void add_edge(int u, int v) {
                // u OR v
                G[u ^ 1].emplace_back(v);
                G[v ^ 1].emplace_back(u);
                Gt[v].emplace_back(u ^ 1);
                Gt[u].emplace_back(v ^ 1);
        }
```

```
void DFS1(int u) {
        vis[u] = true;
        for(int v : G[u]) {
                if(vis[v])
                    continue;
                DFS1(v);
        }
        order.emplace_back(u);
}

void DFS2(int u) {
        vis[u] = true;
        for(int v : Gt[u]) {
                if(vis[v])
                    continue;
                DFS2(v);
        }
        component.emplace_back(u);
}

void get_scc() {
        for(int i = 0; i < 2 * n;
            i++) {
                if(vis[i])
                    continue;
                DFS1(i);
        }
        reverse(order.begin(),
            order.end());
        fill(vis.begin(),
            vis.end(), false);
        int component_id = 0;
        for(int u : order) {
                if(vis[u])
                    continue;
                component.clear();
                DFS2(u);
                for(int x :
                    component)
                    comp[x] =
                    component_id;
                component_id += 1;
        }
}

vector<int> solve() {
        vector<int> res(n);
        get_scc();
        for(int i = 0; i < n; i++)
            {
                int val = 2 * i;
                if(comp[val] ==
                    comp[val ^ 1])
                    return
                    vector<int>();
                if(comp[val] <
                    comp[val ^ 1])
                    res[i] = 0;
                else res[i] = 1;
        }
        return res;
}
};
```

1.7.2 Articulation Points

```
int n;
int m;
int timer;
int low[N];
int tin[N];
bool vis[N];
bool cut_point[N];
vector<int> G[N];

void DFS(int u, int p = -1) {
        tin[u] = low[u] = timer++;
        vis[u] = true;
        int children = 0;
        for(int v : G[u]) {
                if(v == p) continue;
                if(vis[v]) {
                        // Es una
                            backedge,
                            aporta a low[u]
                        low[u] =
                            min(low[u],
                            tin[v]);
                }
                else {
                        // Es una
                            tree-edge,
                            verificar si u
                            es cut point
                        DFS(v, u);
                        // Minimizamos el
                            low del padre
                            con el del hijo
                        low[u] =
                            min(low[u],
                            low[v]);
```

```
                // Ya tenemos
                   procesado low[v]
                if(p != -1 and
                   low[v] >=
                   tin[u]) {
                     // u es
                        articulacion
                        si
                        low[v]
                        >=
                        tin[u]
                        cuando u
                        no es la
                        raz
                     cut_point[u]
                        = true;
                }
                children++;
            }
        }
        // Si es la raiz, es articulacion
           si tiene 2 o ms hijos
        if(p == -1 and children > 1)
           cut_point[u] = true;
}
```

1.7.3 Bellman Ford

```
int n;
int m;
int D[N]; // D[u] : Minima distancia de
   src a u usando <= k aristas en la
   k-sima iteracin
bool vis[N]; // vis[u] : El nodo se ha
   vuelto alcanzable por src
vector<tuple<int, int, int>> edges;

// retorna true si no hay ciclos
   negativos
bool bellman_ford(int src) {
        for(int i = 0; i < n; i++) {
                D[i] = -1;
                vis[i] = false;
        }
        D[src] = 0;
        vis[src] = true;
        for(int i = 1; i < n; i++) {
                for(auto e : edges) {
                        int u, v, w;
                        tie(u, v, w) = e;
                        if(not vis[u])
                           continue;
                        if(not vis[v] or
                           D[v] > D[u] +
                           w) {
                                D[v] = D[u]
                                   + w;
                                vis[v] =
                                   true;
                        }
                }
        }
        for(auto e : edges) {
                int u, v, w;
                tie(u, v, w) = e;
                if(not vis[u]) continue;
                if(not vis[v] or D[v] >
                   D[u] + w) {
                                return false; //
                                   Ciclo negativo
                                   alcanzable por
                                   src
                        }
                }
                return true;
}
```

1.7.4 Bridges

```
int n; // para guardar el numero de
   vertices
int m; // aristas
int k; // numero de componentes
   2-conexas en el arbol final
int C[N]; // componente de cada vertice
int to[N], from[N]; // vertice de salida
   y llegada para cada arista
int timer; // para el DFS
int low[N];
int tin[N];
bool vis[N];
bool bridge[N]; // es puente ?
vector<int> T[N]; // arbol final
vector<pair<int, int>> G[N]; // first =
   v, second = id de arista

void DFS(int u, int p = -1) {
        vis[u] = true;
        low[u] = tin[u] = timer++;
        for(auto edge : G[u]) {
                int v, e;
                tie(v, e) = edge;
```

```
            if(v == p) continue;
            if(vis[v]) {
                low[u] =
                    min(low[u],
                    tin[v]);
            }
            else {
                DFS(v, u);
                low[u] =
                    min(low[u],
                    low[v]);
                if(low[v] >
                    tin[u]) {
                        bridge[e] =
                            true;
                }
            }
        }
}

void build_tree() {
        DFS(1);
        for(int i = 1; i <= n; i++)
            vis[i] = false;
        k = 0;
        for(int i = 1; i <= n; i++) {
                if(vis[i]) continue;
                queue<int> Q;
                Q.emplace(i);
                vis[i] = true;
                while(!Q.empty()) {
                        int u = Q.front();
                            Q.pop();
                        C[u] = k;
                        for(auto edge :
                            G[u]) {
                                int v, e;
                                tie(v, e) =
                                    edge;
                                if(bridge[e])
                                    continue;
                                if(vis[v])
                                    continue;
                                Q.emplace(v);
                                vis[v] =
                                    true;
                        }
                }
                k += 1;
        }
        for(int i = 1; i <= m; i++) {
                if(not bridge[i]) continue;
                int u = C[to[i]], v =
                    C[from[i]];
                T[u].emplace_back(v);
                T[v].emplace_back(u);
        }
}
```

1.7.5 Centroid Decomposition

```
const int INF = 1e6;
const int LOG = 18;
struct CentroidDecomposition{
    vvi tree, cd;
    vi sz, par, ans, dep;
    vector<bool> removed;
    vvi up;
int root;
CentroidDecomposition(int n) :
    tree(n), cd(n), root(-1), sz(n),
    par(n), ans(n, INF), dep(n),
    removed(n) {
    up.assign(LOG, vi(n, -1));
}

void add_edge(int u, int v) {
    tree[u].push_back(v);
    tree[v].push_back(u);
}

int dfs(int u, int p = -1) {
    sz[u] = 1;
    for(int v : tree[u]) {
        if(removed[v] || v == p)
            continue;
        sz[u] += dfs(v, u);
    }
    return sz[u];
}

int find_centroid(int u, int
    comp_sz, int p = -1) {
    for(int v : tree[u]) {
        if(removed[v] || v == p)
            continue;
        if(sz[v] > comp_sz / 2)
            return find_centroid(v,
            comp_sz, u);
    }
    return u;
}
```

