Table of Index:

Binary Exponentiation:

```
typedef long long ll;
ll power(ll a, ll b, ll m) {
    ll ans = 1;
    a %= m;
    while (b > 0) {
      if (b & 1) ans = (ans * a) % m;
      a = (a * a) % m;
      b >>= 1;
    }
    return ans;
}
```

If m is prime, calculate only $x^{n \bmod (m-1)}$ instead of x^n, which follows from Fermat's little theorem.

Modulo Inverse:

```
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;
}
int modInverse(int a, int m) {
    int x, y;
    int g = gcd(a, m, x, y);
    while (x < 0) x += m;
    return x;
}
```

Linear Diophantine Equation:

```
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;
}

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

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

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

int find_all_solutions(int a, int b,
int c, int minx, int maxx, int miny,
int maxy) {

int x, y, g;
if (!find_any_solution(a, b, c, x, y,
g))
return 0; a /= g;
      a /= g;
int sign_a = a > 0 ? +1 : -1; int
sign_b = b > 0 ? +1 : -1;

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

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

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

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

if (lx2 > rx2) swap(lx2, rx2);

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

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

Linear Sieve of Eratosthenes:

```
// space optimized one is faster though
const int MX = 1e7 + 5;
bitset<MX> isPrime;
vector<int> primes;

void sieve(int n) {
    isPrime.set();
```

```
    isPrime[0] = isPrime[1] = false;
    for (int i = 4; i <= n; i += 2)
isPrime[i] = false;
    for (int i = 3; i * i <= n; i += 2)
{
        if (!isPrime[i]) continue;
        for (int j = i * i; j <= n; j
+= i) isPrime[j] = false;
    }
    primes.push_back(2);
    for (int i = 3; i <= n; i += 2) {
        if (isPrime[i])
Primes.push_back(i);
    }
}
```

Segmented Sieve:

```
vector<char> segmentedSieve(long long
L, long long R) {
    // generate all primes up to
sqrt(R)
    long long lim = sqrt(R);
    vector<char> mark(lim + 1, false);
    vector<long long> primes;
    for (long long i = 2; i <= lim;
++i) {
        if (!mark[i]) {
            primes.emplace_back(i);
            for (long long j = i * i; j
<= lim; j += i) mark[j] = true;
        }
    }
    vector<char> isPrime(R - L + 1,
true);
    // 0 denotes L, 1 denotes L + 1, so
on...
    for (long long i : primes)
        for (long long j = max(i * i,
(L + i - 1) / i * i); j <= R; j += i)
            isPrime[j - L] = false;
    if (L == 1) isPrime[0] = false;
    return isPrime;
}
```

Memory Optimized Block Sieve:

Uses O(S + √n) memory only

```
vector<int> prime_generated;
int count_primes(int n) {
    const int S = 10005;
    vector<int> primes;
    int nsqrt = sqrt(n);
    vector<bool> is_prime(nsqrt + 2,
true);

    for (int i = 2; i <= nsqrt; i++) {
        if (!is_prime[i]) continue;
        primes.push_back(i);
        for (int j = i * i; j <= nsqrt;
j += i)
            is_prime[j] = false;
    }

    int result = 0;
    vector<bool> block(S);

    for (int k = 0; k * S <= n; k++) {
        fill(block.begin(),
block.end(), true);
        int start = k * S;
        for (int p : primes) {
            int start_idx = (start + p
- 1) / p;
            int j = max(start_idx, p) *
p - start;
            for (; j < S; j += p)
block[j] = false;
        }

     if (k == 0) block[0] = block[1] =
false;
     for (int i = 0; i < S && start + i
<= n; i++) {
     if (block[i]) {
      result++;
      prime_generated.push_back(start +
      i);
      }
    }
  }
  return result;
}
```

Euler Phi:

Φ(n) = count of coprime numbers ≤ n

$a^b mod\ m = a^{b\ mod\ \Phi(m)}\ mod\ m, m\ and\ a\ coprime$

```
int eulerPhi(int n) {
    int res = n;
    int sqrtn = sqrt(n);
    for (int i = 0; i < Primes.size()
&& Primes[i] <= sqrtn; i++) {
        if (n % Primes[i] == 0) {
            while (n % Primes[i] == 0)
n /= Primes[i];
            sqrtn = sqrt(n);
            res /= Primes[i];
            res *= (Primes[i] - 1);
        }
    }
    if (n != 1) {
        res /= n;
        res *= n - 1;
    }
    return res;
}
```

Miller Rabin Primality Test:

Determinisitic test, guaranteed for upto $7 . 10^{18}$, complexity: 7 times the complexity of a^b mod c.

```
typedef unsigned long long ull;
typedef long long ll;
const ll mod = 1000000007;
ll modpow(ll b, ll e) {
    ll ans = 1;
    for (; e; b = b * b % mod, e /= 2)
        if (e & 1) ans = ans * b % mod;
    return ans;
}

ull modmul(ull a, ull b, ull M) {
    ll ret = a * b - M * ull(1.L / M *
a * b);
    return ret + M * (ret < 0) - M *
(ret >= (ll)M);
}
ull modpow(ull b, ull e, ull mod) {
```

```
    ull ans = 1;
    for (; e; b = modmul(b, b, mod), e /= 2)
        if (e & 1) ans = modmul(ans, b, mod);
    return ans;
}

bool isPrime(ull n) {
    if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
    ull A[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022},
        s = __builtin_ctzll(n - 1), d = n >> s;
    for (ull& a : A) {  // ^ count trailing zeroes
        ull p = modpow(a % n, d, n), i = s;
        while (p != 1 && p != n - 1 && a % n && i--) p = modmul(p, p, n);
        if (p != n - 1 && i != s) return false;
    }
    return true;
}
```

Pollard's Rho Algorithm for Prime Factorization:

Solves UVA – 11476 (Prime Factorization of 800 numbers upto 10^{16})

/*
Range: 10^18 (tested), should be okay up to 2^63-1
miller_rabin(n): returns 1 if prime, 0 otherwise
Magic bases:
n < 4,759,123,141 3 : 2, 7, 61
n < 1,122,004,669,633 4 : 2, 13, 23, 1662803
n < 3,474,749,660,383 6 : 2, 3, 5, 7, 11, 13
n < 2^64 7 : 2, 325, 9375, 28178, 450775, 9780504, 1795265022
Identifies 70000 18 digit primes in 1 second on Toph
pollard_rho(n):
If n is prime, returns n, otherwise returns a proper divisor of n
Able to factorize ~120 18 digit semiprimes in 1 second on Toph
Able to factorize ~700 15 digit semiprimes in 1 second on Toph
Note: for factorizing large number, do trial division upto cubic root and then call pollard rho once.
*/

```
#include <bits/stdc++.h>
#define LL long long
using namespace std;
LL mult(LL a, LL b, LL mod) {
    assert(b < mod && a < mod);
    long double x = a;
    uint64_t c = x * b / mod;
    int64_t r = (int64_t)(a * b - c * mod) % (int64_t)mod;
    return r < 0 ? r + mod : r;
}
LL power(LL x, LL p, LL mod) {
    LL s = 1, m = x;
    while (p) {
        if (p & 1) s = mult(s, m, mod);
        p >>= 1;
        m = mult(m, m, mod);
    }
    return s;
}
bool witness(LL a, LL n, LL u, int t) {
    LL x = power(a, u, n);
    for (int i = 0; i < t; i++) {
        LL nx = mult(x, x, n);
        if (nx == 1 && x != 1 && x != n - 1) return 1;
        x = nx;
    }
    return x != 1;
}
vector<LL> bases = {2, 325, 9375, 28178, 450775, 9780504, 1795265022};
bool miller_rabin(LL n) {
    if (n < 2) return 0;
    if (n % 2 == 0) return n == 2;
    LL u = n - 1;
    int t = 0;
    while (u % 2 == 0) u /= 2, t++;
    // n-1 = u*2^t
    for (LL v : bases) {
        LL a = v % (n - 1) + 1;
        if (witness(a, n, u, t)) return 0;
    }
    return 1;
}
LL gcd(LL u, LL v) {
    if (u == 0) return v;
    if (v == 0) return u;
    int shift = __builtin_ctzll(u | v);
    u >>= __builtin_ctzll(u);
    do {
        v >>= __builtin_ctz(v);
        if (u > v) swap(u, v);
        v = v - u;
    } while (v);
    return u << shift;
}
mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
LL pollard_rho(LL n) {
    if (n == 1) return 1;
    if (n % 2 == 0) return 2;
    if (miller_rabin(n)) return n;
    while (true) {
        LL x = uniform_int_distribution<LL>(1, n - 1)(rng);
        LL y = 2, res = 1;
        for (int sz = 2; res == 1; sz *= 2) {
            for (int i = 0; i < sz && res <= 1; i++) {
                x = mult(x, x, n) + 1;
                res = gcd(abs(x - y), n);
            }
            y = x;
        }
        if (res != 0 && res != n) return res;
    }
}
const int MX = 2.2e5 + 7;
```

```
vector<int> primes;
bool isp[MX];
void sieve() {
    fill(isp + 2, isp + MX, 1);
    for (int i = 2; i < MX; i++)
        if (isp[i]) {
            primes.push_back(i);
            for (int j = 2 * i; j < MX;
j += i) isp[j] = 0;
        }
}
vector<LL> factorize(LL x) {
    vector<LL> ans;
    for (int p : primes) {
        if (1LL * p * p * p > x) break;
        while (x % p == 0) {
            x /= p;
            ans.push_back(p);
        }
    }
    if (x > 1) {
        LL z = pollard_rho(x);
        ans.push_back(z);
        if (z < x) ans.push_back(x /
z);
    }
    return ans;
}
int main() {
    sieve();
    int t;
    cin >> t;
    while (t--) {
        long long x;
        if (!(cin >> x)) break;
        vector<LL> ans = factorize(x);
        sort(ans.begin(), ans.end());
        vector<pair<LL, int>> ff;
        for (LL x : ans) {
            if (ff.size() &&
ff.back().first == x)
                ff.back().second++;
            else
                ff.push_back({x, 1});
        }
        cout << x << " =";
        bool first = true;
        for (auto pr : ff) {
            if (!first) cout << " *";
            first = false;
            cout << " " << pr.first;
            if (pr.second > 1) cout <<
"^" << pr.second;
        }
        cout << endl;
    }
}
```

Matrix Exponentiation (with Fibonacci Finding):

Random example of a recurrence relation:

$$F_n = 2F_{n-1} + 3F_{n-4} + 1$$

$$\begin{bmatrix} F_4 \\ F_3 \\ F_2 \\ F_1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 3 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} F_3 \\ F_2 \\ F_1 \\ F_0 \\ 1 \end{bmatrix}$$

