Team notebook

May 30, 2024

Contents

1 Algebra

1.1 Binary Pow

```
/*
	Description: a ^ b % m
	Time: O(log(N))
	Space: O(1)
*/

ll binary_pow(ll a, ll b, ll m) {
	a %= m;
	ll res = 1;
	while (b > 0) {
```

```
		if (b & 1)
			res = res * a % m;
		a = a * a % m;
		b >>= 1;
	}
	return res;
}
```

1.2 Chinese Remainder Theorem

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

/*
	Description: -
	Time: O(M)
	Space: O(1)
*/

// Import
ll gcd_extended(ll a, ll b, ll& x, ll& y);

pair<ll, ll> chinese_remainder_theorem(vector<pair<ll,
     ll>> congruences) {
	// congruences = [(a1,m1),(a2,m2),...]

	ll p, q;
	auto [a, m] = congruences.back();
	congruences.pop_back();
	while (!congruences.empty())
	{
		auto [ta, tm] = congruences.back();
		congruences.pop_back();
		ll g = gcd_extended(m, tm, p, q);
		if ((a - ta) % g != 0)
		{
			return { -1,-1 };
		}

		ll nm = m / g * tm;
		a = ((a * (tm / g) * q) % nm) + ((ta *
		    (m / g) * p) % nm);
		a = (a + nm + nm) % nm;
		m = nm;
	}

	return { a, m };
}

int main() {
	vector<pair<ll, ll>> congruences;
	congruences.push_back({ 11,23 });
	congruences.push_back({ 1,81 });
	congruences.push_back({ 9,97 });

	auto x = chinese_remainder_theorem(congruences);
	for (auto y : congruences) {
		cout << x.first << " % " << y.second <<
		    " = " << x.first % y.second << endl;
	}
}
```

1.3 Factorial Mod

```
/*
	Description: -
	Time: O(N * log(P))
	Space: O(N)
	https://cp-algorithms.com/algebra/factorial-modulo.html
*/

vector<int> f;

// O(p)
int calc_once(ll n, ll p) {
	f.push_back(1);
	for (int i = 1; i < p; i++)
		f.push_back(f[i - 1] * i % p);

	factorial_mod(n, p);
}

// O(N * log(P))
int factorial_mod(ll n, ll p) {
	// p must be prime
	ll res = 1;
	while (n > 1) {
		if ((n / p) % 2)
			res = p - res;
		res = res * f[n % p] % p;
		n /= p;
	}
	return res;
}
```

1.4 Factorize Pollard Rho

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

/*
	Description: -
	Time: O(sqrt(N))
	Space: O(1)
*/

ll mult(ll a, ll b, ll mod) {
	return (__int128)a * b % mod;
}

ll f(ll x, ll c, ll mod) {
	return (mult(x, x, mod) + c) % mod;
}

ll rho(ll n, ll x0 = 2, ll c = 1) {
	ll x = x0;
	ll y = x0;
	ll g = 1;

	while (g == 1) {
		x = f(x, c, n);
		y = f(y, c, n);
		y = f(y, c, n);
		g = gcd(abs(x - y), n);
	}

	if (g == n) {
		return rho(n, x0 + 1, c);
	}

	return g;
}

int main() {
	cout << rho(2206637);
}
```

1.5 Fibonacci

```
/*
	Description: Calculate the nth number in
	    fibonacci sequence
	Time: O(log(N))
	Space: O(1)
*/

pair<int, int> fib(int n) {
	// returns F_n and F_n+1
```

```
    if (n == 0)
        return { 0, 1 };

    auto p = fib(n >> 1);
    int c = p.first * (2 * p.second - p.first);
    int d = p.first * p.first + p.second * p.second;
    if (n & 1)
        return { d, c + d };
    else
        return { c, d };
}
```

1.6 GCD Extended

```
/*
    Description: -
    Time: O(log(a) + log(b))
    Space: O(1)
*/

int gcd_extended(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;
}
```

1.7 Gausian Elimination

```
const int MAX_N = 3;
// adjust as needed
struct AugmentedMatrix { double mat[MAX_N][MAX_N + 1];
    };
struct ColumnVector { double vec[MAX_N]; };
ColumnVector GaussianElimination(int N, AugmentedMatrix
    Aug) {
    // input: N, Augmented Matrix Aug, output:
        Column vector X, the answer
    for (int i = 0; i < N - 1; ++i) {
        // forward elimination
        int l = i;
        for (int j = i + 1; j < N; ++j)
            // row with max col value
            if (fabs(Aug.mat[j][i]) >
                fabs(Aug.mat[l][i]))
                l = j;
        // remember this row l
        // swap this pivot row, reason: minimize
            floating point error
        for (int k = i; k <= N; ++k)
            swap(Aug.mat[i][k],
                Aug.mat[l][k]);
        for (int j = i + 1; j < N; ++j)
            // actual fwd elimination
            for (int k = N; k >= i; --k)
                Aug.mat[j][k] -=
                    Aug.mat[i][k] *
                    Aug.mat[j][i] /
                    Aug.mat[i][i];
    }
    ColumnVector Ans;
    // back substitution phase
    for (int j = N - 1; j >= 0; --j) {
        // start from back
        double t = 0.0;
        for (int k = j + 1; k < N; ++k)
            t += Aug.mat[j][k] * Ans.vec[k];
        Ans.vec[j] = (Aug.mat[j][N] - t) /
            Aug.mat[j][j]; // the answer is here
    }
    return Ans;
}
```

1.8 Log Mod

```
/*
    Description: Returns x for which ((a^x = b mod
        m)), a and m are co-prime
    Time: O(sqrt(m) * log(m))
    Space: O(sqrt(m))
*/

ll binary_power(ll a, ll b, ll m);

ll discrete_log(ll a, ll b, ll m) {
    a %= m, b %= m;
    ll n = sqrt(m) + 1;
    map<ll, ll> vals; // saves only a single value,
        map<ll, vector<ll>>

    for (ll p = 1; p <= n; ++p)
        vals[binary_power(a, p * n, m)] = p;

    for (ll q = 0; q <= n; ++q) {
        ll cur = binary_power(a, q, m) * b % m;
        if (vals.count(cur)) {
            ll ans = vals[cur] * n - q;
            return ans;
        }
    }

    return -1;
}

// calculate minimum x by mod phi(m)
```

1.9 Mod Inverse

```
/*
    Description: -
    Time: O(log(m))
    Space: O(1)
*/

// Import
ll binary_power(ll a, ll b, ll m);
ll phi(ll n);

ll mod_inv(ll num, ll mod) {
    return binary_power(num, phi(mod) - 1, mod);
}
```

1.10 Phi Totient

```
/*
    Description: -
    TODO: Time: O(?)
    TODO: Space: O(?)
*/

ll phi(ll n) {
    ll result = n;
    for (ll i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            while (n % i == 0)
                n /= i;
            result -= result / i;
        }
    }
```

```
        if (n > 1)
                result -= result / n;
        return result;
}

// TODO
1_to_n(int n) {
        vector<int> phi_vec(n + 1);
        for (int i = 0; i <= n; i++)
                phi_vec[i] = i;

        for (int i = 2; i <= n; i++) {
                if (phi_vec[i] == i) {
                        for (int j = i; j <= n; j += i)
                                phi_vec[j] -= phi_vec[j] /
                                     i;
                }
        }
}
```