```
void decompose(int u, int p = -1) {
    int comp_sz = dfs(u);
    int centroid = find_centroid(u,
        comp_sz);
    if(root == -1) root = centroid;
    par[centroid] = p;
    if(p != -1) {
        cd[p].push_back(centroid);
        cd[centroid].push_back(p);
    }
    removed[centroid] = true;
    for(int v : tree[centroid]) {
        if(!removed[v]) decompose(v,
            centroid);
    }
}

void dfs_LCA(int u, int p = -1) {
    if(p != -1) dep[u] = dep[p] + 1;
    up[0][u] = p;
    for(int v : tree[u]) if(v != p) {
        dfs_LCA(v, u);
    }
}

void init_LCA(int n) {
    dfs_LCA(0);
    for(int lg = 1; lg < LOG; lg++) {
        for(int i = 0; i < n; i++) {
            if(up[lg - 1][i] != -1)
                up[lg][i] = up[lg -
                    1][up[lg - 1][i]];
        }
    }
}
```

```
int lift(int u, int len) {
    while(len) {
        int jump = __builtin_ctz(len);
        u = up[jump][u];
        len &= (len - 1);
    }
    return u;
}

int LCA(int u, int v) {
    if(dep[u] > dep[v]) swap(u, v);
    v = lift(v, dep[v] - dep[u]);
    if(u == v) return u;
    for(int lg = LOG - 1; lg >= 0;
        lg--) {
        if(up[lg][u] != up[lg][v]){
            u = up[lg][u];
            v = up[lg][v];
        }
    }
    return up[0][u];
}

int dist(int u, int v) {
    int lca = LCA(u, v);
    return dep[u] + dep[v] - 2 *
        dep[lca];
}

void update(int u) {
    int fu = u;
    while(u != -1) {
        ans[u] = min(ans[u], dist(fu,
            u));
```

```
            u = par[u];
        }
    }

    int query(int u) {
        int fu = u;
        int res = INF;
        while(u != -1) {
            res = min(res, ans[u] +
                dist(u, fu));
            u = par[u];
        }
        return res;
    }

};
```

1.7.6 Dijkstra

```
int n;
int m;
int D[N];
vector<pair<int, int>> G[N];


// las distancias mnimas se guardan en D
// indexa en 0
void Dijkstra(int src) {
        for(int i = 0; i < n; i++) D[i] =
            -1;
        D[src] = 0;
        priority_queue<pair<int, int>,
            vector<pair<int, int>>,
            greater<pair<int, int>>> Q;
```

```
            // Min PQ
        Q.emplace(0, src);
        while(!Q.empty()) {
                int dis, u;
                tie(dis, u) = Q.top();
                    Q.pop();
                if(dis != D[u]) continue;
                    // Verificacion de que
                    u no ha sido visitado
                    todavia
                for(auto e : G[u]) {
                        int v, w;
                        tie(v, w) = e;
                        if(D[v] == -1 or
                            D[v] > D[u] +
                            w) {
                                D[v] = D[u]
                                    + w;
                                Q.emplace(D[v],
                                    v);
                        }
                }
        }
}
```

1.7.7 Eulerian Walk Directed

```
// indexa en 0!!!
// si el grafo no tiene aristas retornar
    false
struct EulerianDirected {
    int n, m;
    vvi g;
    vi in, out;
    vi path;

    EulerianDirected(int n_, int m_) :
        n(n_), m(m_) {
        g.resize(n);
        in.resize(n);
        out.resize(n);
    }

    void add_edge(int u, int v) {
        g[u].push_back(v);
        in[v]++;
        out[u]++;
    }

    void dfs(int node) {
        while(!g[node].empty()) {
            int son = g[node].back();
            g[node].pop_back();
            dfs(son);
        }
        path.push_back(node);
    }

    vi solve() {
        int in_out = -1, out_in = -1;
        for(int i = 0; i < n; i++) {
            if(abs(in[i] - out[i]) > 1)
                return {};
            if(in[i] == out[i] + 1) {
                if(in_out == -1) in_out =
                    i;
                else return {};
            }
            if(out[i] == in[i] + 1) {
                if(out_in == -1) out_in =
                    i;
                else return {};
            }
        }
        if(in_out != -1) {
            if(out_in == -1) return {};
            dfs(out_in);
        }else{
            for(int i = 0; i < n; i++) {
                if(in[i]) {
                    dfs(i);
                    break;
                }
            }
        }
        if((int)path.size() != m + 1)
            return {};
        reverse(path.begin(), path.end());
        return path;
    }

};
```

1.7.8 Eulerian Walk Undirected

```
struct EulerianUndirected { // eulerian
    walk(Hierholzer)
    int n, m;
    vector<vii> g;
    vi path, degree;
    vector<bool> seen;
```

```
    EulerianUndirected(int n_, int m_) :
        n(n_), m(m_) {
        g.resize(n);
        seen.assign(m, false);
        degree.resize(n);
    }

    void add_edge(int u, int v, int i) {
        g[u].emplace_back(v, i);
        g[v].emplace_back(u, i);
        degree[u]++;
        degree[v]++;
    }

    void dfs(int node) {
        while(!g[node].empty()) {
            auto [son, idx] =
                g[node].back();
            g[node].pop_back();
            if(seen[idx]) continue;
            seen[idx] = true;
            dfs(son);
        }
        path.push_back(node);
    }

    vi solve() {
        int cnt_odd = 0;
        for(int i = 0; i < n; i++) {
            if(degree[i] % 2) {
                cnt_odd++;
            }
        }
        if(cnt_odd > 2) return {};
        int start = -1;
        if(cnt_odd == 2) {
            for(int i = 0; i < n; i++) {
                if(degree[i] % 2) start =
                    i;
            }
        }else{
            for(int i = 0; i < n; i++) {
                if(degree[i]) start = i;
            }
        }
        assert(start != -1);
        dfs(start);
        if((int)path.size() != m + 1)
            return {};
        return path;
    }

};
```

1.7.9 FastSCC

```
struct TarjanSCC {
    vvi G; // Lista de adyacencia
    vi st, low, num;
    vi comp;
    int n, timer;
    int num_comps;