```
const int MOD = 1e9 + 7;
struct Matrix {
    int r, c;
    vector<vector<int>> mat;
    Matrix(int r, int c) {
        this->r = r, this->c = c;
        mat.assign(r, vector<int>(c,
0));
    }
    Matrix(vector<vector<int>> &vec) {
        r = vec.size();
        c = vec[0].size();
        mat.assign(r, vector<int>(c,
0));
        for (int i = 0; i < r; i++) {
            for (int j = 0; j < c; j++)
mat[i][j] = vec[i][j];
        }
    }
};
istream &operator>>(istream &in, Matrix
&mt) {
    for (int i = 0; i < mt.r; i++)
        for (int j = 0; j < mt.c; j++)
in >> mt.mat[i][j];
    return in;
}
ostream &operator<<(ostream &out,
Matrix &mt) {
    for (int i = 0; i < mt.r; i++) {
        for (int j = 0; j < mt.c; j++)
out << mt.mat[i][j] << " ";
        out << endl;
    }
    return out;
}
Matrix mul(Matrix &a, Matrix &b) {
    // assert(a.mat[0].size() ==
b.mat.size());
    Matrix product(a.mat.size(),
b.mat[0].size());
    for (int j = 0; j < a.mat.size();
j++) {
        for (int i = 0; i <
b.mat.size(); i++) {
            for (int k = 0; k <
b.mat[0].size(); k++) {
                product.mat[j][k] =
(product.mat[j][k] + 0LL +

(a.mat[j][i] * 1LL * b.mat[i][k]) %
MOD) %

MOD;
            }
        }
    }
    return product;
}
Matrix power(Matrix a, ll b) {
    // assert(a.mat.size() ==
a.mat[0].size());
    Matrix ans(a.mat.size(),
a.mat.size());
    for (int i = 0; i < ans.mat.size();
i++) ans.mat[i][i] = 1;
    while (b) {
        if (b & 1) ans = mul(ans, a);
        a = mul(a, a);
```

```
        b >>= 1;
    }
    return ans;
}
```

Gaussian Elimination:

```
#include <bits/stdc++.h>
using namespace std;
const double EPS = 1e-9;

int Gauss(vector<vector<double>> a,
vector<double> &ans) {
    // a is the coefficient matrix, and
the last column is
    // added from the rhs of the
equations
    int n = (int)a.size(), m =
(int)a[0].size() - 1;
    vector<int> pos(m, -1);
    double det = 1;
    int rank = 0;
    for (int col = 0, row = 0; col < m
&& row < n; ++col) {
        int mx = row;
        for (int i = row; i < n; i++)
// for partial pivoting
            if (fabs(a[i][col]) >
fabs(a[mx][col])) mx = i;
        if (fabs(a[mx][col]) < EPS) {
            det = 0;
            continue;
        }
        for (int i = col; i <= m; i++)
swap(a[row][i], a[mx][i]);
        if (row != mx) det = -det;
        det *= a[row][col];
        pos[col] = row;
        for (int i = 0; i < n; i++) {
            if (i != row &&
fabs(a[i][col]) > EPS) {
                double c = a[i][col] /
a[row][col];
                for (int j = col; j <=
m; j++) a[i][j] -= a[row][j] * c;
            }
        }
        ++row;
        ++rank;
    }
    ans.assign(m, 0);
    for (int i = 0; i < m; i++) {
        if (pos[i] != -1) ans[i] =
a[pos[i]][m] / a[pos[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 (fabs(sum - a[i][m]) > EPS)
return -1;  // no solution
    }
    for (int i = 0; i < m; i++)
        if (pos[i] == -1) return 2;  //
infinte solutions
    return 1;                        //
unique solution
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<double>> v(n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= m; j++) {
            double x;
            cin >> x;
            v[i].push_back(x);
        }
    }
    vector<double> ans;
    int k = Gauss(v, ans);
    if (k)
        for (int i = 0; i < n; i++)
            cout << fixed <<
setprecision(5) << ans[i] << ' ';
    else
        cout << "no solution\n";
    return 0;
}
```

Sparse Table:

Only done on immutable arrays, can answer queries in O(1) for idempotent operations like min, max, gcd etc. i.e. operations that can be done multiple times without affecting the final result.

```
vector<vector<int>>
range_minimum(vector<int>& vec) {
    int n = vec.size();
    vector<vector<int>> sparse(n,
vector<int>(__lg(n) + 1, INT_MAX));
    for (int i = 0; i < n; i++)
sparse[i][0] = vec[i];
    for (int j = 1; j <= __lg(n); j++)
{
        for (int i = 0; i + (1 << j) -
1 < n; i++) {
            sparse[i][j] =
                min(sparse[i][j - 1],
sparse[i + (1 << (j - 1))][j - 1]);
        }
    }
    return sparse;
}
int min_query(vector<vector<int>>&
sparse, int l, int r) { // minimum of
[l, r]
    int len = r - l + 1;
    return min(sparse[l][__lg(len)],
sparse[r - (1 << (__lg(len))) +
1][__lg(len)]);
}
```

Binary Indexed Tree/Fenwick Tree:

Range query and update both in O(logn), used mostly for range sum (with updates), can be done on other reversible functions as well. Much less constant factors than segment tree.

```
typedef long long ll;

// 1-based indexing
// point-update, range-query
const int MX = 1e5 + 5;
vector<ll> BIT(MX), a(MX);
int n;
void add(int idx, int delta) {  //
increase idx'th element by delta
```

```
    for (; idx <= n; idx += idx & -idx)
BIT[idx] += delta;
}
ll query(int idx) {  // returns sum in
range [1, x]
    ll sum = 0;
    for (; idx > 0; idx -= idx & -idx)
sum += BIT[idx];
    return sum;
}
ll query(int l, int r) { return
query(r) - query(l - 1); }
```

2D Fenwick Tree:

sum(x1, y1, x2, y2) = sum(x2, y2) - sum(x1 - 1, y2) - sum(x2, y1 - 1) + sum(x1 - 1, y1 - 1)

```
const int mx = 1e3 + 5;
int r, c;
vector<vector<ll>> bit(r,
vector<ll>(c));
ll sum(int x, int y) {
    ll ret = 0;
    for (int i = x; i >= 0; i = (i & (i
+ 1)) - 1)
    for (int j = y; j >= 0; j = (j & (j
+ 1)) - 1) ret += bit[i][j];
    return ret;
}
void add(int x, int y, int delta) {
    for (int i = x; i < bit.size(); i =
i | (i + 1))
    for (int j = y; j < bit[i].size();
j = j | (j + 1)) bit[i][j] += delta;
}
```

Min Segment Tree:

```
const int MX = 2e5 + 5;
vector<int> numbers(MX);

struct seg_tree {
    vector<int> vec;
    const int NEUTRAL_ELEMENT =
INT_MAX;
    inline int lc(int x) { return (x <<
1); }        // left child
    inline int rc(int x) { return ((x
<< 1) | 1); }  // right child

    seg_tree(int n) { vec.assign(4 * n,
NEUTRAL_ELEMENT); }
    inline int combine(int a, int b) {
return min(a, b); }
    // call build(1, 1, n)
    void build(int at, int start, int
end) {
        if (start == end) {
            vec[at] = numbers[start];
            return;
        }
        int mid = (start + end) >> 1;
        build(lc(at), start, mid);
        build(rc(at), mid + 1, end);
        vec[at] = combine(vec[lc(at)],
vec[rc(at)]);
        return;
    }
    // for updating index i, call
update(1, 1, n, i)
    void update(int at, int start, int
end, int update_index) {
        if (start == end && start ==
update_index) {
            vec[at] =
numbers[update_index];
            return;
        }
        int mid = (start + end) >> 1;
        if (update_index <= mid) {
            update(lc(at), start, mid,
update_index);
        } else
            update(rc(at), mid + 1,
end, update_index);
        vec[at] = combine(vec[lc(at)],
vec[rc(at)]);
        return;
    }
    // for query [l, r] call query(1,
1, n, l, r)
    int query(int at, int start, int
end, int q_left, int q_right) {
        if (start > q_right || end <
q_left) return NEUTRAL_ELEMENT;
        if (start >= q_left && end <=
q_right) return vec[at];
        int mid = (start + end) >> 1;
        int l = query(lc(at), start,
mid, q_left, q_right);
        int r = query(rc(at), mid + 1,
end, q_left, q_right);
        return combine(l, r);
    }
};
```

Sum Segment Tree Lazy Propagation:

```
const int MX = 2e5 + 5;
vector<int> numbers(MX);

typedef long long ll;

struct seg_tree {
    vector<ll> vec, lazy;
    const ll NEUTRAL_ELEMENT = 0LL;
    inline int lc(int x) { return (x <<
1); }        // left child
    inline int rc(int x) { return ((x
<< 1) | 1); }  // right child
    seg_tree(int n) {
        vec.assign(4 * n,
NEUTRAL_ELEMENT);
        lazy.assign(4 * n,
NEUTRAL_ELEMENT);
    }
    inline ll combine(ll a, ll b) {
return a + b; }
    inline void push(int at, int start,
int end) {
        if (lazy[at] ==
NEUTRAL_ELEMENT) return;
        vec[at] = combine(vec[at],
lazy[at] * (end - start + 1));
        if (start != end) {
            lazy[lc(at)] =
combine(lazy[lc(at)], lazy[at]);
```

```
            lazy[rc(at)] =
combine(lazy[rc(at)], lazy[at]);
        }
        lazy[at] = NEUTRAL_ELEMENT;
    }
    inline void pull(int at) { vec[at]
= combine(vec[lc(at)], vec[rc(at)]); }
    // call build(1, 1, n)
    void build(int at, int start, int
end) {
        lazy[at] = NEUTRAL_ELEMENT;
        if (start == end) {
            vec[at] = numbers[start];
            return;
        }
        int mid = (start + end) >> 1;
        build(lc(at), start, mid);
        build(rc(at), mid + 1, end);
        pull(at);
        return;
    }
    // for incrementing index [l, r] by
val, call update(1, 1, n, l, r, val)
    void update(int at, int start, int
end, int q_left, int q_right, ll val) {
        push(at, start, end);
        if (start > q_right || end <
q_left) return;
        if (start >= q_left && end <=
q_right) {
            lazy[at] = val;
            push(at, start, end);
            return;
        }
        int mid = (start + end) >> 1;
        update(lc(at), start, mid,
q_left, q_right, val);
        update(rc(at), mid + 1, end,
q_left, q_right, val);
        pull(at);
    }
    // for query [l, r] call query(1,
1, n, l, r)
    ll query(int at, int start, int
end, int q_left, int q_right) {
        push(at, start, end);
        if (start > q_right || end <
q_left) return NEUTRAL_ELEMENT;
        if (start >= q_left && end <=
q_right) return vec[at];
        int mid = (start + end) >> 1;
        return combine(query(lc(at),
start, mid, q_left, q_right),
                       query(rc(at),
mid + 1, end, q_left, q_right));
    }
};
```

DSU:

```
const int MX = 1e6 + 5;
vector<int> size_v(MX);
vector<int> parent(MX);

int find_set(int v) {  // O(1)
    if (v == parent[v])
        return v;
    else
        return parent[v] =
find_set(parent[v]);
}
void make_set(int v) {
    parent[v] = v;
    size_v[v] = 1;
}
bool union_sets(int a, int b) {  //
O(1)
    // returns false if same set
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (size_v[a] < size_v[b])
swap(a, b);
        parent[b] = a;
        size_v[a] += size_v[b];
        return true;
    } else
        return false;
}
```

Ordered Set:

```
#include <bits/stdc++.h>
#include
<ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace std;
using namespace __gnu_pbds;
template <typename T>
using ordered_set =
    tree<T, null_type, less<T>,
rb_tree_tag,
tree_order_statistics_node_update>;

#define ll long long
int main() {
    int n, k;
    cin >> n >> k;
    ordered_set<int> children;
    for (int i = 1; i <= n; i++)
children.insert(i);
    int startPosition = 0;
    while (children.size() > 0) {
        startPosition %=
(children.size());
        int posToRemove =
(startPosition + k) % children.size();
        startPosition = posToRemove;
        auto t =
children.find_by_order(posToRemove);
        cout << *t << " ";
        children.erase(t);
    }
    return 0;
}
```

Square Root Decomposition:

```
#include <bits/stdc++.h>
using namespace std;
int n, q;
vector<int> a(n);
int main() {
    // input data
    //  preprocessing
    const int len =
        (int)sqrt(n + .0) + 1;  // size
of the block and the number of blocks
    vector<int> b(len);
```

```
    for (int i = 0; i < n; ++i) b[i /
len] += a[i];

    // answering the queries
    while (q--) {
        int l, r;
        // read input data for the next
query
        cin >> l >> r;
        int sum = 0;
        int c_l = l / len, c_r = r /
len;
        if (c_l == c_r)
            for (int i = l; i <= r;
++i) sum += a[i];
        else {
            for (int i = l, end = (c_l
+ 1) * len - 1; i <= end; ++i)
                sum += a[i];
            for (int i = c_l + 1; i <=
c_r - 1; ++i) sum += b[i];
            for (int i = c_r * len; i
<= r; ++i) sum += a[i];
        }
    }
}
```

Mo's Algorithm for Highest Frequency:

```
#include <bits/stdc++.h>
using namespace std;
#define endl "\n"
typedef long long ll;
struct Query {
    int l, r, index, ans;
};
const int BLOCK_SIZE = 350;
const int MX = 1e5 + 10;
int n, q, max_freq = 1;
int current_left = 0, current_right = -
1;
vector<int> numbers, frequency(MX),
freq_frequency;
vector<Query> queries;
void add(int index) {
    if (index < 0 || index >=
numbers.size()) return;
    int element = numbers[index];
    int previous_frequency =
frequency[element];
    frequency[element]++;
    int current_frequency =
frequency[element];
    if (previous_frequency > 0)
freq_frequency[previous_frequency]--;

freq_frequency[current_frequency]++;
    if
(freq_frequency[current_frequency] == 1
&& current_frequency > max_freq)
        max_freq = current_frequency;
}
void remove(int index) {
    if (index < 0 || index >=
numbers.size()) return;
    int element = numbers[index];
    int previous_frequency =
frequency[element];
    frequency[element]--;
    int current_frequency =
frequency[element];
    if (current_frequency > 0)
freq_frequency[current_frequency]++;
    freq_frequency[previous_frequency]-
-;
    if (max_freq == previous_frequency
&&

freq_frequency[previous_frequency] ==
0)
        max_freq = current_frequency;
}
void processQueries(Query& q) {
    while (current_left > q.l) {
        current_left--;
        add(current_left);
    }
    while (current_right < q.r) {
        current_right++;
        add(current_right);
    }
    while (current_left < q.l) {
        remove(current_left);
        current_left++;
    }
    while (current_right > q.r) {
        remove(current_right);
        current_right--;
    }
    q.ans = max_freq;
}
void solve() {
    cin >> n >> q;
    numbers.resize(n);
    freq_frequency.assign(n + 2, 0);
    for (int i = 0; i < n; i++) cin >>
numbers[i];
    queries.resize(q);
    for (int i = 0; i < q; i++) {
        cin >> queries[i].l >>
queries[i].r;
        queries[i].index = i;
    }
    sort(queries.begin(),
queries.end(), [&](const Query& p,
const Query& q) {
        if (p.l / BLOCK_SIZE != q.l /
BLOCK_SIZE)
            return make_pair(p.l, p.r)
< make_pair(q.l, q.r);
        return (p.l / BLOCK_SIZE & 1) ?
(p.r < q.r) : (p.r > q.r);
    });
    for (int i = 0; i < q; i++)
processQueries(queries[i]);
    sort(queries.begin(),
queries.end(),
         [&](const Query& p, const
Query& q) { return p.index < q.index;
});
    for (int i = 0; i < q; i++) {
        cout << queries[i].ans << endl;
    }
}
```

Lowest Common Ancestor (LCA):

```
const int MX = 10000;
const int LOG = __lg(MX - 1) + 1;
int n;
vector<vector<int>> adj(MX);
```

```
vector<vector<int>> up(
    MX, vector<int>(LOG));  // up[i][j]
is the 2^j th ancestor of i
vector<int> depth(MX);

void dfs(int v, int p) {
    for (int other : adj[v]) {
        if (other == p) continue;
        depth[other] = depth[v] + 1;
        up[other][0] = v;
        for (int j = 1; j < LOG; j++) {
            up[other][j] =
up[up[other][j - 1]][j - 1];
        }
        dfs(other, v);
    }
}
int getKthAncestor(int u, int k) {
    // make sure dfs(root, -1) has been
called
    if (depth[u] < k) return -1;
    for (int j = LOG - 1; j >= 0; j--)
{
        if (k & (1 << j)) u = up[u][j];
    }
    return u;
}
int getDistanceBetweenNodes(int a, int
b) {
    // make sure dfs(root, -1) has been
called
    if (depth[a] < depth[b]) swap(a,
b);
    int k = depth[a] - depth[b];
    int distance = k;
    a = getKthAncestor(a, k);
    if (a == b) return distance;
    for (int j = LOG - 1; j >= 0; j--)
{
        if (up[a][j] != up[b][j]) {
            a = up[a][j];
            b = up[b][j];
            distance += 2 * (1 << j);
        }
    }
    return distance + 2;
}
int get_LCA(int a, int b) {
    // make sure dfs(root, -1) has been
called
    if (depth[a] < depth[b]) swap(a,
b);
    int k = depth[a] - depth[b];
    a = getKthAncestor(a, k);
    if (a == b) return a;
    for (int j = LOG - 1; j >= 0; j--)
{
        if (up[a][j] != up[b][j]) {
            a = up[a][j];
            b = up[b][j];
        }
    }
    return up[a][0];
}
```

Dijkstra's Algorithm (SSSP):

```
// fails if there are negative edges
typedef long long ll; // O(V+ElogV)
struct Edge {
    int to;
    ll weight;
    bool operator>(const Edge& other)
const {
        if (weight != other.weight)
return weight > other.weight;
        return to > other.to;
    }
};

const ll INF = 2e15;
vector<vector<Edge>> adj;

void dijkstra(int source, vector<ll>&
dist, vector<int>& parent) {
    const int n = adj.size();
    dist.assign(n, INF);
    parent.assign(n, -1);
    vector<bool> visited(n + 2, false);

    dist[source] = 0;
    priority_queue<Edge, vector<Edge>,
greater<Edge>> pq;
    pq.push({source, 0});
    while (!pq.empty()) {
        int from = pq.top().to;
        pq.pop();
        if (visited[from]) continue;
        visited[from] = true;
        for (auto& edge : adj[from]) {
            if (dist[edge.to] >
dist[from] + edge.weight) {
                dist[edge.to] =
dist[from] + edge.weight;
                parent[edge.to] = from;
                pq.push({edge.to,
dist[edge.to]});
            }
        }
    }
}
```

Bellman Ford (SSSP):

```
// works with negative edge, detects
negative cycle, O(VE)
bool bellman_ford(int source,
vector<ll>& dist, vector<int>& parent)
{
    const int n = adj.size();
    dist.assign(n, INF);
    parent.assign(n, -1);

    for (int i = 1; i <= n; i++) {
        bool changed = false;
        for (int from = 0; from < n;
from++) {
            for (auto& edge :
adj[from]) {
                int to = edge.to;
                ll weight =
edge.weight;
                if (dist[from] < INF) {
                    if (dist[to] >
dist[from] + weight) {
                        if (i == n)
return false; // negative cycle present
                        dist[to] =
dist[from] + weight;
```

```
                            parent[to] = from;
                            changed = true;
                        }
                    }
                }
            }
            if (!changed) break;
        }
        return true;
}
```

Floyd Warshall (APSP):

```
void floyd_warshall() { // O(V3)
    distfw = adj_mat;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) distfw[i][i] = 0;
    }

    for (int j = 1; j <= n; j++) {
        for (int i = 1; i <= n; i++) {
            for (int k = 1; k <= n; k++) {
                if (distfw[i][k] > distfw[i][j] + distfw[j][k]) {
                    distfw[i][k] = distfw[i][j] + distfw[j][k];
                    if (i != k) parent[i][k] = parent[j][k];
                }
            }
        }
    }
}
```

If there exists a negative cycle between i and j, distfw[i][j] will be negative after the end of the iterations.

Matrix Multiplication (APSP):

```
// can find minimal path with specific number of edges, FW can't
vector<vector<int>> extend_shortest_path(vector<vector<int>>& d,
 vector<vector<int>>& w) {
    vector<vector<int>> new_d(n + 2, vector<int>(n + 2));

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            new_d[i][j] = INF;
            for (int k = 1; k <= n; k++) {
                new_d[i][j] = min(new_d[i][j], d[i][k] + w[k][j]);
            }
        }
    }

    return new_d;
}
void mat_mul_shortest_path_faster() {
    // O(V3logV)
    distmm = adj_mat;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) distmm[i][i] = 0;
    }

    int b = n - 1;
    int m = 1;
    while (m < b) {
        distmm = extend_shortest_path(distmm, distmm);
        m *= 2;
    }
}
```

Max Flow (Edmond Karp):

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

int n;
const int INF = 2e8;
vector<vector<int>> capacity, adj;

int bfs(int s, int t, vector<int>& parent) {
    fill(parent.begin(), parent.end(), -1);
    parent[s] = -2;
    queue<pair<int, int>> q;
    q.push({s, INF});

    while (!q.empty()) {
        int cur = q.front().first;
        int flow = q.front().second;
        q.pop();

        for (int next : adj[cur]) {
            if (parent[next] == -1 && capacity[cur][next]) {
                parent[next] = cur;
                int new_flow = min(flow, capacity[cur][next]);
                if (next == t) return new_flow;
                q.push({next, new_flow});
            }
        }
    }

    return 0;
}

int maxflow(int s, int t) {
    int flow = 0;
    vector<int> parent(n);
    int new_flow;

    while (new_flow = bfs(s, t, parent)) {
        flow += new_flow;
        int cur = t;
        while (cur != s) {
            int prev = parent[cur];
            capacity[prev][cur] -= new_flow;
            capacity[cur][prev] += new_flow;
            cur = prev;
        }
    }

    return flow;
```

```
}
```