1.11 Prime-Check Miller

```
/*
        Description: Checks for being prime in O(1) for
             64 bit numbers
        Time: O(1)
        Space: O(1)
*/

u64 binary_power(u64 base, u64 e, u64 mod) {
        u64 result = 1;
        base %= mod;
        while (e) {
                if (e & 1)
                        result = (u128)result * base %
                             mod;
                base = (u128)base * base % mod;
                e >>= 1;
        }
        return result;
}

bool check_composite(u64 n, u64 a, u64 d, int s) {
        u64 x = binary_power(a, d, n);
        if (x == 1 || x == n - 1)
                return false;
        for (int r = 1; r < s; r++) {
                x = (u128)x * x % n;
                if (x == n - 1)
                        return false;
        }
        return true;
};

bool MillerRabin(u64 n) { // returns true if n is
     prime, else returns false.
        if (n < 2)
                return false;

        int r = 0;
        u64 d = n - 1;
        while ((d & 1) == 0) {
                d >>= 1;
                r++;
        }

        for (int a : {2, 3, 5, 7, 11, 13, 17, 19, 23,
             29, 31, 37}) {
                if (n == a)
                        return true;
                if (check_composite(n, a, d, r))
                        return false;
        }
        return true;
}
```

1.12 Primitive Root Mod

```
/*
        Description: Returns a Generator (g),that for
             every (a)
                                there exist a (k) S.T.
                                     ((g^k = a mod m))
                                generator exists if and
                                     only if m=1,2,4 or
                                     for some odd prime
                                     (p)
                                m=p^k , 2*p^k
        Time: O(log(m) ^ 6)
        Space: O(sqrt(m))
*/

// Import
ll binary_power(ll a, ll b, ll m);
ll phi(ll n);

ll generator(ll m) {
        vector<ll> fact;
        ll totiont = phi(m), n = totiont;
        for (ll i = 2; i * i <= n; ++i)
                if (n % i == 0) {
                        fact.push_back(i);
                        while (n % i == 0)
                                n /= i;
                }
        if (n > 1)
                fact.push_back(n);

        for (ll res = 2; res <= m; ++res) {
                bool ok = true;
                for (size_t i = 0; i < fact.size() &&
                     ok; ++i)
                        ok &= binary_power(res, totiont /
                             fact[i], p) != 1;

                if (ok) return res;
        }

        return -1;
}
```

1.13 Sieve Linear

```
/*
        Description: Factorizes all numbers [2,n]
        Time: O(N)
        Space: O(N)
*/

void sieve_liner(int n) {
        vector<int> leastPrime(n + 1);
        vector<int> pr;

        for (int i = 2; i <= n; ++i) {
                if (leastPrime[i] == 0) {
                        leastPrime[i] = i;
                        pr.push_back(i);
                }
                for (int j = 0; i * pr[j] <= n; ++j) {
                        leastPrime[i * pr[j]] = pr[j];
                        if (pr[j] == leastPrime[i]) {
                                break;
                        }
                }
        }
}
```

1.14 Sieve

```
/*
```

```
	Description: Check whether number is prime or
	     not
	Time: < O(N)
	Space: O(N)
*/

void sieve(int n) {
	vector<bool> is_prime(n + 1, true);
	is_prime[0] = is_prime[1] = false;

	for (int i = 2; i * i <= n; i++) {
		if (is_prime[i]) {
			for (int j = i * i; j <= n; j +=
			     i)
				is_prime[j] = false;
		}
	}
}
```

2 Algorithms

2.1 Longest Increasing Subsequence (LIS)

```
/*
	Description: -
	Time: O(N * log(N))
	Space: O(N)
*/

int longest_increasing_subsequence(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++) {
		for (int l = 1; l <= n; l++) {
			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;
}
```

3 Combinatrics

3.1 Fact Divisors

```
/*
	Description: -
	Time: O(log_k(n))
	Space: O(1)
*/

int fact_pow(int n, int k) {
	int res = 0;
	while (n) {
		n /= k;
		res += n;
	}
	return res;
}

// TODO
```

3.2 binomial coefficient

3.2.1 Binomial

```
/*
	Description: -
	Time: O(K)
	Space: O(1)
*/

int Binomial_Coefficient(int n, int k) {
	double res = 1;
	for (int i = 1; i <= k; ++i)
		res = res * (n - k + i) / i;
	return (int)(res + 0.01);
}
```

3.2.2 Large Modulo

```
/*
	Description: -
	TODO Time: O(K)
	TODO Space: O(1)
*/

#define ll long long

factorial[0] = 1;
for (int i = 1; i <= MAXN; i++) {
	factorial[i] = factorial[i - 1] * i % m;
}
//solving for one inverse
long long binomial_coefficient(int n, int k) {
	return factorial[n] * inverse(factorial[k] *
	     factorial[n - k] % m) % m;
}

//saving all inverse factorials
long long binomial_coefficient(int n, int k) {
	return factorial[n] * inverse_factorial[k] % m
	     * inverse_factorial[n - k] % m;
}

//returns inverse of vector "a"
std::vector<ll> inverse(const std::vector<ll> &a, ll m)
    {
	ll n = a.size();
	if (n == 0) return {};
	std::vector<ll> b(n), prem(n + 1, 1), sufm(n +
	     1, 1);
	ll all_m = 1;
	for (int i = 0; i < n; i++)
	{
		all_m = (a[i] * all_m) % m;
		prem[i + 1] = (prem[i] * a[i]) % m;
		sufm[i + 1] = (sufm[i] * a[n - i - 1]) %
		     m;
	}
	all_m = inv(all_m, m);
	for (int i = 0; i < n; i++)
	{
		b[i] = ((prem[i] * sufm[n - i - 1]) % m
		     * all_m) % m;
	}
	return b;
}
```

3.2.3 Lucas Theorem

```
// FIXME: DELETE

/*
```

```
	Description: A Lucas Theorem based solution to
	    compute nCr % p
	Time: O(log_p(N))
	Space: O(r)
*/

int nCr_Modp_DP(int n, int r, int p) {
	int C[r + 1];
	memset(C, 0, sizeof(C));

	C[0] = 1;

	for (int i = 1; i <= n; i++) {
		for (int j = min(i, r); j > 0; j--)
			C[j] = (C[j] + C[j - 1]) % p;
	}
	return C[r];
}

int nCr_Modp_Lucas(int n, int r, int p) {
	if (r == 0)
		return 1;

	int ni = n % p, ri = r % p;

	return (nCr_Modp_Lucas(n / p, r / p, p) *
	    nCr_Modp_DP(ni, ri, p)) % p;
}
```

4 Data Structures

4.1 DSU

```
// TODO

struct Dsu {
	vector<int> p;

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

	int Find(int x) { return x == p[x] ? x : p[x] =
	    Find(p[x]); }

	int Join(int x, int y) {
		int px = Find(x), py = Find(y);
			if (px == py) return 0;
			p[px] = py;
			return 1;
		}
};
```

4.2 Rope

```
#include <ext/rope> //header with rope
using namespace std;
using namespace __gnu_cxx; //namespace with rope and
    some additional stuff

int main()
{
	ios_base::sync_with_stdio(false);
	rope <int> v; //use as usual STL container
	int n, m;
	cin >> n >> m;
	for (int i = 1; i <= n; ++i)
		v.push_back(i); //initialization
	int l, r;
	for (int i = 0; i < m; ++i)
	{
		cin >> l >> r;
		--l, --r;
		rope <int> cur = v.substr(l, r - l + 1);
		v.erase(l, r - l + 1);
		v.insert(v.mutable_begin(), cur);
	}
	for (rope <int>::iterator it =
	    v.mutable_begin(); it != v.mutable_end();
	    ++it)
		cout << *it << " ";
	return 0;
}
```

4.3 Sparse Table

```
/*
	Description: -
	Time: Construction: O(N * log(N))
	      Query: O(1)
	Space: O(N)
*/

vector<vector<ll>>buildSparseTable(vector<ll> array,
    const ll& (*func)(const ll&, const ll&)) {
	vector<vector<ll>> sparseTable;
	sparseTable.push_back(array);
	ll n = array.size(), j = 1;

	while ((1 << j) <= n) {
		sparseTable.push_back(vector<ll>());
		for (ll i = 0; i < n - (1 << j) + 1;
		    i++) {
			sparseTable[j].push_back(func(sparseTable
			    - 1][i], sparseTable[j -
			    1][i + (1 << (j - 1))]));
		}
		j++;
	}

	return sparseTable;
}

ll querySparseTable(vector<vector<ll>>& st, ll i, ll j,
    const ll& (*func)(const ll&, const ll&)) {
	ll l = j - i + 1, k = 0;
	while ((1 << (k + 1)) <= l)k += 1;

	return func(st[k][i], st[k][j - (1 << k) + 1]);
}

int main() {
	vector<ll> arr = { 1,2,3,4,5 };
	auto st = buildSparseTable(arr, max);
	int i = 0, j = 4;
	querySparseTable(st, i, j, max);
}
```