    TarjanSCC(int n_) : n(n_),
        num_comps(0) {
        G.resize(n);
        comp.assign(n, -1);
        low.resize(n);
        num.assign(n, -1);
        st.clear();
        timer = num_comps = 0;
    }

    void add_edge(int u, int v) {
        G[u].push_back(v);
    }

    void DFS(int u) {
        num[u] = low[u] = timer++;
        st.push_back(u);
        for(int v : G[u]) {
            if(num[v] == -1) {
                DFS(v);
                low[u] = min(low[u],
                    low[v]);
            }else if(comp[v] == -1)
                low[u] = min(low[u],
                low[v]);
        }

        if(num[u] == low[u]) {
            int y = st.back();
            do {
                y = st.back();
                comp[y] = num_comps;
                st.pop_back();
            } while(y != u);
            num_comps++;
        }

    }

    void build_SCC() {
        for(int i = 0; i < n; i++) {
```

```cpp
                if(num[i] == -1) DFS(i);
            }
        }

};


struct SCC {
    vvi G; // Lista de adyacencia
    vvi Gt; // Grafo transpuesto
    vi order; // para ordenar por tiempo
        de salida en dfs1
    vi comp;
    vector<bool> vis;
    int n;
    int num_comps;
    SCC(int n_) : n(n_), num_comps(0) {
        G.resize(n);
        Gt.resize(n);
        vis.assign(n, false);
        comp.assign(n, -1);
    }

    void add_edge(int u, int v) {
        G[u].push_back(v);
        Gt[v].push_back(u);
    }

    void DFS1(int u) {
        vis[u] = true;
        for(int v : G[u]) {
            if(vis[v]) continue;
            DFS1(v);
        }
        order.emplace_back(u);
    }

    void DFS2(int u, int comp_id) {
        vis[u] = true;
        for(int v : Gt[u]) {
            if(vis[v]) continue;
            DFS2(v, comp_id);
        }
        comp[u] = comp_id;
    }

    void build_SCC() {
        // Ordenar por tiempo de salida
        for(int i = 0; i < n; i++) {
            if(vis[i]) continue;
            DFS1(i);
        }
        reverse(order.begin(),
            order.end());
        fill(vis.begin(), vis.end(),
            false);
        int cur_comp_id = 0;
        for(int i : order) {
            if(vis[i]) continue;
            DFS2(i, cur_comp_id);
            cur_comp_id++;
        }
        num_comps = cur_comp_id;
    }

};
```

1.7.10 Floyd Warshall

```cpp
const int N = 100 + 5;
const int inf = 2e9 + 10;

int n;
int m;
int d[N][N];

/*
    * inicializar el arreglo d con INF,
        a menos que i == j
        for(int i = 0; i < n; i++) {
                for(int j = 0; j < n; j++)
                    {
                        d[i][j] = i == j ?
                            0 : inf;
                }
        }
    * para escoger siempre la menor
    * arista en caso de aristas mltiples
        for(int i = 0; i < m; i++) {
                int u, v, w;
                scanf("%d %d %d", &u, &v,
                    &w);
                d[u][v] = min(d[u][v], w);
        }
*/

bool floyd_warshall(){
    for(int k = 0; k < n; k++) {
                for(int i = 0; i < n; i++)
                    {
                        for(int j = 0; j <
                            n; j++) {
```

```
                        if(d[i][k]
                            == inf
                            ||
                            d[k][j]
                            == inf)
                            continue;
                        if(d[i][j]
                            >
                            d[i][k]
                            +
                            d[k][j])
                            d[i][j]
                            =
                            d[i][k]
                            +
                            d[k][j];
                }
            }
        }

    // Termina Floyd-Warshall
    // comprobacion de ciclos
        negativos
    for(int i = 0; i < n; i++) {
        if(d[i][i] < 0) {
            return false;
        }
    }
    return true;
}
```

1.7.11 Heavy Light Decomposition

```
/*
Para inicializar llamar build().
Agregar Segment Tree con un constructor
    vacio,
 actualizaciones puntuales y declarar el
     valor neutro de forma global.
Para consultas sobre aristas guardar el
    valor de cada arista
en su nodo hijo y cambiar pos[u] por
    pos[u]+1 en la linea 54.
*/

typedef int T; //tipo de dato del segtree
const int MX = 1e5+5;
vector<int> g[MX];
int par[MX], dep[MX], sz[MX];
int pos[MX], top[MX], value[MX];
vector<T> arr;
int idx;

int pre(int u, int p, int d) {
    par[u] = p; dep[u] = d;
    int aux = 1;
    for (auto &v : g[u]) {
        if (v != p) {
            aux += pre(v, u, d+1);
            if (sz[v] >= sz[g[u][0]])
                swap(v, g[u][0]);
        }
    }
    return sz[u] = aux;
}

void hld(int u, int p, int t) {
    arr[idx] = value[u]; //vector para
        inicializar el segtree
    pos[u] = idx++;
    top[u] = t < 0 ? t = u : t;
    for (auto &v : g[u]) {
        if (v != p) {
            hld(v, u, t);
            t = -1;
        }
    }
}

segtree sgt;

void build(int n, int root) {
    idx = 0;
    arr.resize(n);
    pre(root, root, 0);
    hld(root, root, -1);
    sgt = segtree(arr);
}

T query(int u, int v) {
    T ans = neutro;
    while (top[u] != top[v]) {
        if (dep[top[u]] > dep[top[v]])
            swap(u, v);
        ans = min(ans,
            sgt.query(pos[top[v]],
            pos[v]));
        v = par[top[v]];
    }
    if (dep[u] > dep[v]) swap(u, v);
    ans = min(ans, sgt.query(pos[u],
        pos[v]));
```

```
    return ans;
}

void upd(int u, T val) {
    sgt.upd(pos[u], val);
}
```

1.7.12 LCA

```
/*
Dados los nodos u y v de un arbol
    determina cual es el ancestro comun
    mas bajo entre u y v.
*Tambien puede determinar la arista de
    peso maximo/minimo entre los nodos u
    y v (Para esto quitar los "//")
Se debe ejecutar la funcion dfs()
    primero, el padre de la raiz es s
    mismo, w es el valor a almacenar del
    padre.
*/
const int N = 4e5+2, inf = 1e9, LOG2 =
    20;
int dep[N]; // Profundidad de cada nodo
int par[LOG2][N]; // Sparse table para
    guardar los padres
//int rmq[LOG2][N]; // Sparse table para
    guardar pesos

struct edge { int v, w; };
vector<edge> g[N];

void dfs(int u, int p, int d, int w){
    dep[u] = d;
    par[0][u] = p;
    // rmq[0][u] = w;
    for(int j = 1; j < LOG2; j++){
        par[j][u] = par[j-1][par[j-1][u]];
        // rmq[j][u] = max(rmq[j-1][u],
            rmq[j-1][par[j-1][u]]);
    }
    for(auto &ed: g[u]){
        int v = ed.v;
        int val = ed.w;
        if(v == p)continue;
        dfs(v, u, d+1, val);
    }
}