MST (Kruskal):

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

struct Edge {
    int from, to;
    ll weight;
    bool operator<(const Edge& other)
const {
        if (weight != other.weight)
return weight < other.weight;
        return make_pair(from, to) <
make_pair(other.from, other.to);
    }
};

int n;
vector<int> parent, size_v;
vector<pair<int, int>> result;  //
resulting MST

int find_set(int v) {  // O(1)
    if (v == parent[v])
        return v;
    else
        return parent[v] =
find_set(parent[v]);
}
bool union_sets(int a, int b) {  //
O(1)
    // returns false if same set
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (size_v[a] < size_v[b])
swap(a, b);
        parent[b] = a;
        size_v[a] += size_v[b];
        return true;
    } else
        return false;
}
ll kruskal(vector<Edge>& edges) {
    parent.resize(n + 2);
    size_v.assign(n + 2, 1);
    for (int i = 0; i <= n; i++)
parent[i] = i;
    sort(edges.begin(), edges.end());
    ll total_weight = 0;
    for (int i = 0; i < edges.size();
i++) {
        int u = edges[i].from;
        int v = edges[i].to;
        ll w = edges[i].weight;
        if (find_set(u) != find_set(v))
{
            total_weight += w;
        result.push_back(make_pair(u,
v));
            union_sets(u, v);
        }
    }
    return total_weight;
}
```

Topological Sorting:

```
int n, d;
vector<vector<int>> adj;
vector<char> color;
stack<int> st;

bool dfs(int node) {
    color[node] = 'G';
    bool flag = true;
    for (auto& they : adj[node]) {
        if (color[they] == 'G') return
false;
        if (color[they] == 'W') flag &=
dfs(they);
    }
    color[node] = 'B';
    st.push(node);
    return flag;
}

bool topSort(vector<int>& ans) {
    color.assign(n + 1, 'W');
    // if s1 is dependent on s2,
adj[s2].push_back(s1)
    vector<int> ret;
    bool possible = true;
    for (int i = 1; i <= n && possible;
i++)
        if (color[i] == 'W') possible
&= dfs(i);
    if (!possible) {
        return false;  // not possible
    } else {
        while (!st.empty()) {
            ans.push_back(st.top());
            st.pop();
        }
        return true;
    }
}
```

SCC (Kosaraju's Algorithm):

```
vector<vector<int>> adj, adj_rev;
vector<bool> visited;
stack<int> st;

void dfs1(int node) {
    visited[node] = true;
    for (auto& other : adj[node]) {
        if (!visited[other])
dfs1(other);
    }
    st.push(node);
}

void dfs2(int node) {
    visited[node] = true;
    cout << node << " ";
    for (auto& other : adj_rev[node]) {
        if (!visited[other])
dfs2(other);
    }
}

int main() {
    int n, s;
    cin >> n >> s;
    visited.assign(n + 1, false);
    adj.resize(n + 1);
    adj_rev.resize(n + 1);
```

```
    for (int i = 0; i < s; i++) {
        int a, b;
        cin >> a >> b;
        adj[a].push_back(b);
        adj_rev[b].push_back(a);
    }
    for (int i = 1; i <= n; i++)
        if (!visited[i]) dfs1(i);
    visited.assign(n + 1, false);
    while (!st.empty()) {
        int now = st.top();
        st.pop();
        if (visited[now]) continue;
        dfs2(now);
        cout << endl;
    }
}
```

Acyclicity Checking and Cycle Finding:

Finds cycle in O(E), the following is for directed graph. For undirected graphs, gray color is not even needed.

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

int n;
vector<vector<int>> adj;
vector<char> color;
vector<int> parent;
int cycle_start, cycle_end;

bool dfs(int v) {
    color[v] = 1;
    for (int u : adj[v]) {
        if (color[u] == 0) {
            parent[u] = v;
            if (dfs(u)) return true;
        } else if (color[u] == 1) {
            cycle_end = v;
            cycle_start = u;
            return true;
        }
    }
    color[v] = 2;
    return false;
}

void find_cycle() {
    color.assign(n, 0);
    parent.assign(n, -1);
    cycle_start = -1;

    for (int v = 0; v < n; v++) {
        if (color[v] == 0 && dfs(v))
break;
    }

    if (cycle_start == -1) {
        cout << "Acyclic" << endl;
    } else {
        vector<int> cycle;
        cycle.push_back(cycle_start);
        for (int v = cycle_end; v !=
cycle_start; v = parent[v])
            cycle.push_back(v);
        cycle.push_back(cycle_start);
        reverse(cycle.begin(),
cycle.end());

        cout << "Cycle found: ";
        for (int v : cycle) cout << v
<< " ";
        cout << endl;
    }
}
```

DAG Reachability:

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

const int N = 3e5 + 9;
/*add_edge(s, t): insert edge (s,t) to
the network if it does not make a cycle
is_reachable(s, t): return true iff
there is a path s --> t
Complexity: amortized O(n) per update*/
// 0-indexed

struct Italiano {
    int n;
    vector<vector<int>> par;
    vector<vector<vector<int>>> child;
    Italiano(int n)
        : n(n), par(n, vector<int>(n, -
1)), child(n, vector<vector<int>>(n))
{}
    bool is_reachable(int s, int t) {
return s == t || par[s][t] >= 0; }
    bool add_edge(int s, int t) {
        if (is_reachable(t, s)) return
false;  // break DAG condition
        if (is_reachable(s, t)) return
true;   // no-modification performed
        for (int p = 0; p < n; ++p) {
            if (is_reachable(p, s) &&
!is_reachable(p, t)) meld(p, t, s, t);
        }
        return 1;
    }
    void meld(int root, int sub, int u,
int v) {
        par[root][v] = u;
        child[root][u].push_back(v);
        for (int c : child[sub][v]) {
            if (!is_reachable(root, c))
meld(root, sub, v, c);
        }
    }
};
// add edges one by one. if it breaks
DAG law then print it
int main() {
    int n, m;
    cin >> n >> m;
    Italiano t(n);
    while (m--) {
        int u, v;
        cin >> u >> v;
        --u;
        --v;
        if (t.is_reachable(v, u))
            cout << u + 1 << ' ' << v +
1 << '\n';
        else
            t.add_edge(u, v);
    }
    cout << 0 << ' ' << 0 << '\n';
    return 0;
```

```
}
```

String Hashing:

```
typedef long long ll;
const int MX = 1e6 + 5;
const int p1 = 137, p2 = 281;
const int MOD1 = 1e9 + 7;
const int MOD2 = 1e9 + 9;
// Some back-up primes: 1072857881,
1066517951, 1040160883
// use double hashing for caution
// if both hash values match, then same
string
// else different
vector<int> POWER1(MX), POWER2(MX);
vector<int> INV1(MX), INV2(MX);
// use gcd, modInverse codes from page
1
void precompute(const int p, const int
MOD, vector<int>& POWER,
                vector<int>& INV) {
    POWER[0] = 1;
    INV[0] = 1;
    int inv_of_p = modInverse(p, MOD);
    for (int i = 1; i < MX; i++) {
        POWER[i] = (POWER[i - 1] * 1LL
* p) % MOD;
        INV[i] = (INV[i - 1] * 1LL *
inv_of_p) % MOD;
    }
}
int compute_hash_for_a_string(const
string& s, const int p, const int MOD,
                              const
vector<int>& POWER) {
    // call precompute() first
    int hash_value = 0;
    const int n = s.size();
    for (int i = 0; i < n; i++) {
        int val = ((s[i] - 'a' + 1) *
1LL * POWER[i]) % MOD;
        ll now = hash_value + val;
        while (now >= MOD) now -= MOD;
        hash_value = now;
    }
    return hash_value;
}
vector<int> compute_prefix_hash(const
string& s, const int p, const int MOD)
{
    const int n = s.size();
    int p_power = 1;
    vector<int> hash_values(n, 0);
    for (int i = 0; i < n; i++) {
        hash_values[i] = (p_power * 1LL
* (s[i] - 'a' + 1)) % MOD;
        if (i > 0) {
            ll now = hash_values[i] +
hash_values[i - 1];
            while (now >= MOD) now -=
MOD;
            hash_values[i] = now;
        }
        p_power = (p_power * 1LL * p) %
MOD;
    }
    return hash_values;
}
int prefix_hash(vector<int>& pref,
vector<int>& INV, int l, int r, const
int p,
                const int MOD) {
    // call precompute() first
    assert(l <= r);
    if (l == 0) return pref[r];
    int inv = INV[l];
    ll now = pref[r] - pref[l - 1];
    while (now < 0) now += MOD;
    while (now >= MOD) now -= MOD;
    return (inv * 1LL * now) % MOD;
}
pair<int, int>
getBothPrefixHashes(vector<int>& pref1,
vector<int>& pref2,
                                   int
l, int r) {
    pair<int, int> ans;
    ans.first = prefix_hash(pref1,
INV1, l, r, p1, MOD1);
    ans.second = prefix_hash(pref2,
INV2, l, r, p2, MOD2);
    return ans;
}
vector<int> compute_suffix_hash(const
string& s, const int p, const int MOD)
{
    const int n = s.size();
    int p_power = 1;
    vector<int> hash_values(n, 0);
    for (int i = n - 1; i >= 0; i--) {
        hash_values[i] = (p_power * 1LL
* (s[i] - 'a' + 1)) % MOD;
        if (i < n - 1) {
            ll now = hash_values[i] +
hash_values[i + 1];
            while (now >= MOD) now -=
MOD;
            hash_values[i] = now;
        }
        p_power = (p_power * 1LL * p) %
MOD;
    }
    return hash_values;
}
int suffix_hash(vector<int>& suf,
vector<int>& INV, int l, int r, const
int p,
                const int MOD) {
    // call precompute() first
    const int n = suf.size();
    assert(l <= r);
    if (r == n - 1) return suf[l];
    int inv = INV[n - 1 - r];
    ll now = suf[l] - suf[r + 1];
    while (now < 0) now += MOD;
    while (now >= MOD) now -= MOD;
    return (inv * 1LL * now) % MOD;
}
pair<int, int>
getBothSuffixHashes(vector<int>& suf1,
vector<int>& suf2, int l,
                                   int
r) {
    pair<int, int> ans;
    ans.first = prefix_hash(suf1, INV1,
l, r, p1, MOD1);
    ans.second = prefix_hash(suf2,
INV2, l, r, p2, MOD2);
```

```
    return ans;
}
```

Trie (Prefix Tree):

Sometimes also helpful for maximum xor problems.