4.4 Trees

4.4.1 Fenwick

```
#include <bits/stdc++.h>
using namespace std;
#define LSOne(S) ((S) & -(S))// the key operation
typedef long long ll;
typedef vector<ll> vll;
typedef vector<int> vi;// for extra flexibility
class FenwickTree {
private:
	vll ft;
public:
	FenwickTree(int m) { ft.assign(m + 1, 0); }//
	    index 0 is not used
	void build(const vll& f) {
		int m = (int)f.size() - 1;
```

```
        ft.assign(m + 1, 0);
        for (int i = 1; i <= m; ++i) {
            ft[i] += f[i];
            if (i + LSOne(i) <= m)
                ft[i + LSOne(i)] += ft[i];
        }
    }
    // internal FT is an array
    // create an empty FT
    // note f[0] is always 0
    // O(m)
    // add this value
    // i has parent
    // add to that parent
    FenwickTree(const vll& f) { build(f); }//
        create FT based on f
    FenwickTree(int m, const vi& s) {
        vll f(m + 1, 0);
        for (int i = 0; i < (int)s.size(); ++i)
            ++f[s[i]];
        build(f);
    }// create FT based on s
    ll rsq(int j) {
        ll sum = 0;
        for (; j; j -= LSOne(j))
            sum += ft[j];
        return sum;
    }
    ll rsq(int i, int j) { return rsq(j) - rsq(i -
        1); } // inc/exclusion
    // updates value of the i-th element by v (v
        can be +ve/inc or -ve/dec)
    void update(int i, ll v) {
        for (; i < (int)ft.size(); i += LSOne(i))
            ft[i] += v;
    }
    int select(ll k) {
        int lo = 1, hi = ft.size() - 1;
        for (int i = 0; i < 30; ++i) {
            int mid = (lo + hi) / 2;
            (rsq(1, mid) < k) ? lo = mid : hi
                = mid;
        }
        return hi;
    }
    // O(log^2 m)
    // 2^30 > 10^9; usually ok
    // BSTA
    // See Section 3.3.1
};
class RUPQ {
private:
    FenwickTree ft;
public:
    RUPQ(int m) : ft(FenwickTree(m)) {}
    void range_update(int ui, int uj, int v) {
        ft.update(ui, v);
        ft.update(uj + 1, -v);
    }
    ll point_query(int i) { return ft.rsq(i); }
};
// RUPQ variant
// internally use PURQ FT
// [ui, ui+1, .., m] +v
// [uj+1, uj+2, .., m] -v
// [ui, ui+1, .., uj] +v
// rsq(i) is sufficient
class RURQ {
    // RURQ variant
private:
    // needs two helper FTs
    RUPQ rupq;
    // one RUPQ and
    FenwickTree purq;
    // one PURQ
public:
    RURQ(int m) : rupq(RUPQ(m)),
        purq(FenwickTree(m)) {} // initialization
    void range_update(int ui, int uj, int v) {
        rupq.range_update(ui, uj, v);
        // [ui, ui+1, .., uj] +v
        purq.update(ui, v * (ui - 1));
        // -(ui-1)*v before ui
        purq.update(uj + 1, -v * uj);
        // +(uj-ui+1)*v after uj
    }
    ll rsq(int j) {
        return rupq.point_query(j) * j -
            // initial calculation
            purq.rsq(j);
        // cancelation factor
    }
    ll rsq(int i, int j) { return rsq(j) - rsq(i -
        1); } // standard
};

int main() {
    vll f = { 0,0,1,0,1,2,3,2,1,1,0 };
    // index 0 is always 0
    FenwickTree ft(f);
    printf("%lld\n", ft.rsq(1, 6)); // 7 =>
        ft[6]+ft[4] = 5+2 = 7
    printf("%d\n", ft.select(7)); // index 6,
        rsq(1, 6) == 7, which is >= 7
    ft.update(5, 1); // update demo
    printf("%lld\n", ft.rsq(1, 10)); // now 12
    printf("=====\n");
    RUPQ rupq(10);
    RURQ rurq(10);
    rupq.range_update(2, 9, 7); // indices in [2,
        3, .., 9] updated by +7
    rurq.range_update(2, 9, 7); // same as rupq
        above
    rupq.range_update(6, 7, 3); // indices 6&7 are
        further updated by +3 (10)
    rurq.range_update(6, 7, 3); // same as rupq
        above
    // idx = 0 (unused) | 1 | 2 | 3 | 4 | 5 | 6 | 7
        | 8 | 9 |10
    // val = -      | 0 | 7 | 7 | 7 | 7 | 10 | 10 | 7
        | 7 | 0
    for (int i = 1; i <= 10; i++)
        printf("%d -> %lld\n", i,
            rupq.point_query(i));
    printf("RSQ(1, 10) = %lld\n", rurq.rsq(1, 10));
        // 62
    printf("RSQ(6, 7) = %lld\n", rurq.rsq(6, 7));
        // 20
    return 0;
}
```

4.4.2 K-Dimensional Tree

```
const int k = 2; // dimensions

struct Node {
    int point[k]; // To store k dimensional point
    Node* left, * right;
};

struct Node* newNode(int arr[]) {
    struct Node* temp = new Node;

    for (int i = 0; i < k; i++)
        temp->point[i] = arr[i];

    temp->left = temp->right = NULL;
    return temp;
}

Node* insertRec(Node* root, int point[], unsigned
    depth) {
    if (root == NULL)
        return newNode(point);

    unsigned cd = depth % k;
```

```
        if (point[cd] < (root->point[cd]))
                root->left = insertRec(root->left,
                    point, depth + 1);
        else
                root->right = insertRec(root->right,
                    point, depth + 1);

        return root;
}

Node* insert(Node* root, int point[]) {
        return insertRec(root, point, 0);
}

bool arePointsSame(int point1[], int point2[]) {
        for (int i = 0; i < k; ++i)
                if (point1[i] != point2[i])
                        return false;

        return true;
}

bool searchRec(Node* root, int point[], unsigned depth)
     {
        if (root == NULL)
                return false;
        if (arePointsSame(root->point, point))
                return true;

        unsigned cd = depth % k;

        if (point[cd] < root->point[cd])
                return searchRec(root->left, point,
                    depth + 1);

        return searchRec(root->right, point, depth + 1);
}

bool search(Node* root, int point[]) {
        return searchRec(root, point, 0);
}

struct Node* root = NULL;
root = insert(root, points[i]);
int point1[k] = { ... }
search(root, point1)
```

4.4.3 STL Ordered Set

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

using namespace __gnu_pbds;
using namespace std;

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

int main() {
        ordered_set tree;

        tree.insert(3);
        tree.order_of_key(1);
        tree.find_by_order(1);
        tree.size();
        tree.max_size();
        *tree.lower_bound(0);
        *tree.upper_bound(4);
}
```

4.4.4 Segment Tree