int lca(int u, int v){
    // int ans = -1;
    if(dep[v] < dep[u])swap(u, v);
    int d = dep[v]-dep[u];
    for(int j = LOG2-1; j >= 0; j--){
        if(d >> j & 1){
            // ans = max(ans, rmq[j][v]);
            v = par[j][v];
        }
    }
    // if(u == v)return ans;
    if(u == v)return u;
    for(int j = LOG2-1; j >= 0; j--){
        if(par[j][u] != par[j][v]){
            // ans = max({ans, rmq[j][u],
                rmq[j][v]});
            u = par[j][u];
            v = par[j][v];
        }
    }
    // return max({ans, rmq[1][u],
        rmq[0][v]}); // si la info es de
        los nodos
    // return max({ans, rmq[0][u],
        rmq[0][v]}); // si la info es de
        las aristas
    return par[0][u];
}
```

1.7.13 Prim

```
// Prim clsico, retorna el MST de un
    grafo

int n;
int m;
int q[N];
bool vis[N];
int wedge[N];
vector<pair<int, int>> G[N];

int Prim(int src) {
        memset(wedge, -1, sizeof wedge);
        wedge[src] = 0;
        priority_queue<pair<int, int>,
            vector<pair<int, int>>,
            greater<pair<int, int>>> Q;
        Q.emplace(0, src);
        while(!Q.empty()) {
                int we, u;
                tie(we, u) = Q.top();
                    Q.pop();
                vis[u] = true;
                for(auto e : G[u]) {
```

```
                int v, w;
                tie(v, w) = e;
                if(wedge[v] == -1
                    or wedge[v] >
                    w) {
                        wedge[v] =
                            w;
                        Q.emplace(wedge[v],
                            v);
                }
            }
        }
        for(int i = 1; i <= n; i++) {
            if(not vis[i]) return -1;
        }
        return accumulate(wedge + 1,
            wedge + n + 1, 0);
}
```

```
/*
 *
 * Caso especial para un grafo completo
 * no es posible construir el grafo
    (memoria)
 * pero es posible un algoritmo O(n^2)
 *
 */
int n;
int x[N];
int y[N];
bool vis[N];
int edge[N]; // Minima arista que cruza
    desde S hasta mi nodo

int dis(int i, int j) {
        return abs(x[i] - x[j]) +
            abs(y[i] - y[j]);
}

int Prim(int src) {
        vis[src] = true;
        for(int i = 0; i < n; i++)
            edge[i] = dis(src, i);
        for(int i = 1; i < n; i++) {
                // El nodo al que llega la
                    arista ligera es al
                    argmin(edge[i]) pero
                    con vis[i] = false
                int v = -1;
                for(int j = 0; j < n; j++)
                    {
                        if(vis[j])
                            continue;
                        if(v == -1 or
                            edge[v] >
                            edge[j]) v = j;
                }
                vis[v] = true;
                for(int j = 0; j < n; j++)
                    {
                        if(vis[j])
                            continue;
                        edge[j] =
                            min(edge[j],
                            dis(v, j));
                }
        }
        return accumulate(edge, edge + n,
            0);
}
```

1.7.14 SCC

```
int n;
int m;
bool in[N];
bool vis[N];
int comp[N];
vector<int> order;
vector<int> component;
vector<int> G[2][N]; // en G[1] es el
    grafo transpuesto

void DFS(int id, int u) {
        vis[u] = id ^ 1;
        for(int v : G[id][u]) {
                if(vis[v] == (id ^ 1))
                    continue;
                DFS(id, v);
        }
        if(id == 0) order.emplace_back(u);
        else component.emplace_back(u);
}

int solve() {
        order.clear();
        component.clear();
        for(int i = 0; i < n; i++) {
                if(vis[i]) continue;
                DFS(0, i);
        }
```

```
    reverse(order.begin(),
       order.end());
    vector<vector<int>> res;
    for(int u : order) {
        if(not vis[u]) continue;
        component.clear();
        DFS(1, u);
        for(int x : component)
           comp[x] = res.size();
        res.emplace_back(component);
    }
    for(int i = 0; i < n; i++) {
        for(int v : G[0][i]) {
            if(comp[i] ==
               comp[v])
               continue;
            in[comp[v]] = true;
        }
    }
    int cnt = 0;
    for(int i = 0; i < res.size();
       i++) cnt += !in[i];
    return cnt;
}
```

1.7.15 Topological Sort

```
vector<int> toposort(int n, int m,
   vector<vector<int>> &G) {
    vector<int> in_degree(n, 0);
    for(int i = 0; i < n; i++) {
        for(int v : G[i])
           in_degree[v] += 1;
    }
    queue<int> Q;
    vector<int> res;
    for(int i = 0; i < n; i++) {
        if(in_degree[i] == 0) {
            Q.emplace(i);
        }
    }
    while(!Q.empty()) {
        int u = Q.front(); Q.pop();
        res.emplace_back(u);
        for(int v : G[u]) {
            in_degree[v] -= 1;
            if(in_degree[v] ==
               0) {
                Q.emplace(v);
            }
        }
    }
    return res.size() < n ?
       vector<int>() : res;
}
```

1.8 Misc

1.8.1 Coordinate Compress

```
// quita el 64 si solo necesitas enteros
mt19937_64
   rng(chrono::steady_clock::now().time_since_epoch().count());

vector<int> d = a;
sort(d.begin(), d.end());
d.resize(unique(d.begin(), d.end()) -
   d.begin());
for (int i = 0; i < n; ++i) {
  a[i] = lower_bound(d.begin(), d.end(),
     a[i]) - d.begin();
}
//original value of a[i] can be obtained
   through d[a[i]]
```

1.8.2 GPHash

