Contents

Chapter 1

Trial of data structure

1.1 KD tree

```
/*  KD-tree :
        queries the k-th closest point in O (k * n ^ (1 - 1 / k)).
        Stores the data in p[].
        Call init (n, k).
        Call min_kth (d, k) / max_kth (d, k).
*/
template <int MAXN = 200000, int MAXK = 2>
struct kd_tree {
    int k, size;
    struct point {
        int data[MAXK], id;
    } p[MAXN];
    struct kd_node {
        int l, r;
        point p, dmin, dmax;
        kd_node() {}
        kd_node (const point &rhs) : l (-1), r (-1), p (rhs), dmin (rhs), dmax (rhs) {}
        void merge (const kd_node &rhs, int k) {
            for (register int i = 0; i < k; i++) {
                dmin.data[i] = std::min (dmin.data[i], rhs.dmin.data[i]);
                dmax.data[i] = std::max (dmax.data[i], rhs.dmax.data[i]);
            }
        }
        long long min_dist (const point &rhs, int k) const {
            register long long ret = 0;
            for (register int i = 0; i < k; i++) {
                if (dmin.data[i] <= rhs.data[i] && rhs.data[i] <= dmax.data[i]) continue;
                ret += std::min (1ll * (dmin.data[i] - rhs.data[i]) * (dmin.data[i] - rhs.data[i]),
                                 1ll * (dmax.data[i] - rhs.data[i]) * (dmax.data[i] - rhs.data[i]));
            }
            return ret;
        }
        long long max_dist (const point &rhs, int k) {
            long long ret = 0;
            for (int i = 0; i < k; i++) {
                int tmp = std::max (std::abs (dmin.data[i] - rhs.data[i]),
                                    std::abs (dmax.data[i] - rhs.data[i]));
                ret += 1ll * tmp * tmp;
            }
            return ret;
        }
    } tree[MAXN * 4];
    struct result {
        long long dist;
        point d;
        result() {}
        result (const long long &dist, const point &d) : dist (dist), d (d) {}
        bool operator > (const result &rhs) const {
            return dist > rhs.dist || (dist == rhs.dist && d.id > rhs.d.id);
        }
        bool operator < (const result &rhs) const {
            return dist < rhs.dist || (dist == rhs.dist && d.id < rhs.d.id);
        }
    };
    long long sqrdist (const point &a, const point &b) {
        long long ret = 0;
        for (int i = 0; i < k; i++)
            ret += 1ll * (a.data[i] - b.data[i]) * (a.data[i] - b.data[i]);
        return ret;
    }
    int alloc() {
        tree[size].l = tree[size].r = -1;
        return size++;
    }
    void build (const int &depth, int &rt, const int &l, const int &r) {
        if (l > r) return;
        register int middle = (l + r) >> 1;
        std::nth_element (p + l, p + middle, p + r + 1,
        [ = ] (const point & a, const point & b) {
            return a.data[depth] < b.data[depth];
        });
        tree[rt = alloc()] = kd_node (p[middle]);
        if (l == r) return;
        build ((depth + 1) % k, tree[rt].l, l, middle - 1);
        build ((depth + 1) % k, tree[rt].r, middle + 1, r);
```

```
            if (~tree[rt].l) tree[rt].merge (tree[tree[rt].l], k);
            if (~tree[rt].r) tree[rt].merge (tree[tree[rt].r], k);
        }
        std::priority_queue<result, std::vector<result>, std::less <result> > heap_l;
        std::priority_queue<result, std::vector<result>, std::greater <result> > heap_r;
        void _min_kth (const int &depth, const int &rt, const int &m, const point &d) {
            result tmp = result (sqrdist (tree[rt].p, d), tree[rt].p);
            if ((int)heap_l.size() < m)
                heap_l.push (tmp);
            else if (tmp < heap_l.top()) {
                heap_l.pop();
                heap_l.push (tmp);
            }
            int x = tree[rt].l, y = tree[rt].r;
            if (~x && ~y && sqrdist (d, tree[x].p) > sqrdist (d, tree[y].p)) std::swap (x, y);
            if (~x && ((int)heap_l.size() < m || tree[x].min_dist (d, k) < heap_l.top().dist))
                _min_kth ((depth + 1) % k, x, m, d);
            if (~y && ((int)heap_l.size() < m || tree[y].min_dist (d, k) < heap_l.top().dist))
                _min_kth ((depth + 1) % k, y, m, d);
        }
        void _max_kth (const int &depth, const int &rt, const int &m, const point &d) {
            result tmp = result (sqrdist (tree[rt].p, d), tree[rt].p);
            if ((int)heap_r.size() < m)
                heap_r.push (tmp);
            else if (tmp > heap_r.top()) {
                heap_r.pop();
                heap_r.push (tmp);
            }
            int x = tree[rt].l, y = tree[rt].r;
            if (~x && ~y && sqrdist (d, tree[x].p) < sqrdist (d, tree[y].p)) std::swap (x, y);
            if (~x && ((int)heap_r.size() < m || tree[x].max_dist (d, k) >= heap_r.top().dist))
                _max_kth ((depth + 1) % k, x, m, d);
            if (~y && ((int)heap_r.size() < m || tree[y].max_dist (d, k) >= heap_r.top().dist))
                _max_kth ((depth + 1) % k, y, m, d);
        }
        void init (int n, int k) {
            this -> k = k; size = 0;
            int rt = 0;
            build (0, rt, 0, n - 1);
        }
        point min_kth (const point &d, const int &m) {
            heap_l = decltype (heap_l) ();
            _min_kth (0, 0, m, d);
            return heap_l.top ().d;
        }
        point max_kth (const point &d, const int &m) {
            heap_r = decltype (heap_r) ();
            _max_kth (0, 0, m, d);
            return heap_r.top ().d;
        }
    };
```