```
typedef vector<int> vi;

class SegmentTree {
private:
        int n;
        vi A, st, lazy;
        // OOP style
        // n = (int)A.size()
        // the arrays
        int l(int p) { return p << 1; }
        int r(int p) { return (p << 1) + 1; }
        // go to left child
        // go to right child
        int conquer(int a, int b) {
                if (a == -1) return b;
                if (b == -1) return a;
                return min(a, b);
        }
        // corner case
        // RMQ
        void build(int p, int L, int R) {
                if (L == R)
                        st[p] = A[L];
                else {
                        int m = (L + R) / 2;
                        build(l(p), L, m);
                        build(r(p), m + 1, R);
                        st[p] = conquer(st[l(p)],
                            st[r(p)]);
                }
        }
        void propagate(int p, int L, int R) {
                if (lazy[p] != -1) {
                        st[p] = lazy[p];
                        if (L != R)
                                lazy[l(p)] = lazy[r(p)] =
                                    lazy[p];
                        else
                                A[L] = lazy[p];
                        lazy[p] = -1;
                }
        }
        // O(n)
        // base case
        // has a lazy flag
        // [L..R] has same value
        // not a leaf
        // propagate downwards
        // L == R, a single index
        // time to update this
        // erase lazy flag
        int RMQ(int p, int L, int R, int i, int j) {
                // O(log n)
                propagate(p, L, R);
                // lazy propagation
                if (i > j) return -1;
                // infeasible
                if ((L >= i) && (R <= j)) return st[p];
                // found the segment
                int m = (L + R) / 2;
                return conquer(RMQ(l(p), L, m, i
                        , min(m, j)),
                        RMQ(r(p), m + 1, R, max(i, m +
                            1), j
                        ));
        }
        void update(int p, int L, int R, int i, int j,
            int val) { // O(log n)
                propagate(p, L, R);
                // lazy propagation
                if (i > j) return;
                if ((L >= i) && (R <= j)) {
                        // found the segment
                        lazy[p] = val;
                        // update this
                        propagate(p, L, R);
                        // lazy propagation
```

```
            } else {
                int m = (L + R) / 2;
                update(l(p), L, m, i
                       , min(m, j), val);
                update(r(p), m + 1, R, max(i, m +
                    1), j
                       , val);
                int lsubtree = (lazy[l(p)] != -1)
                    ? lazy[l(p)] : st[l(p)];
                int rsubtree = (lazy[r(p)] != -1)
                    ? lazy[r(p)] : st[r(p)];
                st[p] = (lsubtree <= rsubtree) ?
                    st[l(p)] : st[r(p)];
            }
        }
public:
        SegmentTree(int sz) : n(sz), st(4 * n), lazy(4
             * n, -1) {}
        SegmentTree(const vi& initialA) :
             SegmentTree((int)initialA.size()) {
                A = initialA;
                build(1, 0, n - 1);
        }
        void update(int i, int j, int val) { update(1,
             0, n - 1, i, j, val); }
        int RMQ(int i, int j) { return RMQ(1, 0, n - 1,
             i, j); }
};

int main() {
        vi A = { 18, 17, 13, 19, 15, 11, 20, 99 };
        SegmentTree st(A);
        printf("RMQ(1, 3) = %d\n", st.RMQ(1, 3));
        // remains 13
        printf("RMQ(4, 7) = %d\n", st.RMQ(4, 7));
        // now 15
        printf("RMQ(3, 4) = %d\n", st.RMQ(3, 4));
        // remains 15
}
```

5 Geometry

5.1 Basics

5.1.1 3D Point

```
struct point3d {
        ftype x, y, z;
        point3d() {}
        point3d(ftype x, ftype y, ftype z) : x(x),
             y(y), z(z) {}
        point3d& operator+=(const point3d& t) {
                x += t.x;
                y += t.y;
                z += t.z;
                return *this;
        }
        point3d& operator-=(const point3d& t) {
                x -= t.x;
                y -= t.y;
                z -= t.z;
                return *this;
        }
        point3d& operator*=(ftype t) {
                x *= t;
                y *= t;
                z *= t;
                return *this;
        }
        point3d& operator/=(ftype t) {
                x /= t;
                y /= t;
                z /= t;
                return *this;
        }
        point3d operator+(const point3d& t) const {
                return point3d(*this) += t;
        }
        point3d operator-(const point3d& t) const {
                return point3d(*this) -= t;
        }
        point3d operator*(ftype t) const {
                return point3d(*this) *= t;
        }
        point3d operator/(ftype t) const {
                return point3d(*this) /= t;
        }
};
point3d operator*(ftype a, point3d b) {
        return b * a;
}

ftype dot(point3d a, point3d b) {
        return a.x * b.x + a.y * b.y + a.z * b.z;
}
point3d cross(point3d a, point3d b) {
        return point3d(a.y * b.z - a.z * b.y,
                a.z * b.x - a.x * b.z,
                a.x * b.y - a.y * b.x);
}
ftype triple(point3d a, point3d b, point3d c) {
        return dot(a, cross(b, c));
}
point3d intersect(point3d a1, point3d n1, point3d a2,
     point3d n2, point3d a3, point3d n3) {
        point3d x(n1.x, n2.x, n3.x);
        point3d y(n1.y, n2.y, n3.y);
        point3d z(n1.z, n2.z, n3.z);
        point3d d(dot(a1, n1), dot(a2, n2), dot(a3,
             n3));
        return point3d(triple(d, y, z),
                triple(x, d, z),
                triple(x, y, d)) / triple(n1, n2, n3);
}
```

5.2 Convex Hull

5.2.1 Grahams Scan

```
/*
        Descricomplex<double>ion: -
        Time: O(N * log(N))
        Space: O(N)
*/

int orientation(complex<double> a, complex<double> b,
     complex<double> c) {
        double v = a.real() * (b.imag() - c.imag())
                + b.real() * (c.imag() - a.imag())
                + c.real() * (a.imag() - b.imag());

        if (v < 0) return -1; // clockwise
        if (v > 0) return +1; // counter-clockwise
        return 0;
}

bool cw(complex<double> a, complex<double> b,
     complex<double> c, bool include_collinear) {
        int o = orientation(a, b, c);
        return o < 0 || (include_collinear && o == 0);
}
bool collinear(complex<double> a, complex<double> b,
     complex<double> c) { return orientation(a, b, c)
     == 0; }

void convex_hull(vector<complex<double>>& a, bool
     include_collinear = false) {
        complex<double> p0 = *min_element(a.begin(),
             a.end(), [](complex<double> a,
             complex<double> b) {
                return make_pair(a.imag(), a.real()) <
                    make_pair(b.imag(), b.real());
             });
```

```
    sort(a.begin(), a.end(), [&p0](const
          complex<double>& a, const complex<double>&
          b) {
              int o = orientation(p0, a, b);
              if (o == 0)
                      return (p0.real() - a.real()) *
                              (p0.real() - a.real())
                      + (p0.imag() - a.imag()) *
                              (p0.imag() - a.imag())
                      < (p0.real() - b.real()) *
                              (p0.real() - b.real())
                      + (p0.imag() - b.imag()) *
                              (p0.imag() - b.imag());
              return o < 0;
              });
    if (include_collinear) {
              int i = (int)a.size() - 1;
              while (i >= 0 && collinear(p0, a[i],
                    a.back())) i--;
              reverse(a.begin() + i + 1, a.end());
    }

    vector<complex<double>> st;
    for (int i = 0; i < (int)a.size(); i++) {
              while (st.size() > 1 && !cw(st[st.size()
                    - 2], st.back(), a[i],
                    include_collinear))
                      st.pop_back();
              st.push_back(a[i]);
    }

    if (include_collinear == false && st.size() ==
          2 && st[0] == st[1])
              st.pop_back();

    a = st;
}
```

5.2.2 Monotone Chain

```
// FIXME DELETE

/*
        Descricomplex<double>ion: -
        TODO Time: O(?)
        TODO Space: O(?)
*/

int orientation(complex<double> a, complex<double> b,
      complex<double> c) {
          double v = a.real() * (b.imag() - c.imag()) +
                b.real() * (c.imag() - a.imag()) + c.real()
                * (a.imag() - b.imag());
          if (v < 0) return -1; // clockwise
          if (v > 0) return +1; // counter-clockwise
          return 0;
}