```
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

struct custom_hash {
  static uint64_t splitmix64(uint64_t x)
     {
    x += 0x9e3779b97f4a7c15;
    x = (x ^ (x >> 30)) *
       0xbf58476d1ce4e5b9;
    x = (x ^ (x >> 27)) *
       0x94d049bb133111eb;
    return x ^ (x >> 31);
  }
  size_t operator()(uint64_t x) const {
    static const uint64_t FIXED_RANDOM =
       chrono::steady_clock::now().time_since_e
    return splitmix64(x + FIXED_RANDOM);
  }
};

gp_hash_table<int, int, custom_hash> mp;
```

1.8.3 Larsch

```cpp
#include <functional>
#include <memory>
#include <vector>

template <class T> class larsch {
  struct reduce_row;
  struct reduce_col;

  struct reduce_row {
    int n;
    std::function<T(int, int)> f;
    int cur_row;
    int state;
    std::unique_ptr<reduce_col> rec;

    reduce_row(int n_) : n(n_), f(),
        cur_row(0), state(0), rec() {
      const int m = n / 2;
      if (m != 0) {
        rec =
            std::make_unique<reduce_col>(m);
      }
    }

    void set_f(std::function<T(int,
        int)> f_) {
      f = f_;
      if (rec) {
        rec->set_f([&](int i, int j) -> T
            { return f(2 * i + 1, j); });
      }
    }

    int get_argmin() {
      const int cur_row_ = cur_row;
      cur_row += 1;
      if (cur_row_ % 2 == 0) {
        const int prev_argmin = state;
        const int next_argmin = [&]() {
          if (cur_row_ + 1 == n) {
            return n - 1;
          } else {
            return rec->get_argmin();
          }
        }();
        state = next_argmin;
        int ret = prev_argmin;
        for (int j = prev_argmin + 1; j
            <= next_argmin; j += 1) {
          if (f(cur_row_, ret) >
              f(cur_row_, j)) {
            ret = j;
          }
        }
        return ret;
      } else {
        if (f(cur_row_, state) <=
            f(cur_row_, cur_row_)) {
          return state;
        } else {
          return cur_row_;
        }
      }
    }
  };

  struct reduce_col {
    int n;
    std::function<T(int, int)> f;
    int cur_row;
    std::vector<int> cols;
    reduce_row rec;

    reduce_col(int n_) : n(n_), f(),
        cur_row(0), cols(), rec(n) {}

    void set_f(std::function<T(int,
        int)> f_) {
      f = f_;
      rec.set_f([&](int i, int j) -> T {
          return f(i, cols[j]); });
    }

    int get_argmin() {
      const int cur_row_ = cur_row;
      cur_row += 1;
      const auto cs = [&]() ->
          std::vector<int> {
        if (cur_row_ == 0) {
          return {{0}};
        } else {
          return {{2 * cur_row_ - 1, 2 *
              cur_row_}};
        }
      }();
      for (const int j : cs) {
        while ([&]() {
          const int size = cols.size();
          return size != cur_row_ &&
              f(size - 1, cols.back()) >
              f(size - 1, j);
        }()) {
          cols.pop_back();
```

```
      }
      if (cols.size() != n) {
        cols.push_back(j);
      }
    }
    return cols[rec.get_argmin()];
  }
};

std::unique_ptr<reduce_row> base;

public:
  larsch(int n, std::function<T(int,
      int)> f)
      :
        base(std::make_unique<reduce_row>(n))
        {
    base->set_f(f);
  }

  int get_argmin() { return
      base->get_argmin(); }
};
```

1.8.4 LIS

```
int lis(vector<int> const& a) {
    int n = a.size();
    const int INF = 1e9;
    vector<int> d(n+1, INF);
    d[0] = -INF;

    for (int i = 0; i < n; i++) {
        int l = upper_bound(d.begin(),
            d.end(), a[i]) - d.begin();
        if (d[l-1] < a[i] && a[i] < d[l])
            d[l] = a[i];
    }

    int ans = 0;
    for (int l = 0; l <= n; l++) {
        if (d[l] < INF)
            ans = l;
    }
    return ans;
}
```

1.9 Number theory

1.9.1 Chinese Remainder Theorem

```
/*
Encuentra un x tal que para cada i : x
    es congruente con A_i mod M_i
Devuelve {x, lcm}, donde x es la
    solucion con modulo lcm (lcm =
    LCM(M_0, M_1, ...)). Dado un k : x +
    k*lcm es solucion tambien.
Si la solucion no existe o la entrada no
    es valida devuelve {-1, -1}
Agregar Extended Euclides.
*/

pair<int, int> crt(vector<int> A,
    vector<int> M) {
    int n = A.size(), ans = A[0], lcm =
        M[0];
    for (int i = 1; i < n; i++) {
        int d = euclid(lcm, M[i]);
        if ((A[i] - ans) % d) return {-1,
            -1};
        int mod = lcm / d * M[i];
        ans = (ans + x * (A[i] - ans) / d
            % (M[i] / d) * lcm) % mod;
        if (ans < 0) ans += mod;
        lcm = mod;
    }
    return {ans, lcm};
}
```

1.9.2 Extended Euclidean

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

// Iterativo
int gcd(int a, int b, int& x, int& y) {
    x = 1, y = 0;
    int x1 = 0, y1 = 1, a1 = a, b1 = b;
    while (b1) {
```

```
        int q = a1 / b1;
        tie(x, x1) = make_tuple(x1, x - q
            * x1);
        tie(y, y1) = make_tuple(y1, y - q
            * y1);
        tie(a1, b1) = make_tuple(b1, a1 -
            q * b1);
    }
    return a1;
}
```

```
// UFPS
// El algoritmo de Euclides extendido
    retorna el gcd(a, b) y calcula los
    coeficientes enteros X y Y que
    satisfacen la ecuacion: a*X + b*Y =
    gcd(a, b).
int x, y;
/// O(log(max(a, b)))
int euclid(int a, int b) {
    if(b == 0) { x = 1; y = 0; return a;
        }
    int d = euclid(b, a%b);
    int aux = x;
    x = y;
    y = aux - a/b*y;
    return d;
}
```

1.9.3 Lineal Sieve

```
const int N = 100000000 + 5;
vector<int> primes;
bitset<N> composite;

void lineal(int n){
    for(int i = 2; i <= n; i++) {
            if(not composite[i])
                primes.emplace_back(i);
            for(int p : primes) {
                if(i * p > n)
                    break;
                composite[i * p] =
                    true;
                if(i % p == 0)
                    break;
            }
        }
}
```

1.9.4 Miller Rabin

```
// El algoritmo de Miller-Rabin
    determina si un numero es primo o no.
    Agregar Modular Exponentiation (para
    m ll) y Modular Multiplication.

/// O(log^3(n))
bool test(ll n, int a) {
    if (n == a) return true;
    ll s = 0, d = n-1;
    while (d%2 == 0) s++, d /= 2;
    ll x = expmod(a, d, n);
    if (x == 1 || x+1 == n) return true;
    for (int i = 0; i < s-1; i++) {
        x = mulmod(x, x, n);
        if (x == 1) return false;
        if (x+1 == n) return true;
    }
    return false;
}

bool is_prime(ll n) {
    if (n == 1) return false;
    int ar[] = {2,3,5,7,11,13,17,19,23};
    for (auto &p : ar) if (!test(n, p))
        return false;
    return true;
}
```