1.2 Splay tree

```
/*  Splay Tree :
        Solver for sequence problems.
        Maintain msg and tag, and update accordingly.
        This sample is collected from BZOJ 1500.
*/
const int INF = 1E9;
template <int MAXN = 510000>
    struct splay_tree {
        //  TODO : Maintain messages here.
        struct msg {
            int size;
            int l_max;
            int r_max;
            int sum_max;
            int sum;
            explicit msg (int size = 0, int l_max = 0, int r_max = 0, int sum_max = 0, int sum = 0) :
                size (size), l_max (l_max), r_max (r_max), sum_max (sum_max), sum (sum) {}
        };
        //  TODO : Maintain tags here.
        struct tag {
            bool r;
            int mod;
            explicit tag (bool r = false, int mod = INF) : r (r), mod (mod) {}
        };
        struct node {
            int c[2], f;
            msg m;
            tag t;
            node () {
                c[0] = c[1] = f = -1;
                m = msg ();
                t = tag ();
            }
            //  TODO : Maintain node values here.
            int val;
        } n[MAXN];
        int root;
        //  TODO : msg & tag operations.
        msg merge (const msg &a, const msg &b) {
            return msg (a.size + b.size,
                    std::max (a.l_max, a.sum + b.l_max),
                    std::max (b.r_max, a.r_max + b.sum),
```

```
                std::max (std::max (a.sum_max, b.sum_max), a.r_max + b.l_max),
                a.sum + b.sum);
    }
    msg gen (int a) {
        return msg (1, n[a].val, n[a].val, n[a].val, n[a].val);
    }
    tag merge (const tag &a, const tag &b) {
        if (b.mod != INF) return tag (a.r ^ b.r, b.mod);
        return tag (a.r ^ b.r, a.mod);
    }
    void push (int x, const tag &t) {
        if (t.mod != INF) {
            n[x].val = t.mod;
            n[x].m.l_max = (t.mod >= 0 ? t.mod * n[x].m.size : t.mod);
            n[x].m.r_max = (t.mod >= 0 ? t.mod * n[x].m.size : t.mod);
            n[x].m.sum_max = (t.mod >= 0 ? t.mod * n[x].m.size : t.mod);
            n[x].m.sum = t.mod * n[x].m.size;
        }
        if (t.r) {
            std::swap (n[x].c[0], n[x].c[1]);
            std::swap (n[x].m.l_max, n[x].m.r_max);
        }
        n[x].t = merge (n[x].t, t);
    }
    //  Splay tree operations.
    void push_down (int x) {
        if (~n[x].c[0]) push (n[x].c[0], n[x].t);
        if (~n[x].c[1]) push (n[x].c[1], n[x].t);
        n[x].t = tag ();
    }
    void update (int x) {
        n[x].m = gen (x);
        if (~n[x].c[0]) n[x].m = merge (n[n[x].c[0]].m, n[x].m);
        if (~n[x].c[1]) n[x].m = merge (n[x].m, n[n[x].c[1]].m);
    }
    void rotate (int x, int k) {
        push_down (x); push_down (n[x].c[k]);
        int y = n[x].c[k]; n[x].c[k] = n[y].c[k ^ 1]; n[y].c[k ^ 1] = x;
        if (n[x].f != -1) n[n[x].f].c[n[n[x].f].c[1] == x] = y;
        n[y].f = n[x].f; n[x].f = y; if (~n[x].c[k]) n[n[x].c[k]].f = x;
        update (x); update (y);
    }
    void splay (int x, int s = -1) {
        push_down (x);
        while (n[x].f != s) {
            if (n[n[x].f].f != s) rotate (n[n[x].f].f, n[n[n[x].f].f].c[1] == n[x].f);
            rotate (n[x].f, n[n[x].f].c[1] == x);
        }
        update (x);
        if (s == -1) root = x;
    }
    //  Alloc & free.
    int buf[MAXN], size;
    splay_tree () {
        root = -1; size = MAXN;
        for (int i = 0; i < MAXN; ++i) buf[i] = i;
    }
    int alloc () {
        while (size == 0);
        n[buf[--size]] = node ();
        return buf[size];
    }
    void del (int x) {
        buf[size++] = x;
    }
    void del_tree (int x) {
        if (!~x) return;
        del (x);
        del_tree (n[x].c[0]);
        del_tree (n[x].c[1]);
    }
    //  TODO : Put your own operations here. REMEMBER TO PUSH_DOWN!!
};
```