bool cw(complex<double> a, complex<double> b,
      complex<double> c, bool include_collinear) {
          int o = orientation(a, b, c);
          return o < 0 || (include_collinear && o == 0);
}
bool ccw(complex<double> a, complex<double> b,
      complex<double> c, bool include_collinear) {
          int o = orientation(a, b, c);
          return o > 0 || (include_collinear && o == 0);
}

void convex_hull(vector<complex<double>>& a, bool
      include_collinear = false) {
          if (a.size() == 1)
                  return;

          sort(a.begin(), a.end(), [](complex<double> a,
                complex<double> b) {
                  return make_pair(a.real(), a.imag()) <
                        make_pair(b.real(), b.imag());
                  });
          complex<double> p1 = a[0], p2 = a.back();
          vector<complex<double>> up, down;
          up.push_back(p1);
          down.push_back(p1);
          for (int i = 1; i < (int)a.size(); i++) {
                  if (i == a.size() - 1 || cw(p1, a[i],
                        p2, include_collinear)) {
                          while (up.size() >= 2 &&
                                !cw(up[up.size() - 2],
                                up[up.size() - 1], a[i],
                                include_collinear))
                                  up.pop_back();
                          up.push_back(a[i]);
                  }
                  if (i == a.size() - 1 || ccw(p1, a[i],
                        p2, include_collinear)) {
                          while (down.size() >= 2 &&
                                !ccw(down[down.size() - 2],
                                down[down.size() - 1], a[i],
                                include_collinear))
                                  down.pop_back();
                          down.push_back(a[i]);
                  }
          }
          if (include_collinear && up.size() == a.size())
                {
                  reverse(a.begin(), a.end());
                  return;
          }
          a.clear();
          for (int i = 0; i < (int)up.size(); i++)
                  a.push_back(up[i]);
          for (int i = down.size() - 2; i > 0; i--)
                  a.push_back(down[i]);
}
```

5.3 Polygons

5.3.1 Area

```
// Clockwise: negative
// Counter-clockwise: positive

double triangle_signed_area(complex<double> a,
      complex<double> b, complex<double> c) {
          // return cross(b - a, c - b) / 2.0;
          return (conj(b - a) * (c - b)).imag() / 2.0;
}

// Polygon is sum of signed triangles
// => sum of: (for each edge) triangle(origin,
      edge.from, edge.to)


// BETTER APPROACH
// NOT SELF INTERSECTING
// for each edge: calculate the area between y=0 and
      the edge
//                          add the sign according
      to the orientation
//                          sum up all the areas
//
```

5.3.2 Minkowski sum of convex polygons

```
/*
        Description: -
        Time: O(|P| + |Q|)
        Space: O(|P| + |Q|)
*/
```

```
// TODO

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;
}
```

5.4 Sweeping Line

```
// TODO

// /*
//     Description: -
//     TODO Time: O(N)
//     TODO Space: O(N)
// */
//
// 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);
// }
```

5.5 Triangles

Let a, b, c be length of the three sides of a triangle.

$$p = (a + b + c) * 0.5$$

The inradius is defined by:

$$iR = \sqrt{\frac{(p-a)(p-b)(p-c)}{p}}$$

The radius of its circumcircle is given by the formula:

$$cR = \frac{abc}{\sqrt{(a+b+c)(a+b-c)(a+c-b)(b+c-a)}}$$

5.6 misc

5.6.1 Half-plane intersection

```
/*
      Description: -
      TODO Time: O(?)
      TODO Space: O(?)
*/

// TODO: convert to complex

// Redefine epsilon and infinity as necessary. Be
    mindful of precision errors.
const long double eps = 1e-9, inf = 1e9;

// Basic point/vector struct.
struct Point {

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

      // Addition, subtraction, multiply by constant,
          dot product, cross product.

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

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

      friend Point operator * (const Point& p, const
          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++) {

            // 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;
}
```

5.6.2 Nearest pair of points

```
/*
    Description: -
    TODO Time: O(?)
    TODO Space: O(?)
*/


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];
        }
    }
}

t.resize(n);
sort(a.begin(), a.end(), cmp_x());
mindist = 1E20;
rec(0, n);

// Generalization: finding a triangle with minimal
     perimeter
// TODO
```

6 Graph

6.1 Bellmanford

```
/*
	Description: -
	Time: O(V * E)
	Space: O(V)
*/

vector<int> bellmanford(vector<tuple<int, int, int>>
     edges, int start, int n) {

	vector<int> ans(n, INT32_MAX);
	ans[start] = 0;
	bool change = true;

	for (int i = 0; i < n && change; i++) {
		change = false;
		for (auto e : edges) {
			if (ans[get<0>(e)] + get<2>(e) <
			     ans[get<1>(e)]) {
				ans[get<1>(e)] =
				     ans[get<0>(e)] +
				     get<2>(e);
				change = true;
			}
		}
	}

	if (change)
		ans[start] = -1;

	return ans;
}
```

6.2 Blossem

```
#include <iostream>
#include <vector>
#include <cstring>
using namespace std;

const int N = 500; // Max number of vertices

vector<int> graph[N]; // Adjacency list representation
int match[N]; // Stores the matching
bool vis[N]; // Visited array

// Find an augmenting path
bool dfs(int u) {
	vis[u] = true;
	for (int v : graph[u]) {
		if (!vis[v]) {
			vis[v] = true;
			if (match[v] == -1 ||
			     dfs(match[v])) {
				match[u] = v;
				match[v] = u;
				return true;
			}
		}
	}
	return false;
}

// Blossom algorithm
int blossom(int n) {
	memset(match, -1, sizeof(match));
	int ans = 0;
	for (int i = 0; i < n; ++i) {
		if (match[i] == -1) {
			memset(vis, false, sizeof(vis));
			if (dfs(i)) {
				++ans;
			}
		}
	}
	return ans;
}

int main() {
	// Example usage
	int n, m; // Number of vertices and edges
	cin >> n >> m;
	for (int i = 0; i < m; ++i) {
		int u, v; // Edge (u, v)
		cin >> u >> v;
		graph[u].push_back(v);
		graph[v].push_back(u);
	}
	int maxMatching = blossom(n);
	cout << "Maximum matching size: " <<
	     maxMatching << endl;
	return 0;
}
```

6.3 DFS - Iterative

```
/*
	Description: -
	Time: O(V + E)
	Space: O(V)
*/

ll n; // Number of nodes
vector<vector<ll>> adj; // Adjacency list of graph
vector<bool> visited;

void dfs() {
	ll start = 0;

	stack<ll> st;
	st.push(start);
	visited[node] = true;

	while (!st.empty()) {
		ll top = st.top();
		// Code ...
	}
}
```

6.4 DFS - Recursive

```
/*
	Description: -
	Time: O(V + E)
	Space: O(V)
*/

ll n; // number of nodes
vector<vector<ll>> adj; // adjacency list of graph
vector<bool> visited;

void dfs(ll node) {
	visited[node] = true;

	// Code ...

	for (auto x : adj[node]) {
		if (!visited[x]) {
			dfs(x);
		}
	}
}
```

6.5 Finding Bridges

```
/*
	Description: -
	Time: O(V + E)
	Space: O(V)
*/
```

```
ll n; // number of nodes
vector<vector<ll>> adj; // adjacency list of graph

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

void dfs(ll v, ll p = -1) {
	visited[v] = true;
	tin[v] = low[v] = timer++;
	for (ll to : adj[v]) {
		if (to == p) continue;

		if (visited[to]) {
			low[v] = min(low[v], tin[to]);
		} else {
			dfs(to, v);
			low[v] = min(low[v], low[to]);
			if (low[to] > tin[v]) {
				// The edge (v, to) is a
				    bridge
			}
		}
	}
}