1.9.5 Mobius

```
/*
La funcion mu de Mobius devuelve 0 si n
    es divisible por algun cuadrado (x^2).
Si n es libre de cuadrados entonces
    devuelve 1 o -1 si n tiene un numero
    par o impar de factores primos
    distintos.
* Calcular Mobius para todos los numeros
    menores o iguales a MX con Sieve of
    Eratosthenes.
*/
const int MX = 1e6;
short mu[MX+1] = {0, 1};
///  O(MX log(log(MX)))
void mobius() {
    for (int i = 1; i <= MX; i++) {
        if (!mu[i]) continue;
        for (int j = i*2; j <= MX; j +=
            i) {
```

```
            mu[j] -= mu[i];
        }
    }
}
```

1.9.6 Modular Multiplication

```
/* Calcula (a*b) % m sin overflow cuando
   m es ll. */

/// O(1)
ll mulmod(ll a, ll b, ll m) {
    ll r = a*b-(ll)((long
        double)a*b/m+.5)*m;
    return r < 0 ? r+m : r;
}
```

1.9.7 Natural Sieve

```
const int N = 100000000 + 5;
bitset<N> composite;

void natural(int n){
    // (WARNING) Todos los pares son
        primos
    for(int i = 3; i * i <= n; i += 2) {
            if(not composite[i]) {
                for(int j = i * i;
                    j <= n; j += i)
                    composite[j] =
                    true;
            }
        }
}
```

1.9.8 Pollard Rho

```
/*
La funcion Rho de Pollard calcula un
    divisor no trivial de n. Agregar
    Modular Multiplication.
*/
ll gcd(ll a, ll b) { return a ? gcd(b%a,
    a) : b; }

ll rho(ll n) {
    if (!(n&1)) return 2;
    ll x = 2, y = 2, d = 1;
    ll c = rand() % n + 1;
    while (d == 1) {
        x = (mulmod(x, x, n) + c) % n;
        y = (mulmod(y, y, n) + c) % n;
        y = (mulmod(y, y, n) + c) % n;
        d = gcd(abs(x-y), n);
    }
    return d == n ? rho(n) : d;
}

* Version optimizada

ll add(ll a, ll b, ll m) { return (a +=
    b) < m ? a : a-m; }

ll rho(ll n) {
    static ll s[MX];
    while (1) {
        ll x = rand()%n, y = x, c =
            rand()%n;
        ll *px = s, *py = s, v = 0, p = 1;
        while (1) {
            *py++ = y = add(mulmod(y, y,
                n), c, n);
            *py++ = y = add(mulmod(y, y,
                n), c, n);
            if ((x = *px++) == y) break;
            ll t = p;
            p = mulmod(p, abs(y-x), n);
            if (!p) return gcd(t, n);
            if (++v == 26) {
                if ((p = gcd(p, n)) > 1 &&
                    p < n) return p;
                v = 0;
            }
        }
        if (v && (p = gcd(p, n)) > 1 && p
            < n) return p;
    }
}
```

1.10 Strings

1.10.1 Aho Corasick 2

```
const int N = 1e6 + 10;
const int sigma = 30;
int term[N], suflink[N], trie[N][sigma];
vi tree[N];
bool vis[N], ans[N];
int sz = 0;
```

```
// dsu
int par[N];
int get(int a) {return a == par[a] ? a :
    par[a] = get(par[a]); }
void unite(int a, int b) {
    a = get(a); b = get(b);
    if(a == b) return;
    par[a] = b;
}

void add_trie(const string &s, int id) {
    int node = 0;
    for(char c : s) {
        int now = c - 'a';
        if(!trie[node][now])
            trie[node][now] = ++sz;
        int last = node;
                node = trie[node][now];
    }
    if(term[node]) unite(term[node], id);
    else term[node] = id;
}

void BFS(int src) {
    queue<int> q;
        q.push(src);
    while(q.size()) {
        int v = q.front(); q.pop();
        int u = suflink[v];
        if(v) tree[u].push_back(v);
        for(int c = 0; c < sigma; c++) {
            if(trie[v][c]) {
                        suflink[trie[v][c]]
                            = (v ==
                            0 ? 0 :
                            trie[u][c]);
                        q.push(trie[v][c]);
            }else trie[v][c] = (v == 0 ?
                0 : trie[u][c]);
        }
    }
}

bool DFS(int src) {
    bool exists = vis[src];
    for(int u : tree[src]) exists |=
        DFS(u);
    return ans[get(term[src])] = exists;
}
```

1.10.2 Aho Corasick

```
/*
    El trie (o prefix tree) guarda un
        diccionario de strings como un
        arbol enraizado.
    Aho corasick permite encontrar las
        ocurrencias de todos los strings
        del trie en un string s.
*/
const int alpha = 26; //cantidad de
    letras del lenguaje
const char L = 'a'; //primera letra del
    lenguaje

struct node {
    int next[alpha], end;
    //int link, exit, cnt; //para aho
        corasick
    int& operator[](int i) { return
        next[i]; }
};

vector<node> trie = {node()};

void add_str(string &s, int id = 1) {
    int u = 0;
    for (auto ch : s) {
        int c = ch-L;
        if (!trie[u][c]) {
            trie[u][c] = trie.size();
            trie.push_back(node());
        }
        u = trie[u][c];
    }
    trie[u].end = id; //con id > 0
    //trie[u].cnt++; //para aho corasick
}

// aho corasick
void build_ac() {
    queue<int> q; q.push(0);
    while (q.size()) {
        int u = q.front(); q.pop();
        for (int c = 0; c < alpha; ++c) {
            int v = trie[u][c];
            if (!v) trie[u][c] =
                trie[trie[u].link][c];
            else q.push(v);
            if (!u || !v) continue;
            trie[v].link =
                trie[trie[u].link][c];
                    trie[v].exit =
                        trie[trie[v].link].end
```

```
                        ? trie[v].link
                        :
                        trie[trie[v].link].exit;
            trie[v].cnt +=
               trie[trie[v].link].cnt;
         }
      }
}

vector<int> cnt; //cantidad de
   ocurrencias en s para cada patron

void run_ac(string &s) {
   int u = 0, sz = s.size();
   for (int i = 0; i < sz; ++i) {
      int c = s[i]-L;
      while (u && !trie[u][c]) u =
         trie[u].link;
      u = trie[u][c];
      int x = u;
      while (x) {
         int id = trie[x].end;
         if (id) cnt[id-1]++;
         x = trie[x].exit;
      }
   }
}
```