```
//Space Complexity : O(Sum of |S_i|)
const int LOG = 20;
struct Trie{
    struct Node{
        Node* Child[2];
        Node()
{Child[0]=Child[1]=NULL;}
    };
    Node* root;

    void Init() {root = new Node();}
    bool Check(int x,int b)
    {return (x>>b)&1;}

    void Insert(int n){
        Node *Cur = root;
        for(int i = LOG-1; i>=0; i--){
            bool ID = Check(n,i);
            if(!Cur->Child[ID]) Cur-
>Child[ID] = new Node();
            Cur = Cur->Child[ID];
        }
    }

    int MinQuery(int XOR){
        int Ans = 0;
        Node *Cur = root;
        for(int i = LOG-1; i>=0; i--){
            bool ID = Check(XOR,i);
            if(Cur->Child[ID]) Cur =
Cur->Child[ID];
            else Cur = Cur->Child[!ID],
Ans += (1<<i);
        }
        return Ans;
    }
};
```

Aho-Korasick Algorithm:

```
// Aho-Corasick
// Complexity : |Text| + Sum of all
|Pattern| + O(number of Occurrences)
// if occurrence positions needed,
Worst Case Complexity : (SumLen) Root
(SumLen)
#include <bits/stdc++.h>
using namespace std;
const int MAXT = 1000005;  // Length of
Text
const int MAXP = 1000005;  // Sun of
all |Pattern|
const int MAXQ = 1000005;  // Number of
Patterns

int n;
map<char,int> Next[MAXP];
int Root;                  // AC
automaton Root
int Nnode;                 // Total node
count
int Link[MAXP];            // failure
links
int Len[MAXP];             // Len[i] =
length of i-th pattern
vector<int> End[MAXP];     // End[i] =
indices of patterns those end in node i
// vector<int> Occ[MAX];  // Occ[i] =
occurrences of i-th pattern
vector<int> edgeLink[MAXP];
vector<int> perNodeText[MAXP];

int in[MAXQ], out[MAXQ];
int euler[MAXT];
int Time;

void Clear(int node){
    Next[node].clear();
    End[node].clear();
    edgeLink[node].clear();
    perNodeText[node].clear();
}

void init(){
    Time = 0;
    Root = Nnode = 0;
    Clear(Root);
}

void insertword(string p,int ind){
    int len = p.size();
    int now = Root;
    for(int i=0; i<len; i++){
        if(!Next[now][p[i]]){
            Next[now][p[i]] = ++Nnode;
            Clear(Nnode);
        }
        now = Next[now][p[i]];
    }
    End[now].push_back(ind);
}

void push_links(){
    queue<int> q;
    Link[0] = -1;
    q.push(0);
    while(!q.empty()){
        int u = q.front();
        q.pop();
        for(auto edge : Next[u]){
            char ch = edge.first;
            int v = edge.second;
            int j = Link[u];

            while(j != -1 &&
!Next[j][ch]) j = Link[j];
            if(j != -1) Link[v] =
Next[j][ch];
            else Link[v] = 0;

            q.push(v);

edgeLink[Link[v]].push_back(v);
            // for(int x :
End[Link[v]]) End[v].push_back(x);
        }
    }
}

void traverse(string s){
    int len = s.size();
    int now = Root;
    for(int i = 0; i < len; i++)    {
```

```
        while(now != -1 &&
!Next[now][s[i]]) now = Link[now];
        if(now!=-1) now =
Next[now][s[i]];
        else now = 0;

perNodeText[now].push_back(i+1);  //
using 1 based indexing for text indices
        // for(int
x=0;x<End[now].size();x++)
Occ[End[now][x]].push_back(i);
    }
}

// After dfs, the occurence of ith
query string will be the count of
// all the occurrence of the subtree
under the endNode of ith string
void dfs(int pos){
    for(int q : End[pos]) in[q] = Time
+ 1;
    for(int val : perNodeText[pos])
euler[++Time] = val;
    for(int to : edgeLink[pos])
dfs(to);
    for(int q : End[pos]) out[q] =
Time;
}

int main(){
    // init();
    // push_links();
    // traverse(s);
    // dfs(Root);
}
```

Manachar's Algorithm:

Works with palindromic substrings.

```
#include <bits/stdc++.h>
using namespace std;
#define endl "\n"
int main() {
    string s;
    cin >> s;
    const int n = s.size();
    vector<int> d1(n, 0);
    // radius of maximum odd length
palindrome centered at i
    // here d1[i] = the palindrome has
d1[i] - 1 right characters from i
    // e.g. for aba, d1[1] = 2;
    for (int i = 0, l = 0, r = -1; i <
n; i++) {
        int k = (i > r) ? 1 : min(d1[l
+ r - i], r - i);
        while (0 <= i - k && i + k < n
&& s[i - k] == s[i + k]) k++;
        d1[i] = k--;
        if (i + k > r) {
            l = i - k;
            r = i + k;
        }
    }
    vector<int> d2(n, 0);
    // radius of maximum even length
palindrome centered at i
    // here d2[i] = the palindrome has
d2[i] - 1 right characters from i
    // e.g. for abba, d2[2] = 2;
    for (int i = 0, l = 0, r = -1; i <
n; i++) {
        int k = (i > r) ? 0 : min(d2[l
+ r - i + 1], r - i + 1);
        while (0 <= i - k - 1 && i + k
< n && s[i - k - 1] == s[i + k]) k++;
        d2[i] = k--;
        if (i + k > r) {
            l = i - k - 1;
            r = i + k;
        }
    }
    for (int i = 0; i < n; i++) cout <<
d1[i] << ' ';
    cout << endl;
    for (int i = 0; i < n; i++) cout <<
d2[i] << ' ';
    cout << endl;
    // number of palindromes
    long long ans = 0;
    for (int i = 0; i < n; i++) ans +=
d1[i] + d2[i];
    cout << ans << endl;
    return 0;
}
```

KMP Fail + Z-Algorithm:

```
//0-indexed string s[0...n-1]
//fail[i] = maximum x such that
s[0....x-1] matches with s[i-x+1...i]
int fail[MAX];
void build_failure(string s){
    int n=s.size();
    for(int i=1;i<n;i++){
        int j=fail[i-1];
        while(j>0 && s[i]!=s[j])
j=fail[j-1];
        if(s[i]==s[j]) j++;
        fail[i]=j;
    }
}

int KMPSearch(string txt, string pat){
    int Count = 0;
    int i = 0;
    int j = 0;
    while(i < txt.size()){
        if(txt[i] == pat[j]) i++, j++;
        if(j == pat.size()) Count++, j
= fail[j - 1];
        else if(i < txt.size() &&
txt[i] != pat[j]){
            if(j != 0) j = fail[j - 1];
            else i++;
        }
    }
    return Count;
}

//0-indexed string s[0...n-1]
//z[i] = maximum x such that s[0....x-
1] matches with s[i..i+x-1]
int z[MAX];
void zAlgo(string s){
    int L=0,R=0;
    int n=s.size();
    for(int i=1;i<n;i++){
        if(i>R) {L=R=i; while(R<n &&
s[R-L]==s[R]) R++; z[i]=R-L; R--;}
```

```
        else{
            int k=i-L;
            if(z[k]<R-i+1) z[i]=z[k];
            else {L=i; while(R<n &&
s[R-L]==s[R]) R++; z[i]=R-L; R--;}
        }
    }
}
```

Suffix Automaton:

```
// Tested : HDU 4270 - Dynamic Lover
// Tested : SPOJ SUBLEX, SPOJ LCS, CF
128B

#include<bits/stdc++.h>
using namespace std;
#define ll long long int
const int ALPHA = 28;
const int MAXLEN = 300005;
const int LOG = 20;
int TotalLen, Size;
int Root, Last;

// Dstnct_substr can be calculated
online as St[pos].Dstnct_substr =
St[pos].Len - St[St[pos].link].Len
// Occurrence can be calculated online,
// First, Every node pos except a clone
node, St[pos].Occurrence = 1
// then, St[pos].Occurence = sum of
St[i].Occurrence where St[i].Link =
pos|
// Must make isvlid array 0 for more
than one test case

struct Node{
    int Link, Len;
    ll Occurrence;    // How many times
each state (endpos) occurs
    ll Word;          // How many
substrings can be reached from this
node
    ll Dstnct_substr; // How many
distinct substrings can be reached from
this node
    int FirstPos, version, baseID;
    int Next[ALPHA];
    void Clear(){
        Len = Occurrence = Word =
Dstnct_substr = 0;
        Link = baseID = FirstPos =
version = -1;
        memset(Next, 0, sizeof(Next));
    }
};

Node St[MAXLEN*2];
bool isValid[MAXLEN*2];
vector<int>CurrList;
vector<int>LastList;
vector < int > linkTree[2 * MAXLEN];
int Par[2 * MAXLEN], P[2 *
MAXLEN][LOG];
int perPrefNode[MAXLEN]; ///[i] = node
created after inserting ith position in
SAM

inline void CreateNode(int dep){
    St[Size].Clear();
    St[Size].Len = dep;
    St[Size].FirstPos = dep;
    St[Size].baseID = Size;
    isValid[Size] = true;
}

inline void init(){
    Size = 0;
    Root = Last = TotalLen = 0;
    St[Root].Clear();
    CurrList.clear();
    LastList.clear();
    for(int i = 0; i < 2 * MAXLEN; i++)
linkTree[i].clear();
}

inline bool has(int u, int ch){
    int x = St[u].Next[ch];
    return isValid[St[x].baseID];
}

inline void SAM(int pos, int ch){
    TotalLen++;
    int Curr = ++Size;
    CreateNode(St[Last].Len + 1);

    int p = Last;
    while(p !=-1 && !has(p,ch)){
        St[p].Next[ch] = Curr;
        p = St[p].Link;
    }

    if(p == -1) {
        St[Curr].Link = Root;
    }
    else{
        int q = St[p].Next[ch];
        if(St[q].Len == St[p].Len + 1)
St[Curr].Link = q;
        else{
            int Clone = ++Size;
            CreateNode(St[p].Len + 1);

            St[Clone].FirstPos =
St[q].FirstPos;

memcpy(St[Clone].Next,St[q].Next,sizeof
(St[q].Next));
            St[Clone].Link =
St[q].Link;
            St[Clone].baseID =
St[q].baseID;
            St[q].Link = St[Curr].Link
= Clone;

            while(p != -1 &&
St[p].Next[ch] == q) St[p].Next[ch] =
Clone, p = St[p].Link;

linkTree[St[Clone].Link].push_back(Clon
e);
            Par[Clone] =
St[Clone].Link;
        }
    }
    perPrefNode[pos] = Curr; //pass pos
as parameter
```

```
linkTree[St[Curr].Link].push_back(Curr)
;
    Par[Curr] = St[Curr].Link;
    CurrList.push_back(Curr);
    LastList.push_back(Last);
    Last = Curr;
}

inline void Del(int len){
    if(!len) return;

    for(int i = 0; i < len; i++){

isValid[St[CurrList.back()].baseID] =
false;
        CurrList.pop_back();
        Last = LastList.back();
        LastList.pop_back();
        TotalLen--;
    }
}

inline void MarkTerminal(int u, int v =
1){
    while(u != -1) St[u].version = v,
u = St[u].Link, St[u].Occurrence = 1;
}