1.3 Link-cut tree

```
/*  Link-cut Tree :
        Dynamic tree that supports path operations.
        Usage :
            Maintain query values in msg.
            Maintain modifications in tag.
            Edit merge (), gen () and push () accordingly.
*/
template <int MAXN = 100000>
struct lct {
    struct msg {
        int size;
        explicit msg (int size = 0) : size (size) {}
    };
    struct tag {
        int r;
        explicit tag (int r = 0) : r (r) {}
    };
    struct node {
        int c[2];
        int f, p;
        msg m;
        tag t;
        node () {
```

```
            c[0] = c[1] = f = p = -1;
            m = msg ();
            t = tag ();
        }
    } n[MAXN];
    msg merge (const msg &a, const msg &b) {    // Merge two messages.
        return msg (a.size + b.size);
    }
    msg gen (int a) {
        return msg (1);
    }
    tag merge (const tag &a, const tag &b) {    // Merge two tags.
        return tag (a.r ^ b.r);
    }
    void push (int x, const tag &t) {
        if (t.r) std::swap (n[x].c[0], n[x].c[1]);
        n[x].t = merge (n[x].t, t); // Remember to update messages manually.
    }
    void update (int x) {
        n[x].m = gen (x);
        if (~n[x].c[0]) n[x].m = merge (n[n[x].c[0]].m, n[x].m);
        if (~n[x].c[1]) n[x].m = merge (n[x].m, n[n[x].c[1]].m);
    }
    void push_down (int x) {
        if (~n[x].c[0]) push (n[x].c[0], n[x].t);
        if (~n[x].c[1]) push (n[x].c[1], n[x].t);
        n[x].t = tag ();
    }
    void rotate (int x, int k) {
        push_down (x); push_down (n[x].c[k]);
        int y = n[x].c[k]; n[x].c[k] = n[y].c[k ^ 1]; n[y].c[k ^ 1] = x;
        if (n[x].f != -1) n[n[x].f].c[n[n[x].f].c[1] == x] = y;
        n[y].f = n[x].f; n[x].f = y; if (~n[x].c[k]) n[n[x].c[k]].f = x;
        std::swap (n[x].p, n[y].p);
        update (x); update (y);
    }
    void splay (int x, int s = -1) {
        push_down (x);
        while (n[x].f != s) {
            if (n[n[x].f].f != s) rotate (n[n[x].f].f, n[n[n[x].f].f].c[1] == n[x].f);
            rotate (n[x].f, n[n[x].f].c[1] == x);
        }
        update (x);
    }
    void access (int x) {
        int u = x, v = -1;
        while (u != -1) {
            splay (u); push_down (u);
            if (~n[u].c[1]) n[n[u].c[1]].f = -1, n[n[u].c[1]].p = u;
            n[u].c[1] = v;
            if (~v) n[v].f = u, n[v].p = -1;
            update (u); u = n[v = u].p;
        }
        splay (x);
    }
    void setroot (int x) {
        access (x);
        push (x, tag (1));
    }
    void link (int x, int y) {
        setroot (x);
        n[x].p = y;
    }
    void cut (int x, int y) {
        access (x); splay (y, -1);
        if (n[y].p == x) n[y].p = -1;
        else {
            access (y); splay (x, -1);
            n[x].p = -1;
        }
    }
    void directed_link (int x, int y) {
        access (x);
        n[x].p = y;
    }
    void directed_cut (int x) {
        access (x);
        if (~n[x].c[0]) n[n[x].c[0]].f = -1;
        n[x].c[0] = -1;
        update (x);
    }
};
```