1.10.3 CP Algo SA

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

typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;
typedef vector<vector<int> > vvi;
typedef vector<pii> vii;

template <typename T>
inline T gcd(T a, T b) { while (b != 0)
   swap(b, a %= b); return a; }

const int MAX_LEN = 500000 + 10;
struct state{
      int len = 0;
      int link = 0;
      map<char, int> nxt;
};

state st[MAX_LEN * 2];
int sz = 0, last = 0;

void SA_init() {
      st[0].len = 0;
      st[0].link = -1;
      sz++;
      last = 0;
}

void SA_extend(char c) {
      int cur = sz++;
      st[cur].len = st[last].len + 1;
      int p = last;
      while(p != -1 &&
         !st[p].nxt.count(c)) {
            st[p].nxt[c] = cur;
            p = st[p].link;
      }
      if(p == -1) {
            st[cur].link = 0;
      }else{
            int q = st[p].nxt[c];
            if(st[p].len + 1 ==
               st[q].len) {
                  st[cur].link = q;
            }else{
                  int clone = sz++;
                  st[clone].len =
                     st[p].len + 1;
                  st[clone].nxt =
                     st[q].nxt;
                  st[clone].link =
                     st[q].link;
                  while(p != -1 &&
                     st[p].nxt[c] ==
                     q) {
                        st[p].nxt[c]
                           = clone;
                        p =
                           st[p].link;
                  }
                  st[q].link =
                     st[cur].link =
                     clone;
            }
      }
      last = cur;
}

bool exists(const string& s) {
      int cur = 0;
      for(char c : s) {
```

```
            if(st[cur].nxt.count(c))
                cur = st[cur].nxt[c];
            else return false;
        }
        return true;
}

signed main() {
        ios_base::sync_with_stdio(false);
        cin.tie(0);

        string t; cin >> t;
        SA_init();
        for(char c : t) SA_extend(c);
        int qq; cin >> qq;
        while(qq--) {
                string s; cin >> s;
                if(exists(s)) cout <<
                    "YES\n";
                else cout << "NO\n";
        }
}
```

1.10.4 Hashing

```
inline int add(int a, int b, const int
    &mod) { return a+b >= mod ? a+b-mod :
    a+b; }
inline int sbt(int a, int b, const int
    &mod) { return a-b < 0 ? a-b+mod :
    a-b; }
inline int mul(int a, int b, const int
    &mod) { return 1ll*a*b % mod; }

const int X[] = {257, 359}; // 31 43
const int MOD[] = {(int)1e9+7,
    (int)1e9+9};
const int N = 1e5 + 10;
int pows[N][2], ipows[N][2];
int h[2];

int binpow(int a, int exp, const int
    &mod) {
    int res = 1;
    while(exp > 0) {
        if(exp % 2) res = mul(res, a,
            mod);
        a = mul(a, a, mod);
        exp >>= 1;
    }
    return res;
}

struct Hashing {
    string s;
    int n;
    vvi ph;

    Hashing(string &s) : s(s) {
        n = s.size();
        ph.assign(n, vi(2));
    }

    void build() {
        for(int j = 0; j < 2; j++) {
            ph[0][j] = s[0];
            for(int i = 1; i < n; i++) {
                ph[i][j] = add(ph[i -
                    1][j], mul(pows[i][j],
                    s[i], MOD[j]), MOD[j]);
            }
        }
    }

    pii substr_hash(int l, int r) {
        if(l == 0) return
            make_pair(ph[r][0], ph[r][1]);
        h[0] = mul(sbt(ph[r][0], ph[l -
            1][0], MOD[0]), ipows[l][0],
            MOD[0]);
        h[1] = mul(sbt(ph[r][1], ph[l -
            1][1], MOD[1]), ipows[l][1],
            MOD[1]);
        return make_pair(h[0], h[1]);
    }
};

void init() {
    for(int j = 0; j < 2; j++) {
        pows[0][j] = 1;
        for(int i = 1; i < N; i++)
            pows[i][j] = mul(pows[i -
            1][j], X[j], MOD[j]);
        ipows[N - 1][j] = binpow(pows[N -
            1][j], MOD[j] - 2, MOD[j]);
        for(int i = N - 1; i > 0; i--)
            ipows[i - 1][j] =
            mul(ipows[i][j], X[j],
            MOD[j]);
    }
}
```

1.10.5 Manacher

```
// para verificar si un substring es palindromo
// return pal[l + r] >= (r - l + 1) + 1; indexando en 0

vi manacher_odd(string s) {
    int n = s.size();
    s = "@" + s + "$";
    vi len(n + 1);
    int l = 1, r = 1;
    for(int i = 1; i <= n; i++) {
        len[i] = min(r - i, len[l + (r - i)]);
        while(s[i - len[i]] == s[i + len[i]]) len[i]++;
        if(i + len[i] > r) {
            l = i - len[i];
            r = i + len[i];
        }
    }
    len.erase(begin(len));
    return len;
}

vi manacher(string s) {
    string ns(1, '#');
    for(char c : s) {
        ns.push_back(c);
        ns.push_back('#');
    }
    auto res = manacher_odd(ns);
    return vi(res.begin() + 1, res.end() - 1);
}
```

1.10.6 Prefix Function

```
vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
```

1.10.7 Suffix Array

```
struct SuffixArray {
        vi sa, lcp;
    string s;
        SuffixArray(string& s_, int lim=256) : s(s_) { // or basic_string<int>
        int n = s.size() + 1, k = 0, a, b;
                vi x(s.begin(), s.end()), y(n), ws(max(n, lim)), rank(n);
                x.push_back(0), sa = lcp = y, iota(sa.begin(), sa.end(), 0);
    s.push_back('$');
            for(int j = 0, p = 0; p < n; j = max(1, j * 2), lim = p) {
                p = j, iota(y.begin(), y.end(), n - j);
                for(int i = 0; i < n; i++) if (sa[i] >= j) y[p++] = sa[i] - j;
                fill(ws.begin(), ws.end(), 0);
                for(int i = 0; i < n; i++) ws[x[i]]++;
                for(int i = 1; i < lim; i++) ws[i] += ws[i - 1];
                for (int i = n; i--;) sa[--ws[x[y[i]]]] = y[i];
                swap(x, y), p = 1, x[sa[0]] = 0;
                for(int i = 1; i < n; i++) {
            a = sa[i - 1];
            b = sa[i];
            x[b] = (y[a] == y[b] && y[a + j] == y[b + j]) ? p - 1 : p++;
        }
```

```
        }
        for(int i = 1; i < n; i++)
            rank[sa[i]] = i;
        for (int i = 0, j; i < n -
            1; lcp[rank[i++]] = k)
                for (k && k--, j =
                    sa[rank[i] - 1];
                        s[i
                            +
                            k]
                            ==
                            s[j
                            +
                            k];
                            k++);
    }