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

6.6 Heavy Light Decomposition

```
// Decompose graph to a set of discount paths

vector<vi> AL;
vi par, heavy;// undirected tree
int heavy_light(int x) {
	int size = 1;
	int max_child_size = 0;
	for (auto& y : AL[x]) {
		if (y == par[x]) continue;
		par[y] = x;
		int child_size = heavy_light(y);
		if (child_size > max_child_size) {
			max_child_size = child_size;
			heavy[x] = y;
		}
		size += child_size;
	}
	return size;
}

vi group;
void decompose(int x, int p) {
	group[x] = p;
	for (auto& y : AL[x]) {
		if (y == par[x]) continue;
		if (y == heavy[x])
			decompose(y, p);
		else
			decompose(y, y);
	}
}
```

6.7 Karp

```
/*
	Description: Find minimum average weight of a
	    cycle in connected and directed graph.
	Time: O(V^3)
	Space: O(V^2)
*/

const int V = 4;

// a struct to represent edges
struct edge {
	int from, weight;
};

// vector to store edges
vector <edge> edges[V];

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

// calculates the shortest path
void shortest_path(int dp[][V]) {
	// initializing all distances as -1
	for (int i = 0; i <= V; i++)
		for (int j = 0; j < V; j++)
			dp[i][j] = -1;

	// Shortest distance from first vertex to in
	    itself consisting of 0 edges
	dp[0][0] = 0;

	// filling up the dp table
	for (int i = 1; i <= V; i++) {
		for (int j = 0; j < V; j++) {
			for (int k = 0; k <
			    edges[j].size(); k++) {
				if (dp[i -
				    1][edges[j][k].from]
				    != -1) {
					int curr_wt = dp[i
					    -
					    1][edges[j][k].from]
					    +
					    edges[j][k].weight;

					if (dp[i][j] == -1)
						dp[i][j] =
						    curr_wt;
					else
						dp[i][j] =
						    min(dp[i][j],
						    curr_wt);
				}
			}
		}
	}
}

// Returns minimum value of average weight of a cycle
    in graph.
double min_avg_weight()
{
	int dp[V + 1][V];
	shortest_path(dp);

	// array to store the avg values
	double avg[V];
	for (int i = 0; i < V; i++)
		avg[i] = -1;

	// Compute average values for all vertices
	    using weights of shortest paths store in dp.
	for (int i = 0; i < V; i++) {
		if (dp[V][i] != -1) {
			for (int j = 0; j < V; j++)
				if (dp[j][i] != -1)
					avg[i] = max(avg[i],
					    ((double)dp[V][i]
					    -
					    dp[j][i])
```

```
                                        / (V -
                                        j));
                }
        }

        // Find minimum value in avg[]
        double result = avg[0];
        for (int i = 0; i < V; i++)
                if (avg[i] != -1 && avg[i] < result)
                        result = avg[i];

        return result;
}

// Driver
int main() {
        add_edge(0, 1, 1);
        add_edge(0, 2, 10);
        add_edge(1, 2, 3);

        cout << min_avg_weight();
}
```

6.8 LCA

```
/*
        Description: -
        Time: Processing: O(N * log(N))
                  Query: O(N)
        Space: O(N + E)
*/

// read this
    https://www.topcoder.com/thrive/articles/Range%20Minimum%20Query%20and%20Lowest%20Common%20Ancestor
// please read it ffs. <O(N), O(1)> ALGORITHM FOR THE
    RESTRICTED RMQ and how to be used in LCA

int n, l;
vector<vector<int>> adj;

int timer;
vector<int> tin, tout;
vector<vector<int>> up;

void dfs(int v, int p)
{
        tin[v] = ++timer;
        up[v][0] = p;
        for (int i = 1; i <= l; ++i)
                up[v][i] = up[up[v][i - 1]][i - 1];
        for (int u : adj[v]) {
                if (u != p)
                        dfs(u, v);
        }

        tout[v] = ++timer;
}

bool is_ancestor(int u, int v)
{
        return tin[u] <= tin[v] && tout[u] >= tout[v];
}

int lca(int u, int v)
{
        if (is_ancestor(u, v))
                return u;
        if (is_ancestor(v, u))
                return v;
        for (int i = l; i >= 0; --i) {
                if (!is_ancestor(up[u][i], v))
                        u = up[u][i];
        }
        return up[u][0];
}

void preprocess(int root) {
        tin.resize(n);
        tout.resize(n);
        timer = 0;
        l = ceil(log2(n));
        up.assign(n, vector<int>(l + 1));
        dfs(root, root);
}
```

6.9 Max Flows - Dinic

```
/*
        Description: -
        Time: O(E * sqrt(E))
        Space: O(V + E)
*/

struct FlowEdge {
        int v, u;
        long long cap, flow = 0;
        FlowEdge(int v, int u, long long cap) : v(v),
            u(u), cap(cap) {}
};

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

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

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

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

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

```
                    continue;
                long long tr = dfs(u, min(pushed,
                    edges[id].cap -
                    edges[id].flow));
                if (tr == 0)
                    continue;
                edges[id].flow += tr;
                edges[id ^ 1].flow -= tr;
                return tr;
            }
            return 0;
        }

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

6.10 Planar Graph

Euler's formula states that if a finite, connected, planar graph is drawn in the plane without any edge intersections, and v is the number of vertices, e is the number of edges and f is the number of faces (regions bounded by edges, including the outer, infinitely large region), then:

$$f + v = e + 2$$

It can be extended to non connected planar graphs with c connected components:

$$f + v = e + c + 1$$

6.11 Strongly Connected Components - Kosaraju

```
/*
        Description: -
        Time: O(V + E)
        Space: O(V + E)
*/

// pouya cp-algo kosaraju
vector<vector<int>> adj, adj_rev;
vector<bool> used;
vector<int> order, component;

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

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

        order.push_back(v);
}

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

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

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

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

        used.assign(n, false);

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

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

        for (auto v : order)
                if (!used[v]) {
                        dfs2(v);

                        // ... processing next component
                            ...

                        component.clear();
                }
}
```

6.12 Topological Sort

```
// Class to represent a graph
class Graph {
        int V; // No. of vertices
        list<int>* adj; // Pointer to an array
            containing
        // adjacency lists

public:
        Graph(int V); // Constructor
        void addEdge(int v,
                int w); // Function to add an edge to
                    graph
        void topologicalSort(); // prints a Topological
            Sort of
        // the complete graph
};

Graph::Graph(int V)
{
        this->V = V;
        adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
        adj[v].push_back(w); // Add w to vs list.
}

// Function to perform Topological Sort
void Graph::topologicalSort()
{
        // Create a vector to store in-degree of all
            vertices
        vector<int> in_degree(V, 0);

        // Traverse adjacency lists to fill in_degree of
        // vertices
```

```
for (int v = 0; v < V; ++v) {
        for (auto const& w : adj[v])
                in_degree[w]++;
}

// Create a queue and enqueue all vertices with
// in-degree 0
queue<int> q;
for (int i = 0; i < V; ++i) {
        if (in_degree[i] == 0)
                q.push(i);
}

// Initialize count of visited vertices
int count = 0;

// Create a vector to store topological order
vector<int> top_order;

// One by one dequeue vertices from queue and
     enqueue
// adjacent vertices if in-degree of adjacent
     becomes 0
while (!q.empty()) {
        // Extract front of queue (or perform
             dequeue)
        // and add it to topological order
        int u = q.front();
        q.pop();
        top_order.push_back(u);

        // Iterate through all its neighbouring
             nodes
        // of dequeued node u and decrease their
             in-degree
        // by 1
        list<int>::iterator itr;
        for (itr = adj[u].begin(); itr !=
             adj[u].end();
                ++itr)
                // If in-degree becomes zero, add
                     it to queue
                if (--in_degree[*itr] == 0)
                        q.push(*itr);

        count++;
}

// Check if there was a cycle
if (count != V) {
        cout << "Graph contains cycle\n";
        return;
}

        // Print topological order
        for (int i : top_order)
                cout << i << " ";
}

// Driver code
int main()
{
        // Create a graph given in the above diagram
        Graph g(6);
        g.addEdge(5, 2);
        g.addEdge(5, 0);
        g.addEdge(4, 0);
        g.addEdge(4, 1);
        g.addEdge(2, 3);
        g.addEdge(3, 1);

        cout << "Following is a Topological Sort of the
             given "
                "graph\n";
        g.topologicalSort();

        return 0;
}
```