Chapter 2

Trial of formula

2.1 Zeller's congruence

```
/*  Zeller's congruence :
        converts between a specific date and its index.
        (y must be greater than or equal to 0 in all cases.)
        (0 = Monday, 1 = Tuesday, ..., 6 = Sunday) (mod 7)
*/
int get_id (int y, int m, int d) {
    if (m < 3) { --y; m += 12; }
    return 365 * y + y / 4 - y / 100 + y / 400 + (153 * (m - 3) + 2) / 5 + d - 307;
}
std::tuple <int, int, int> date (int id) {
    int x = id + 1789995, n, i, j, y, m, d;
    n = 4 * x / 146097; x -= (146097 * n + 3) / 4;
    i = (4000 * (x + 1)) / 1461001; x -= 1461 * i / 4 - 31;
    j = 80 * x / 2447; d = x - 2447 * j / 80;
    x = j / 11;
    m = j + 2 - 12 * x; y = 100 * (n - 49) + i + x;
    return std::make_tuple (y, m, d);
}
```

2.2 Lattice points below segment

```
/*  Lattice points below segment :
        solves for sigma [(a + b * i) / m] where 0 <= i < n.
*/
long long solve(long long n, long long a, long long b, long long m){
    if (b == 0) return n * (a / m);
    if (a >= m) return n * (a / m) + solve (n, a % m, b, m);
    if (b >= m) return (n - 1) * n / 2 * (b / m) + solve (n, a, b % m, m);
    return solve ((a + b * n) / m, (a + b * n) % m, m, b);
}
```

Chapter 3

Trial of number theory

3.1 Constants and basic functions

```
/*  Basic constants & functions :
        long long inverse (const long long &x, const long long &mod) :
            returns the inverse of x modulo mod.
            i.e. x * inv (x) % mod = 1.
        int fpm (int x, int n, int mod) :
            returns x^n % mod. i.e. Fast Power with Modulo.
        void euclid (const long long &a, const long long &b,
                     long long &x, long long &y) :
            solves for ax + by = gcd (a, b).
        long long gcd (const long long &a, const long long &b) :
            solves for the greatest common divisor of a and b.
        long long mul_mod (const long long &a, const long long &b, const long long &mod) :
            returns a * b % mod.
        long long llfpm (const long long &x, const long long &n, const long long &mod) :
            returns x^n % mod.
*/
const double PI = acos (-1.);
long long abs (const long long &x) { return x > 0 ? x : -x; }
int fpm (int x, int n, int mod) {
    register int ans = 1, mul = x;
    while (n) {
        if (n & 1) ans = int (1ll * ans * mul % mod);
        mul = int (1ll * mul * mul % mod);
        n >>= 1;
    }
    return ans;
}
void euclid (const long long &a, const long long &b,
             long long &x, long long &y) {
    if (b == 0) x = 1, y = 0;
    else euclid (b, a % b, y, x), y -= a / b * x;
}
long long inverse (long long x, long long m) {
    long long a, b;
    euclid (x, m, a, b);
    return (a % m + m) % m;
}
long long gcd (const long long &a, const long long &b) {
    if (!b) return a;
    long long x = a, y = b;
    while (x > y ? (x = x % y) : (y = y % x));
    return x + y;
}
long long mul_mod (const long long &a, const long long &b, const long long &mod) {
    long long d = (long long) floor (a * (double) b / mod + 0.5);
    long long ret = a * b - d * mod;
    if (ret < 0) ret += mod;
    return ret;
}
long long llfpm (const long long &x, const long long &n, const long long &mod) {
    long long ans = 1, mul = x, k = n;
    while (k) {
        if (k & 1) ans = mul_mod (ans, mul, mod);
        mul = mul_mod (mul, mul, mod);
        k >>= 1;
    }
    return ans;
}
```