    //Longest Common Substring:
        construir el suffixArray s = s1 +
        "#" + s2 + "$" y m = s2.size()
    // pair<int, int> lcs() {
    //     int mx = -1, ind = -1;
    //     for (int i = 1; i < n; i++) {
    //         if (((sa[i] < n-m-1) !=
        (sa[i-1] < n-m-1)) && mx <
        lcp[i]) {
    //             mx = lcp[i]; ind = i;
    //         }
    //     }
    //     return {mx, ind};
    // }

};
```

1.10.8 Suffix Automaton Racso

```
struct SuffixAutomaton {
        int nodes;
        vector<int> link; // suffix link
        vector<int> len; // max length of
            the state
        vector<int> firstpos; // last
            position of first occurrence
            of state
        vector<vector<int>> nxt; //
            transitions
        vector<bool> is_clone; // clone
            attribute (for counting)

        SuffixAutomaton() {
                len.emplace_back(0);
                link.emplace_back(-1);
                nxt.emplace_back(vector<int>(26,
                    0));
                firstpos.emplace_back(-1);
                is_clone.emplace_back(false);
                nodes = 1;
        }

        void add_node(int new_len, int
            new_link, int new_fp, bool
            new_clone) {
                len.emplace_back(new_len);
                link.emplace_back(new_link);
                nxt.emplace_back(vector<int>(26,
                    0));
                firstpos.emplace_back(new_fp);
                is_clone.emplace_back(new_clone);
        }

        int add(int p, int c) {
                auto getNxt = [&] () {
                        if (p == -1)
                            return 0;
                        int q = nxt[p][c];
                        if (len[p] + 1 ==
                            len[q]) return
                            q;
                        int clone =
                            nodes++;
                        add_node(len[p] +
                            1, link[q],
                            firstpos[q],
                            true);
                        nxt[nodes - 1] =
                            nxt[q];
                        link[q] = clone;
                        while(~p and
                            nxt[p][c] == q)
                            {
                                nxt[p][c] =
                                    clone;
                                p = link[p];
                        }
                        return clone;
                };
                // if (nxt[p][c]) return
                    getNxt();
                // ^ need if adding > 1
                    string
                int cur = nodes++; // make
                    new state
                add_node(len[p] + 1, -1,
                    firstpos[p] + 1,
```

```
                false);
            while(~p and !nxt[p][c]) {
                nxt[p][c] = cur;
                p = link[p];
            }
            int x = getNxt();
            link[cur] = x;
            return cur;
        }

    void init(string s) { // add
        string to automaton
            int p = 0;
            for(auto c : s) {
                p = add(p, c -
                    'a');
            }
    }

void prepro() {
    vector<int> topo_order(nodes);
    iota(topo_order.begin(),
        topo_order.end(), 0);
    sort(topo_order.begin(),
        topo_order.end(), [&](int a,
        int b) {
        return len[a] > len[b];
    });

    cnt.assign(nodes, 0);
    for (int u : topo_order) {
        for (int c = 0; c < 26; c++) {
            int v = nxt[u][c];
            if (v) cnt[u] += cnt[v];
        }
                cnt[u]++;
            }
        }

        string get_kth(int k) {
            int u = 0;
            string result = "";

            while (k > 0) {
                for (int c = 0; c < 26; c++) {
                    if (nxt[u][c]) {
                        int v = nxt[u][c];
                        if (cnt[v] >= k) {
                            result += char(c +
                                'a');
                            k--;
                            u = v;
                            break;
                        } else {
                            k -= cnt[v];
                        }
                    }
                }
            }

            return result;
        }
};
```

1.10.9 Suffix Automaton

```
struct suffixAutomaton {
    struct node {
        int len, link; bool end;
        map<char, int> next;
        int cnt; ll in, out;
    };

    vector<node> sa;
    int last; ll substrs = 0;

    suffixAutomaton() {}
    suffixAutomaton(string &s) {
        sa.reserve(s.size()*2);
        last = add_node();
        sa[0].link = -1;
        sa[0].in = 1;
        for (char &c : s) add_char(c);
        for (int p = last; p; p =
            sa[p].link) sa[p].end = 1;
    }

    int add_node() { sa.pb({}); return
        sa.size()-1; }

    void add_char(char c) {
        int u = add_node(), p = last;
        sa[u].len = sa[last].len + 1;
        while (p != -1 &&
            !sa[p].next.count(c)) {
            sa[p].next[c] = u;
            sa[u].in += sa[p].in;
            substrs += sa[p].in;
            p = sa[p].link;
        }
        if (p != -1) {
            int q = sa[p].next[c];
            if (sa[p].len + 1 !=
                sa[q].len) {
```

```
            int clone = add_node();
            sa[clone] = sa[q];
            sa[clone].len = sa[p].len
                + 1;
            sa[clone].in = 0;
            sa[q].link = sa[u].link =
                clone;
            while (p != -1 &&
                sa[p].next[c] == q) {
                sa[p].next[c] = clone;
                sa[q].in -= sa[p].in;
                sa[clone].in +=
                    sa[p].in;
                p = sa[p].link;
            }
        } else sa[u].link = q;
    }
    last = u;
}

void run(string &s) {
    int u = 0;
    for (int i = 0; i < s.size();
        ++i) {
        while (u &&
            !sa[u].next.count(s[i])) u
            = sa[u].link;
        if (sa[u].next.count(s[i])) u
            = sa[u].next[s[i]];
    }
}

int match_str(string &s) {
    int u = 0, n = s.size();
    for (int i = 0; i < n; ++i) {
            if (!sa[u].next.count(s[i]))
                return 0;
            u = sa[u].next[s[i]];
        }
        return count_occ(u);
    }

    int count_occ(int u) {
        if (sa[u].cnt != 0) return
            sa[u].cnt;
        sa[u].cnt = sa[u].end;
        for (auto &v : sa[u].next)
            sa[u].cnt += count_occ(v.S);
        return sa[u].cnt;
    }

    ll count_paths(int u) {
        if (sa[u].out != 0) return
            sa[u].out;
        for (auto &v : sa[u].next)
            sa[u].out += count_paths(v.S)
                + 1;
        return sa[u].out;
    }

    node& operator[](int i) { return
        sa[i]; }
};
```

1.10.10 Trie

```
const int N = 1e6 + 100;
int trie[N][26]; // N = suma de
    longitudes
bool stop[N];
int ct = 0;

void insert(string word) {
        int node = 0;
        for(int i = 0; i <
            (int)word.size(); i++) {
                if(!trie[node][word[i] -
                    'a'])
                    trie[node][word[i] -
                    'a'] = ++ct;
                node = trie[node][word[i]
                    - 'a'];
        }
        stop[node] = true;
}
```

1.10.11 Z Function

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

```
    }
}
```

```
    return z;
}
```