7 Math

7.1 FFT

```
/**
 * Fast Fourier Transform.
 * Useful to compute convolutions.
 * computes:
 *    C(f star g)[n] = sum_m(f[m] * g[n - m])
 * for all n.
 * test: icpc live archive, 6886 - Golf Bot
 * */

 // TODO

using namespace std;
#include<bits/stdc++.h>
#define D(x) cout << #x " = " << (x) << endl
#define endl '\n'

const int MN = 262144 << 1;
int d[MN + 10], d2[MN + 10];
const double PI = acos(-1.0);

int rev(int id, int len) {
        int ret = 0;
        for (int i = 0; (1 << i) < len; i++) {
                ret <<= 1;
                if (id & (1 << i)) ret |= 1;
        }
        return ret;
}

complex<double> A[1 << 20];

void FFT(complex<double>* a, int len, int DFT) {
        for (int i = 0; i < len; i++)
                A[rev(i, len)] = a[i];

        for (int s = 1; (1 << s) <= len; s++) {
                int m = (1 << s);
                complex<double> wm =
                    complex<double>(cos(DFT * 2 * PI /
                    m), sin(DFT * 2 * PI / m));

                for (int k = 0; k < len; k += m) {
                        complex<double> w =
                            complex<double>(1, 0);
                        for (int j = 0; j < (m >> 1);
                            j++) {
                            complex<double> t = w *
                                A[k + j + (m >> 1)];
                            complex<double> u = A[k +
                                j];
                            A[k + j] = u + t;
                            A[k + j + (m >> 1)] = u -
                                t;
                            w = w * wm;
                        }
                }
        }
        if (DFT == -1) for (int i = 0; i < len; i++)
              A[i].real() /= len, A[i].imag() /= len;
        for (int i = 0; i < len; i++) a[i] = A[i];
        return;
}

complex<double> in[1 << 20];

void solve(int n) {
        memset(d, 0, sizeof d);
        int t;
        for (int i = 0; i < n; ++i) {
                cin >> t;
                d[t] = true;
        }
        int m;
```

```
    cin >> m;
    vector<int> q(m);
    for (int i = 0; i < m; ++i)
        cin >> q[i];

    for (int i = 0; i < MN; ++i) {
        if (d[i])
            in[i] = complex<double>(1, 0);
        else
            in[i] = complex<double>(0, 0);
    }

    FFT(in, MN, 1);
    for (int i = 0; i < MN; ++i) {
        in[i] = in[i] * in[i];
    }
    FFT(in, MN, -1);

    int ans = 0;
    for (int i = 0; i < q.size(); ++i) {
        if (in[q[i]].real() > 0.5 || d[q[i]]) {
            ans++;
        }
    }
    cout << ans << endl;
}

int main() {
    ios_base::sync_with_stdio(false);cin.tie(NULL);
    int n;
    while (cin >> n)
        solve(n);
    return 0;
}
```

7.2 Fibonacci Properties

Let A, B and n be integer numbers.

$$k = A - B \tag{1}$$

$$F_A F_B = F_{k+1} F_A^2 + F_k F_A F_{A-1} \tag{2}$$

$$\sum_{i=0}^{n} F_i^2 = F_{n+1} F_n \tag{3}$$

$ev(n)$ = returns 1 if n is even.

$$\sum_{i=0}^{n} F_i F_{i+1} = F_{n+1}^2 - ev(n) \tag{4}$$

$$\sum_{i=0}^{n} F_i F_{i-1} = \sum_{i=0}^{n-1} F_i F_{i+1} \tag{5}$$

7.3 IFFT

```
void IFFT(vector<cd>& A) {
    for (auto& p : A) p = conj(p);
    // complex conjugate
    // a + bi -> a - bi
    FFT(A);
    for (auto& p : A) p = conj(p);// complex
        conjugate
    // **not needed for our purpose**
    for (auto& p : A) p /= A.size();// scale down
        (1/n)
}
```

7.4 Lucas Theorem

For non-negative integers m and n and a prime p, the following congruence relation holds: :

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

where :

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

and :

$$n = n_k p^k + n_{k-1} p^{k-1} + \cdots + n_1 p + n_0$$

are the base p expansions of m and n respectively. This uses the convention that $\binom{m}{n} = 0$ if $m \leq n$.

8 Misc

8.1 Fraction

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

    fraction(long long a, long long b) {
        long long g = gcd(a, b);
        x = a / g;
        y = b / g;
    }

    bool operator < (const fraction& o) const {
        return (x * o.y < y * o.x);
    }
};
```

8.2 IO

```
// Read while input is available
while (getline(cin, s)) {
    // Work with s
}

// Set number of precision digits for floating point
   variables
// Place it at the start of the code
cout << fixed << setprecision(5);

// Print with fixed precision points
printf("%.5f", x);
```

8.3 Matrix

```
const int MN = 111;
const int mod = 10000;

struct matrix {
    int r, c;
    int m[MN][MN];

    matrix(int _r, int _c) : r(_r), c(_c) {
        memset(m, 0, sizeof m);
    }

    void print() {
        for (int i = 0; i < r; ++i) {
            for (int j = 0; j < c; ++j)
                cout << m[i][j] << " ";
            cout << endl;
```

```
        }
    }

    int x[MN][MN];
    matrix& operator *= (const matrix& o) {
        memset(x, 0, sizeof x);
        for (int i = 0; i < r; ++i)
            for (int k = 0; k < c; ++k)
                if (m[i][k] != 0)
                    for (int j = 0; j <
                        c; ++j) {
                        x[i][j] =
                            (x[i][j]
                            +
                            ((m[i][k]
                            *
                            o.m[k][j])
                            % mod))
                            % mod;
                    }
        memcpy(m, x, sizeof(m));
        return *this;
    }
};

void matrix_pow(matrix b, long long e, matrix& res) {
    memset(res.m, 0, sizeof res.m);
    for (int i = 0; i < b.r; ++i)
        res.m[i][i] = 1;

    if (e == 0) return;
    while (true) {
        if (e & 1) res *= b;
        if ((e >>= 1) == 0) break;
        b *= b;
    }
}
```

8.4 Template

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

#define ll long long
#define pll pair<ll, ll>
#define vll vector<ll>
#define sll set<ll>
#define MOD 1000000007
#define Deb(x) cout << #x " : "<<x<<endl;
using u64 = uint64_t;
using u128 = __uint128_t;

int main() {
    ios::sync_with_stdio(false); cin.tie(0);
        cout.tie(0);
    ll n, m, q;
    string s;

    // Code ...
}
```

9 String Processing

9.1 Builtin Hash

```
/*
    Description: -
    Time: O(N)
    Space: O(1)
*/

hash<string> hash_func;

hash_func("123");
```

9.2 Palindromes

```
/*
    Description: -
    Time: O(N)
    Space: O(N)
*/

vector<int> odd_palindromes(vector<int> s) {
    int n = s.size();
    vector<int> oddP(n, 0);
    int L = -1, R = -1;
    for (int i = 0; i < n; i++)
    {
        if (i < R) {
            oddP[i] = min(R - i, oddP[L + R -
                i]);
        }
        int left = i - oddP[i] - 1, right = i +
            oddP[i] + 1;
        while (left >= 0 && right < n && s[left]
            == s[right]) {
            oddP[i]++;
            right++;
            left--;
        }
        if (i + oddP[i] > R)
        {
            L = i - oddP[i];R = i + oddP[i];
        }
    }
    return oddP;
}

vector<int> even_palindromes(vector<int> s) {
    int n = s.size();
    vector<int> evenP(n - 1, 0);
    int L = -1, R = -1;
    for (int i = 0; i < n - 1; i++)
    {
        if (i < R) {
            evenP[i] = min(R - i, evenP[L + R
                - i - 1]);
        }
        int left = i - evenP[i], right = i +
            evenP[i] + 1;
        while (left >= 0 && right < n && s[left]
            == s[right]) {
            evenP[i]++;
            right++;
            left--;
        }
        if (i + evenP[i] > R)
        {
            L = i - evenP[i] + 1;R = i +
                evenP[i];
        }
    }
    return evenP;
}
```