// Returns Smallest Substring with
length len
// If i + len > s.size(), consider
substring(i, s.size())
int LexSmallestSubStr(int len, int
idx){
    MarkTerminal(Last,idx);

    int cur = Root;
    for(int i= 0; i< len; i++){
        if(cur > Root &&
St[cur].version == idx) return TotalLen
- i + 1;
        for(int ch = 0; ch < ALPHA;
ch++){
            if(!has(cur, ch ))
continue;
                cur = St[cur].Next[ch];
                break;
            }
        }
    return St[cur].FirstPos - len + 1;
}

// Returns longest common substring of
2 strings
int LCS(char * s1, char * s2){
    int len1 = strlen(s1), len2 =
strlen(s2);
    init();
    for(int i = 0; i < len1; i++)
SAM(i, s1[i] - 'a');

    int curNode = 0, curLen = 0, ans =
0;
    for(int i = 0; i < len2; i++){
        int ch = s2[i] - 'a';
        while(curNode > -1 &&
St[curNode].Next[ch] == 0)
            curNode = St[curNode].Link,
curLen = St[curNode].Len;

        if(curNode == -1) curNode = 0;
        if(St[curNode].Next[ch])
curNode = St[curNode].Next[ch],
curLen++;
        ans = max(ans, curLen);
    }
    return ans;
}

// Must call MarkTerminal Before
void dfs_sam(int pos){
    if(St[pos].Dstnct_substr) return;
    int resDstnct = 1;
    for(int i = 0; i < ALPHA; i++)
if(St[pos].Next[i]){
        int to = St[pos].Next[i];
        dfs_sam(to);
        resDstnct +=
St[to].Dstnct_substr;
        St[pos].Occurrence +=
St[to].Occurrence;
        St[pos].Word += St[to].Word;
    }
    St[pos].Dstnct_substr = resDstnct;
    St[pos].Word += St[pos].Occurrence;
}

void PrintKthLexSubstr(int k){
    // Must call dfs_sam before
    int cur = Root;
    while(k > 0){
        int tmp = 0;
        for(int i = 0; i < 26; i++){
            if(St[cur].Next[i] == 0)
continue;
            int to = St[cur].Next[i];
            if(tmp + St[to].Word >= k){
                k -= (tmp +
St[to].Occurrence), cur = to;
                printf("%c", 'a' + i);
                break;
            }
            tmp += St[to].Word;
        }
    }
}

void PrintKthLexDstSubstr(int k){
    // Must call dfs_sam before
    int cur = Root;
    while(k > 0){
        int tmp = 0;
        for(int i = 0; i < 26; i++){
            if(St[cur].Next[i] == 0)
continue;
            int to = St[cur].Next[i];
            if(tmp +
St[to].Dstnct_substr >= k){
                k -= (tmp + 1), cur =
to;
                printf("%c", 'a' + i);
                break;
            }
            tmp +=
St[to].Dstnct_substr;
        }
    }
```

```
}

void BuildSparse(int n){
    // Using 0 for uninitialized parent (Remember and use carefully)
    for(int i=1; i<=n; i++) for(int j=0; j<LOG; j++) P[i][j] = 0;

    for(int i=1; i<=n; i++) P[i][0] = Par[i];
    for(int j=1; j<LOG; j++){
        for(int i=1; i<=n; i++){
            if(P[i][j-1] != 0){
                int x = P[i][j-1];
                P[i][j] = P[x][j-1];
            }
        }
    }
}

int getNodeSubstring(int l, int r){
    int cur = perPrefNode[r], curLen = r - l + 1;
    if(St[St[cur].Link].Len + 1 <= curLen) return cur;