9.3 Perfect Prefix

```
/*
    Description: Perfect prefix pi function
    Time: O(N * log(N))
    Space: O(N)
*/

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;
}

// count the number of each prefix
vector<int> ans(n + 1);
for (int i = 0; i < n; i++)
        ans[pi[i]]++;
for (int i = n - 1; i > 0; i--)
ans[pi[i - 1]] += ans[i];
for (int i = 0; i <= n; i++)
ans[i]++;
```

9.4 Rabin Karp

```
/*
        Description: -
        Time: O(max(N, M))
        Space: O(max(N, M))
*/

vector<int> rabin_karp(string const& s, string const&
     t) {
        const int p = 31;
        const int m = 1e9 + 9;
        int S = s.size(), T = t.size();

        vector<long long> p_pow(max(S, T));
        p_pow[0] = 1;
        for (int i = 1; i < (int)p_pow.size(); i++)
                p_pow[i] = (p_pow[i - 1] * p) % m;

        vector<long long> h(T + 1, 0);
        for (int i = 0; i < T; i++)
                h[i + 1] = (h[i] + (t[i] - 'a' + 1) *
                     p_pow[i]) % m;
        long long h_s = 0;
        for (int i = 0; i < S; i++)
                h_s = (h_s + (s[i] - 'a' + 1) *
                     p_pow[i]) % m;

        vector<int> occurrences;
        for (int i = 0; i + S - 1 < T; i++) {
                long long cur_h = (h[i + S] + m - h[i])
                     % m;
                if (cur_h == h_s * p_pow[i] % m)
                        occurrences.push_back(i);
        }
        return occurrences;
}
```

9.5 Rolling Hash

```
/*
        Description: -
        Time: O(N)
        Space: O(N)
*/

vector<ll> rolling_hash(vector<int> s, ll p = 53) {
        int n = s.size();
        vector<ll> rh(n + 1, 0);
        ll p_pow = 1;
        for (int i = 0; i < n; i++)
        {
                rh[i + 1] = (rh[i] + (s[i] * p_pow) %
                     MOD) % MOD;
                p_pow = (p_pow * p) % MOD;
        }
        return rh;
}

// Description: -
// Time: O(log(s2 - s1))
// Space: O(1)
bool compare(vector<ll> rh, int s1, int e1, int s2, int
     e2, int p = 53) {
        if (s2 < s1) {
                swap(s1, s2);
                swap(e1, e2);
        }
        ll h1 = rh[e1 + 1] - rh[s1], h2 = rh[e2 + 1] -
             rh[s2];

        return(h1 * binpow(p, s2 - s1, MOD)) % MOD ==
             h2;
}
```

9.6 Suffix Array

```
/*
        Description: -
        Time: O(N * log(N))
        Space: O(N)
*/

vector<int> buildSuffixArray(vector<int>& s) {
        int n = s.size();
        vector<int> suffixArray(n), rank(n), temp(n);

        for (int i = 0; i < n; i++) {
                suffixArray[i] = i;
                rank[i] = s[i];
        }

        for (int k = 1; k < n; k *= 2) {
                auto cmp = [&](int i, int j) {
                        if (rank[i] != rank[j]) return
                             rank[i] < rank[j];
                        int ri = (i + k < n) ? rank[i +
                             k] : -1;
                        int rj = (j + k < n) ? rank[j +
                             k] : -1;
                        return ri < rj;
                        };
                sort(suffixArray.begin(),
                     suffixArray.end(), cmp);

                temp[suffixArray[0]] = 0;
                for (int i = 1; i < n; i++) {
                        temp[suffixArray[i]] =
                             temp[suffixArray[i - 1]];
                        if (cmp(suffixArray[i - 1],
                             suffixArray[i])) {
                                temp[suffixArray[i]]++;
                        }
                }
                rank = temp;
        }

        return suffixArray;
}

// function to build LCP array
vector<int> buildLCPArray(vector<int> s, const
     vector<int>& suffixArray) {
        int n = s.size();
        vector<int> lcp(n, 0), rank(n, 0);

        for (int i = 0; i < n; i++) {
                rank[suffixArray[i]] = i;
        }

        int h = 0;
        for (int i = 0; i < n; i++) {
                if (rank[i] > 0) {
                        int j = suffixArray[rank[i] - 1];
```

```
                        while (i + h < n && j + h < n &&
                             s[i + h] == s[j + h]) {
                                h++;
                        }
                        lcp[rank[i]] = h;
                        if (h > 0) h--;
                }
        }

        return lcp;
}

bool comp(const string& s, int i, const string& t, int
     k) {
        while (k + i < s.size() && k < t.size() && s[i
             + k] == t[k]) k++;
        if (k + i == s.size())return true;
        if (k == t.size())return false;
        return s[i + k] < t[k];

}

bool queryExists(const string& s, const string& subs,
     const vector<int>& suffixArray,
     vector<vector<int>>& lcpSparseTable) {
        int n = s.size(), l = 0, r = n - 1;
        while (l + 1 < r) {
                int g = (l + r) / 2;
                int k = querySparseTable(lcpSparseTable,
                     l, r, min);
                bool b = comp(s, suffixArray[g], subs,
                     k);

                if (b) l = g;
                else r = g;
        }
        int i = 0;
        while (suffixArray[l] + i < s.size() && i <
             subs.size() && s[suffixArray[l] + i] ==
             subs[i])i++;
        return i == subs.size();
}


// TODO
// between n strings s1,...,sn finds longest common
     between atleast k of them (undefined behavior k=1)
int longestCommonSubstring(const vector<string>
     strings, int k) {
        vector<int> s;

        int n = strings.size();
        for (int i = 0; i < strings.size();i++) {
                for (auto c : strings[i]) {
                        s.push_back(c);
                }
                s.push_back(-i - 1);
        }
        auto suffixArray = buildSuffixArray(s);
        auto LCPArray = buildLCPArray(s, suffixArray);
        auto LCPSparseTable =
             buildSparseTable(LCPArray, min);
        vector<int> type(s.size());
        int j = 0, cnt = strings[0].size() + 1;
        for (int i = 0; i < s.size(); i++)
        {
                if (i == cnt)
                {
                        j += 1;
                        cnt += strings[j].size() + 1;
                }
                type[i] = j;
        }
        int ans = 0;
        vector<int>freq(n, 0);
        int i = n, ii = n;
        cnt = 0;
        for (int i = n; i < s.size(); i++) {
                if (!freq[type[suffixArray[i]]])cnt++;
                freq[type[suffixArray[i]]]++;

                while (cnt >= k) {
                        ans = max(ans,
                             querySparseTable(LCPSparseTable,
                             ii + 1, i, min));
                        freq[type[suffixArray[ii]]]--;
                        if (!freq[type[suffixArray[ii]]])
                             cnt--;
                        ii++;
                }
        }
        return ans;
}

int countUniqueSubStrings(const vector<int>& lcpArray) {
        int n = lcpArray.size(), c = 0;
        for (int i = 0; i < n; i++) {
                c += lcpArray[i];
        }
        return n * (n + 1) / 2 - c;
}


// longest substring repeated in same string.
int longestRepeatedSubStrings(const vector<int>&
     lcpArray) {
        int n = lcpArray.size(), c = 0;
        for (int i = 0; i < n; i++) {
                c = max(c, lcpArray[i]);
        }
        return c;
}
```

9.7 Z-function

```
/*
        Description: Longest common prefix between s
             and s[i:]
        Time: O(N)
        Space: O(M)
*/

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 - 1]);
                }
                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;
}
```