    for(int i = LOG - 1; i >= 0; i--){
        if(P[cur][i] == 0) continue;
        int ncur = P[cur][i];
        if(St[ncur].Len >= curLen) cur = ncur;
    }
}
// must call init() at the start
```

Geometry: (entirely from ACM-Library by BUET-Gifted-Hypocrites)

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

const double pi = 4 * atan(1);
const double eps = 1e-6;

inline int dcmp (double x) { if (fabs(x) < eps) return 0; else return x < 0 ? -1 : 1; }
double fix_acute(double th) {return th<-pi ? (th+2*pi): th>pi ? (th-2*pi) : th;}

inline double getDistance (double x, double y) { return sqrt(x * x + y * y); }
inline double torad(double deg) { return deg / 180 * pi; }

struct Point {
    double x, y;
    Point (double x = 0, double y = 0): x(x), y(y) {}
    void read () { scanf("%lf%lf", &x, &y); }
    void write () { printf("%lf %lf", x, y); }

    bool operator == (const Point& u) const { return dcmp(x - u.x) == 0 && dcmp(y - u.y) == 0; }
    bool operator != (const Point& u) const { return !(*this == u); }
    bool operator < (const Point& u) const { return dcmp(x - u.x) < 0 || (dcmp(x-u.x)==0 && dcmp(y-u.y) < 0); }
    bool operator > (const Point& u) const { return u < *this; }
    bool operator <= (const Point& u) const { return *this < u || *this == u; }
    bool operator >= (const Point& u) const { return *this > u || *this == u; }
    Point operator + (const Point& u) { return Point(x + u.x, y + u.y); }
    Point operator - (const Point& u) { return Point(x - u.x, y - u.y); }
    Point operator * (const double u) { return Point(x * u, y * u); }
    Point operator / (const double u) { return Point(x / u, y / u); }
    double operator * (const Point& u) { return x*u.y - y*u.x; }
};
typedef Point Vector;
typedef vector<Point> Polygon;

struct Line {
    double a, b, c;
    Line (double a = 0, double b = 0, double c = 0): a(a), b(b), c(c) {}
};

struct Segment{
    Point a;
    Point b;
    Segment(){}
    Segment(Point aa,Point bb) {a=aa,b=bb;}
};

struct DirLine {
    Point p;
    Vector v;
    double ang;
    DirLine () {}
    DirLine (Point p, Vector v): p(p), v(v) { ang = atan2(v.y, v.x); }
    bool operator < (const DirLine& u) const { return ang < u.ang; }
};

namespace Punctual {
    double getDistance (Point a, Point b) { double x=a.x-b.x, y=a.y-b.y; return sqrt(x*x + y*y); }
};

namespace Vectorial {
    double getDot (Vector a, Vector b) { return a.x * b.x + a.y * b.y; }
    double getCross (Vector a, Vector b) { return a.x * b.y - a.y * b.x; }
    double getLength (Vector a) { return sqrt(getDot(a, a)); }
    double getPLength (Vector a) { return getDot(a, a); }
    double getAngle (Vector u) { return atan2(u.y, u.x); }
```

```
    double getSignedAngle (Vector a,
Vector b) {return getAngle(b)-
getAngle(a);}
    Vector rotate (Vector a, double
rad) { return Vector(a.x*cos(rad)-
a.y*sin(rad),
a.x*sin(rad)+a.y*cos(rad)); }
    Vector ccw(Vector a, double co,
double si) {return Vector(a.x*co-
a.y*si, a.y*co+a.x*si);}
    Vector cw (Vector a, double co,
double si) {return
Vector(a.x*co+a.y*si, a.y*co-a.x*si);}
    Vector scale(Vector a, double s =
1.0) {return a / getLength(a) * s;}
    Vector getNormal (Vector a) {
double l = getLength(a); return
Vector(-a.y/l, a.x/l); }
};

namespace ComplexVector {
    typedef complex<double> Point;
    typedef Point Vector;

    double getDot(Vector a, Vector b) {
return real(conj(a)*b); }
    double getCross(Vector a, Vector b)
{ return imag(conj(a)*b); }
    Vector rotate(Vector a, double rad)
{ return a*exp(Point(0, rad)); }
};

namespace Linear {
    using namespace Vectorial;

    Line getLine (double x1, double y1,
double x2, double y2) { return Line(y2-
y1, x1-x2, y1*x2-x1*y2); }
    Line getLine (double a, double b,
Point u) { return Line(a, -b, u.y * b -
u.x * a); }

    bool getIntersection (Line p, Line
q, Point& o) {
        if (fabs(p.a * q.b - q.a * p.b)
< eps)
            return false;
        o.x = (q.c * p.b - p.c * q.b) /
(p.a * q.b - q.a * p.b);
        o.y = (q.c * p.a - p.c * q.a) /
(p.b * q.a - q.b * p.a);
        return true;
    }

    bool getIntersection (Point p,
Vector v, Point q, Vector w, Point& o)
{
        if (dcmp(getCross(v, w)) == 0)
return false;
        Vector u = p - q;
        double k = getCross(w, u) /
getCross(v, w);
        o = p + v * k;
        return true;
    }

    double getDistanceToLine (Point p,
Point a, Point b) { return
fabs(getCross(b-a, p-a) / getLength(b-
a)); }
    double getDistanceToSegment (Point
p, Point a, Point b) {
        if (a == b) return getLength(p-
a);
        Vector v1 = b - a, v2 = p - a,
v3 = p - b;
        if (dcmp(getDot(v1, v2)) < 0)
return getLength(v2);
        else if (dcmp(getDot(v1, v3)) >
0) return getLength(v3);
        else return fabs(getCross(v1,
v2) / getLength(v1));
    }

    double getDistanceSegToSeg (Point
a,Point b,Point c,Point d){
        double Ans=INT_MAX;

Ans=min(Ans,getDistanceToSegment(a,c,d)
);

Ans=min(Ans,getDistanceToSegment(b,c,d)
);

Ans=min(Ans,getDistanceToSegment(c,a,b)
);

Ans=min(Ans,getDistanceToSegment(d,a,b)
);
        return Ans;
    }

    Point getPointToLine (Point p,
Point a, Point b) { Vector v = b-a;
return a+v*(getDot(v, p-a) /
getDot(v,v)); }
    bool onSegment (Point p, Point a,
Point b) { return dcmp(getCross(a-p, b-
p)) == 0 && dcmp(getDot(a-p, b-p)) <=
0; }

    bool haveIntersection (Point a1,
Point a2, Point b1, Point b2) {
        if(onSegment(a1,b1,b2)) return
true;
        if(onSegment(a2,b1,b2)) return
true;
        if(onSegment(b1,a1,a2)) return
true;
        if(onSegment(b2,a1,a2)) return
true;  //Case of touch

        double c1=getCross(a2-a1, b1-
a1), c2=getCross(a2-a1, b2-a1),
c3=getCross(b2-b1, a1-b1),
c4=getCross(b2-b1,a2-b1);
        return dcmp(c1)*dcmp(c2) < 0 &&
dcmp(c3)*dcmp(c4) < 0;
    }
    bool onLeft(DirLine l, Point p) {
return dcmp(l.v * (p-l.p)) >= 0; }
}

namespace Triangular {
    using namespace Vectorial;
```

```
    double getAngle (double a, double
b, double c) { return acos((a*a+b*b-
c*c) / (2*a*b)); }
    double getArea (double a, double b,
double c) { double s =(a+b+c)/2; return
sqrt(s*(s-a)*(s-b)*(s-c)); }
    double getArea (double a, double h)
{ return a * h / 2; }
    double getArea (Point a, Point b,
Point c) { return fabs(getCross(b - a,
c - a)) / 2; }
    double getDirArea (Point a, Point
b, Point c) { return getCross(b - a, c
- a) / 2;}

    //ma/mb/mc = length of median from
side a/b/c
    double getArea_(double ma,double
mb,double mc) {double s=(ma+mb+mc)/2;
return 4/3.0 * sqrt(s*(s-ma)*(s-mb)*(s-
mc));}

    //ha/hb/hc = length of
perpendicular from side a/b/c
    double get_Area(double ha,double
hb,double hc){
        double H=(1/ha+1/hb+1/hc)/2;
double _A_ = 4 * sqrt(H * (H-1/ha)*(H-
1/hb)*(H-1/hc)); return 1.0/_A_;
    }

    bool pointInTriangle(Point a, Point
b, Point c, Point p){
        double s1 = getArea(a,b,c);
        double s2 = getArea(p,b,c) +
getArea(p,a,b) + getArea(p,c,a);
        return dcmp(s1 - s2) == 0;
    }
};

namespace Polygonal {
    using namespace Vectorial;
    using namespace Linear;
    using namespace Triangular;
    double getSignedArea (Point* p, int
n) {
        double ret = 0;
        for (int i = 0; i < n-1; i++)
            ret += (p[i]-p[0]) *
(p[i+1]-p[0]);
        return ret/2;
    }

    int getConvexHull (Point* p, int n,
Point* ch) {
        sort(p, p + n);

        // preparing lower hull
        int m = 0;
        for (int i = 0; i < n; i++){
            while (m > 1 &&
dcmp(getCross(ch[m-1]-ch[m-2], p[i]-
ch[m-1])) <= 0) m--;
            ch[m++] = p[i];
        }

        // preparing upper hull
        int k = m;
        for (int i = n-2; i >= 0; i--){
            while (m > k &&
dcmp(getCross(ch[m-1]-ch[m-2], p[i]-
ch[m-2])) <= 0) m--;
            ch[m++] = p[i];
        }
        if (n > 1) m--;
        return m;
    }

    int isPointInPolygon (Point o,
Point* p, int n) {
        int wn = 0;
        for (int i = 0; i < n; i++) {
            int j = (i + 1) % n;
            if (onSegment(o, p[i],
p[j]) || o == p[i]) return 0;
            int k = dcmp(getCross(p[j]
- p[i], o-p[i]));
            int d1 = dcmp(p[i].y -
o.y);
            int d2 = dcmp(p[j].y -
o.y);
            if (k > 0 && d1 <= 0 && d2
> 0) wn++;
            if (k < 0 && d2 <= 0 && d1
> 0) wn--;
        }
        return wn ? -1 : 1;
    }

    void rotatingCalipers(Point *p, int
n, vector&lt;Segment&gt;& sol) {
        sol.clear();
        int j = 1; p[n] = p[0];
        for (int i = 0; i < n; i++) {
            while (getCross(p[j+1]-
p[i+1], p[i]-p[i+1]) > getCross(p[j]-
p[i+1], p[i]-p[i+1]))
                j = (j+1) % n;

sol.push_back(Segment(p[i],p[j]));
            sol.push_back(Segment(p[i +
1],p[j + 1]));
        }
    }

    void rotatingCalipersGetRectangle
(Point *p, int n, double& area, double&
perimeter) {
        p[n] = p[0];
        int l = 1, r = 1, j = 1;
        area = perimeter = 1e20;

        for (int i = 0; i < n; i++) {
            Vector v = (p[i+1]-p[i]) /
getLength(p[i+1]-p[i]);
            while (dcmp(getDot(v,
p[r%n]-p[i]) - getDot(v, p[(r+1)%n]-
p[i])) < 0) r++;
            while (j < r ||
dcmp(getCross(v, p[j%n]-p[i]) -
getCross(v,p[(j+1)%n]-p[i])) < 0) j++;
            while (l < j ||
dcmp(getDot(v, p[l%n]-p[i]) - getDot(v,
p[(l+1)%n]-p[i])) > 0) l++;
```

```
            double w = getDot(v,
p[r%n]-p[i])-getDot(v, p[l%n]-p[i]);
            double h =
getDistanceToLine (p[j%n], p[i],
p[i+1]);
            area = min(area, w * h);
            perimeter = min(perimeter,
2 * w + 2 * h);
        }
    }

    Polygon cutPolygon (Polygon u,
Point a, Point b) {
        Polygon ret;
        int n = u.size();
        for (int i = 0; i < n; i++) {
            Point c = u[i], d =
u[(i+1)%n];
            if (dcmp((b-a)*(c-a)) >= 0)
ret.push_back(c);
            if (dcmp((b-a)*(d-c)) != 0)
{
                Point t;
                getIntersection(a, b-a,
c, d-c, t);
                if (onSegment(t, c, d))
                    ret.push_back(t);
            }
        }
        return ret;
    }

    int halfPlaneIntersection(DirLine*
li, int n, Point* poly) {
        sort(li, li + n);

        int first, last;
        Point* p = new Point[n];
        DirLine* q = new DirLine[n];
        q[first=last=0] = li[0];

        for (int i = 1; i < n; i++) {
            while (first < last &&
!onLeft(li[i], p[last-1])) last--;
            while (first < last &&
!onLeft(li[i], p[first])) first++;
            q[++last] = li[i];

            if (dcmp(q[last].v *
q[last-1].v) == 0) {
                last--;
                if (onLeft(q[last],
li[i].p)) q[last] = li[i];
            }

            if (first < last)
                getIntersection(q[last-
1].p, q[last-1].v, q[last].p,
q[last].v, p[last-1]);
        }

        while (first < last &&
!onLeft(q[first], p[last-1])) last--;
        if (last - first <= 1) { delete
[] p; delete [] q; return 0; }
        getIntersection(q[last].p,
q[last].v, q[first].p, q[first].v,
p[last]);

        int m = 0;
        for (int i = first; i <= last;
i++) poly[m++] = p[i];
        delete [] p; delete [] q;
        return m;
    }

    Polygon simplify (const Polygon&
poly) {
        Polygon ret;
        int n = poly.size();
        for (int i = 0; i < n; i++) {
            Point a = poly[i];
            Point b = poly[(i+1)%n];
            Point c = poly[(i+2)%n];
            if (dcmp((b-a)*(c-b)) != 0
&& (ret.size() == 0 || b !=
ret[ret.size()-1]))
                ret.push_back(b);
        }
        return ret;
    }

    Point ComputeCentroid( Point* p,int
n){
        Point c(0,0);
        double scale = 6.0 *
getSignedArea(p,n);
        for (int i = 0; i < n; i++){
            int j = (i+1) % n;
            c = c +
(p[i]+p[j])*(p[i].x*p[j].y -
p[j].x*p[i].y);
        }
        return c / scale;
    }

    // Tested :
https://www.spoj.com/problems/INOROUT
    // pt must be in ccw order with no
three collinear points
    // returns inside = 1, on = 0,
outside = -1
    int pointInConvexPolygon(Point* pt,
int n, Point p){
        assert(n >= 3);
        int lo = 1 , hi = n - 1 ;
        while(hi - lo > 1){
            int mid = (lo + hi) / 2;
            if(getCross(pt[mid] -
pt[0], p - pt[0]) > 0) lo = mid;
            else hi = mid;
        }

        bool in =
pointInTriangle(pt[0], pt[lo], pt[hi],
p);
        if(!in) return -1;


        if(getCross(pt[lo] - pt[lo-1],
p - pt[lo-1]) == 0) return 0;
        if(getCross(pt[hi] - pt[lo], p
- pt[lo]) == 0) return 0;
        if(getCross(pt[hi] -
pt[(hi+1)%n], p - pt[(hi+1)%n]) == 0)
return 0;
```

```
        return 1;
    }

    // Tested :
https://toph.co/p/cover-the-points
    // Calculate [ACW, CW] tangent pair
from an external point
    #define CW          -1
    #define ACW          1
    int direction(Point st, Point ed,
Point q)       {return dcmp(getCross(ed
- st, q - ed));}
    bool isGood(Point u, Point v, Point
Q, int dir)  {return direction(Q, u, v)
!= -dir;}
    Point better(Point u, Point v,
Point Q, int dir) {return direction(Q,
u, v) == dir ? u : v;}

    Point tangents(Point* pt, Point Q,
int dir, int lo, int hi){
        while(hi - lo > 1){
            int mid = (lo + hi)/2;
            bool pvs = isGood(pt[mid],
pt[mid - 1], Q, dir);
            bool nxt = isGood(pt[mid],
pt[mid + 1], Q, dir);

            if(pvs && nxt) return
pt[mid];
            if(!(pvs || nxt)){
                Point p1 = tangents(pt,
Q, dir, mid+1, hi);
                Point p2 = tangents(pt,
Q, dir, lo, mid - 1);
                return better(p1, p2,
Q, dir);
            }

            if(!pvs){
                if(direction(Q,
pt[mid], pt[lo]) == dir)  hi = mid - 1;
                else if(better(pt[lo],
pt[hi], Q, dir) == pt[lo]) hi = mid -
1;
                else lo = mid + 1;
            }
            if(!nxt){
                if(direction(Q,
pt[mid], pt[lo]) == dir)  lo = mid + 1;
                else if(better(pt[lo],
pt[hi], Q, dir) == pt[lo]) hi = mid -
1;
                else lo = mid + 1;
            }
        }

        Point ret = pt[lo];
        for(int i = lo + 1; i <= hi;
i++) ret = better(ret, pt[i], Q, dir);
        return ret;
    }

    // [ACW, CW] Tangent
    pair<Point, Point>
get_tangents(Point* pt, int n, Point
Q){
        Point acw_tan = tangents(pt, Q,
ACW, 0, n - 1);
        Point cw_tan = tangents(pt, Q,
CW, 0, n - 1);
        return make_pair(acw_tan,
cw_tan);
    }
};

struct Circle {
    Point o;
    double r;
    Circle () {}
    Circle (Point o, double r = 0):
o(o), r(r) {}
    void read () { o.read(),
scanf("%lf", &r); }
    Point point(double rad) { return
Point(o.x + cos(rad)*r, o.y +
sin(rad)*r); }
    double getArea (double rad) {
return rad * r * r / 2; }
    //area of the circular sector cut
by a chord with central angle alpha
    double sector(double alpha) {return
r * r * 0.5 * (alpha - sin(alpha));}
};

namespace Circular {
    using namespace Linear;
    using namespace Vectorial;
    using namespace Triangular;

    int getLineCircleIntersection
(Point p, Point q, Circle O, double&
t1, double& t2, vector<Point>& sol) {
        Vector v = q - p;
        //sol.clear();
        double a = v.x, b = p.x -
O.o.x, c = v.y, d = p.y - O.o.y;
        double e = a*a+c*c, f =
2*(a*b+c*d), g = b*b+d*d-O.r*O.r;
        double delta = f*f - 4*e*g;
        if (dcmp(delta) < 0) return 0;
        if (dcmp(delta) == 0) {
            t1 = t2 = -f / (2 * e);
            sol.push_back(p + v * t1);
            return 1;
        }

        t1 = (-f - sqrt(delta)) / (2 *
e); sol.push_back(p + v * t1);
        t2 = (-f + sqrt(delta)) / (2 *
e); sol.push_back(p + v * t2);
        return 2;
    }

    // signed area of intersection of
circle(c.o, c.r) and
    // triangle(c.o, s.a, s.b)
[cross(a-o, b-o)/2]
    double
areaCircleTriIntersection(Circle c,
Segment s){
        using namespace Linear;
        double OA = getLength(c.o -
s.a);
        double OB = getLength(c.o -
s.b);
```

```
        // sector
        if
(dcmp(getDistanceToSegment(c.o, s.a,
s.b) - c.r) >= 0)
            return
fix_acute(getSignedAngle(s.a - c.o, s.b
- c.o)) * (c.r*c.r) / 2.0;

        // triangle
        if (dcmp(OA - c.r) <= 0 &&
dcmp(OB - c.r) <= 0)
            return getCross(c.o-
s.b,s.a-s.b) / 2.0;

        // three part: (A, a) (a, b)
(b, B)
        vector<Point>Sect; double
t1,t2;
        getLineCircleIntersection(s.a,
s.b, c, t1, t2, Sect);
        return
areaCircleTriIntersection(c,
Segment(s.a, Sect[0]))
            +
areaCircleTriIntersection(c,
Segment(Sect[0], Sect[1]))
            +
areaCircleTriIntersection(c,
Segment(Sect[1], s.b));
    }

    // area of intersecion of
circle(c.o, c.r) and simple
polyson(p[])
    // Tested : ZOJ 2675 - Little
Mammoth
    double areaCirclePolygon(Circle c,
Polygon p){
        double res = .0;
        int n = p.size();
        for (int i = 0; i < n; ++ i)
            res +=
areaCircleTriIntersection(c,
Segment(p[i], p[(i+1)%n]));
        return fabs(res);
    }

    // interior          (d < R - r)
----> -2
    // interior tangents (d = R - r)
----> -1
    // concentric        (d = 0)
    // secants           (R - r < d < R
+ r) ---->  0
    // exterior tangents (d = R + r)
---->  1
    // exterior          (d > R + r)
---->  2
    int getPos(Circle o1, Circle o2) {
        using namespace Vectorial;
        double d = getLength(o1.o -
o2.o);
        int in = dcmp(d - fabs(o1.r -
o2.r)), ex = dcmp(d - (o1.r + o2.r));
        return in<0 ? -2 : in==0? -1 :
ex==0 ? 1 : ex>0? 2 : 0;
    }

    int getCircleCircleIntersection
(Circle o1, Circle o2, vector<Point>&
sol) {
        double d = getLength(o1.o -
o2.o);
        if (dcmp(d) == 0) {
            if (dcmp(o1.r - o2.r) == 0)
return -1;
            return 0;
        }
        if (dcmp(o1.r + o2.r - d) < 0)
return 0;
        if (dcmp(fabs(o1.r-o2.r) - d) >
0) return 0;

        Vector v = o2.o - o1.o;
        double co = (o1.r*o1.r +
getPLength(v) - o2.r*o2.r) / (2 * o1.r
* getLength(v));
        double si = sqrt(fabs(1.0 -
co*co));
        Point p1 = scale(cw(v,co, si),
o1.r) + o1.o;
        Point p2 = scale(ccw(v,co, si),
o1.r) + o1.o;

        sol.push_back(p1);
        if (p1 == p2) return 1;
        sol.push_back(p2);
        return 2;
    }

    double areaCircleCircle(Circle o1,
Circle o2){
        Vector AB = o2.o - o1.o;
        double d = getLength(AB);
        if(d >= o1.r + o2.r) return 0;
        if(d + o1.r <= o2.r) return pi
* o1.r * o1.r;
        if(d + o2.r <= o1.r) return pi
* o2.r * o2.r;

        double alpha1 = acos((o1.r *
o1.r + d * d - o2.r * o2.r) / (2.0 *
o1.r * d));
        double alpha2 = acos((o2.r *
o2.r + d * d - o1.r * o1.r) / (2.0 *
o2.r * d));
        return o1.sector(2*alpha1) +
o2.sector(2*alpha2);
    }

    int getTangents (Point p, Circle o,
Vector* v) {
        Vector u = o.o - p;
        double d = getLength(u);
        if (d < o.r) return 0;
        else if (dcmp(d - o.r) == 0) {
            v[0] = rotate(u, pi / 2);
            return 1;
        } else {
            double ang = asin(o.r / d);
            v[0] = rotate(u, -ang);
            v[1] = rotate(u, ang);
            return 2;
        }
    }
    int getTangentPoints (Point p,
Circle o, vector<Point>& v) {
```

```
        Vector u = p - o.o ;
        double d = getLength(u);
        if (d < o.r) return 0;
        else if (dcmp(d - o.r) == 0) {
            v.push_back(o.o+u);
            return 1;
        } else {
            double ang = acos(o.r / d);
            u = u / getLength(u) * o.r;
            v.push_back(o.o+rotate(u, -
ang));
            v.push_back(o.o+rotate(u,
ang));
            return 2;
        }
    }

    int getTangents (Circle o1, Circle
o2, Point* a, Point* b) {
        int cnt = 0;
        if (dcmp(o1.r-o2.r) < 0) {
swap(o1, o2); swap(a, b); }
        double d2 = getPLength(o1.o -
o2.o);
        double rdif = o1.r - o2.r, rsum
= o1.r + o2.r;
        if (dcmp(d2 - rdif * rdif) < 0)
return 0;
        if (dcmp(d2) == 0 && dcmp(o1.r
- o2.r) == 0) return -1;

        double base = getAngle(o2.o -
o1.o);
        if (dcmp(d2 - rdif * rdif) ==
0) {
            a[cnt] = o1.point(base);
b[cnt] = o2.point(base); cnt++;
            return cnt;
        }

        double ang = acos( (o1.r -
o2.r) / sqrt(d2) );
        a[cnt] = o1.point(base+ang);
b[cnt] = o2.point(base+ang); cnt++;
        a[cnt] = o1.point(base-ang);
b[cnt] = o2.point(base-ang); cnt++;

        if (dcmp(d2 - rsum * rsum) ==
0) {
            a[cnt] = o1.point(base);
b[cnt] = o2.point(pi+base); cnt++;
        }
        else if (dcmp(d2 - rsum * rsum)
> 0) {
            double ang = acos( (o1.r +
o2.r) / sqrt(d2) );
            a[cnt] =
o1.point(base+ang); b[cnt] =
o2.point(pi+base+ang); cnt++;
            a[cnt] = o1.point(base-
ang); b[cnt] = o2.point(pi+base-ang);
cnt++;
        }
        return cnt;
    }

    Circle CircumscribedCircle(Point
p1, Point p2, Point p3) {
        double Bx = p2.x - p1.x, By =
p2.y - p1.y;
        double Cx = p3.x - p1.x, Cy =
p3.y - p1.y;
        double D = 2 * (Bx * Cy - By *
Cx);
        double cx = (Cy * (Bx * Bx + By
* By) - By * (Cx * Cx + Cy * Cy)) / D +
p1.x;
        double cy = (Bx * (Cx * Cx + Cy
* Cy) - Cx * (Bx * Bx + By * By)) / D +
p1.y;
        Point p = Point(cx, cy);
        return Circle(p, getLength(p1 -
p));
    }

    Circle InscribedCircle(Point p1,
Point p2, Point p3) {
        double a = getLength(p2 - p3);
        double b = getLength(p3 - p1);
        double c = getLength(p1 - p2);
        Point p = (p1 * a + p2 * b + p3
* c) / (a + b + c);
        return Circle(p,
getDistanceToLine(p, p1, p2));
    }

    //distance From P : distance from Q
=  rp : rq
    Circle getApolloniusCircle(const
Point& P,const Point& Q, double rp,
double rq ){
        rq *= rq ;
        rp *= rp ;
        double a = rq - rp ;
        assert(dcmp(a));
        double g = rq * P.x - rp * Q.x
; g /= a ;
        double h = rq * P.y - rp * Q.y
; h /= a ;
        double c = rq*P.x*P.x-
rp*Q.x*Q.x+rq*P.y*P.y-rp*Q.y*Q.y ;
        c /= a ;
        Point o(g,h);
        double R = g*g +h*h - c ;
        R = sqrt(R);
        return Circle(o,R);
    }
};
```

Starting Template:

Create “debug.h” in the same directory:

```
#include <bits/stdc++.h>
using namespace std;
void __print(int x) { cerr << x; }
void __print(long x) { cerr << x; }
void __print(long long x) { cerr << x;
}
void __print(unsigned x) { cerr << x; }
void __print(unsigned long x) { cerr <<
x; }
void __print(unsigned long long x) {
cerr << x; }
void __print(float x) { cerr << x; }
void __print(double x) { cerr << x; }
void __print(long double x) { cerr <<
x; }
```

```
void __print(char x) { cerr << '\'' <<
x << '\''; }
void __print(const char *x) { cerr <<
'\"' << x << '\"'; }
void __print(const string &x) { cerr <<
'\"' << x << '\"'; }
void __print(bool x) { cerr << (x ?
"true" : "false"); }

template <typename T, typename V>
void __print(const pair<T, V> &x) {
    cerr << '{';
    __print(x.first);
    cerr << ',';
    __print(x.second);
    cerr << '}';
}
template <typename T>
void __print(const T &x) {
    int f = 0;
    cerr << '{';
    for (auto &i : x) cerr << (f++ ?
"," : ""), __print(i);
    cerr << "}";
}
void _print() { cerr << "]\n"; }//one
template <typename T, typename... V>
void _print(T t, V... v) { //one
    __print(t);
    if (sizeof...(v)) cerr << ", ";
    _print(v...); //one
}
#define debug(x...)                  \
    cerr << "[" << #x << "] = ["; \
    _print(x) //one
// add a newline at the end
```

Now the main template:

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

// g++ -DLOCAL -O3 -std=c++14 .\a.cpp
#ifdef LOCAL
#include "debug.h"
#else
#define debug(...)
#endif

#define endl "\n"
typedef long long ll;
void solve() {}
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
#ifdef LOCAL
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
#endif
    int t; cin >> t;
    for (int tc = 1; tc <= t; tc++) {
        // cout << "Case " << tc << ":
";
        solve();
    }
}
```

Bit Hacks:

- `x |= (1 << b)` sets bit b.
- `x &= ~(1 << b)` resets bit b.
- `x ^= (1 << b)` toggles bit b.
- `x & -x` is the LSB in x.
- `if (x && (!((x – 1) & x)))` checks whether x is a power of 2.
- `x = x & (x – 1)` turns off the lowest set bit.
- `cout << bitset<8>(x)` prints a number in binary format.
- `__builtin_popcount(x)` returns number of ones in x’s binary. For long long, use `__builtin_popcountll(x)`.
- `__builtin_clz` (leading 0), `__builtin_ctz`(trailing 0