3.2 Discrete Fourier transform

```
/*  Discrete Fourier transform :
        int dft::init (int n) :
            initializes the transformation with dimension n.
            Returns the recommended size.
        void dft::solve (complex *a, int n, int f) :
            transforms array a with dimension n to its image representation.
            Transforms back when f = 1. (n should be 2^k)
*/
template <int MAXN = 1000000>
struct dft {
    typedef std::complex <double> complex;
    complex e[2][MAXN];
```

```
	int init (int n) {
		int len = 1;
		for (; len <= 2 * n; len <<= 1);
		for (int i = 0; i < len; i++) {
			e[0][i] = complex (cos (2 * PI * i / len), sin (2 * PI * i / len));
			e[1][i] = complex (cos (2 * PI * i / len), -sin (2 * PI * i / len));
		}
		return len;
	}
	void solve (complex *a, int n, int f) {
		for (int i = 0, j = 0; i < n; i++) {
			if (i > j) std::swap (a[i], a[j]);
			for (int t = n >> 1; (j ^= t) < t; t >>= 1);
		}
		for (int i = 2; i <= n; i <<= 1)
			for (int j = 0; j < n; j += i)
				for (int k = 0; k < (i >> 1); k++) {
					complex A = a[j + k];
					complex B = e[f][n / i * k] * a[j + k + (i >> 1)];
					a[j + k] = A + B;
					a[j + k + (i >> 1)] = A - B;
				}
		if (f == 1) {
			for (int i = 0; i < n; i++)
				a[i] = complex (a[i].real () / n, a[i].imag ());
		}
	}
};
```

3.3 Fast Fourier transform for integer

```
/*	Fast Fourier transform for integer :
	Usage : int_fft::solve (int a[N], int b[N]) : calculate a[] * b[].
		The result is in a[].
*/
template <const int N = 65536, int L = 15, int MOD = 1000003>
struct int_fft {
	typedef std::complex <double> complex;
	int MASK; complex w[N];
	int_fft () {
		MASK = (1 << L) - 1;
		for (int i = 0; i < N; ++i)
			w[i] = complex (cos(2 * i * PI / N), sin(2 * i * PI / N));
	}
	void FFT (complex p[], int n) {
		for (int i = 1, j = 0; i < n - 1; ++i) {
			for (int s = n; j ^= s >>= 1, ~j & s;);
			if (i < j) swap(p[i], p[j]);
		}
		for (int d = 0; (1 << d) < n; ++d) {
			int m = 1 << d, m2 = m * 2, rm = n >> (d + 1);
			for (int i = 0; i < n; i += m2) {
				for (int j = 0; j < m; ++j) {
					complex &p1 = p[i + j + m], &p2 = p[i + j];
					complex t = w[rm * j] * p1;
					p1 = p2 - t, p2 = p2 + t;
				}
			}
		}
	}
	complex A[N], B[N], C[N], D[N];
	void solve (int a[N], int b[N]) {
		for (int i = 0; i < N; ++i) {
			A[i] = complex (a[i] >> L, a[i] & MASK);
			B[i] = complex (b[i] >> L, b[i] & MASK);
		}
		FFT(A, N), FFT(B, N);
		for (int i = 0; i < N; ++i) {
			int j = (N - i) % N;
			complex da = (A[i] - conj(A[j])) * complex(0, -0.5),
					db = (A[i] + conj(A[j])) * complex(0.5, 0),
					dc = (B[i] - conj(B[j])) * complex(0, -0.5),
					dd = (B[i] + conj(B[j])) * complex(0.5, 0);
			C[j] = da * dd + da * dc * complex(0, 1);
			D[j] = db * dd + db * dc * complex(0, 1);
		}
		FFT(C, N), FFT(D, N);
		for (int i = 0; i < N; ++i) {
			long long da = (long long)(C[i].imag() / N + 0.5) % MOD,
				db = (long long)(C[i].real() / N + 0.5) % MOD,
				dc = (long long)(D[i].imag() / N + 0.5) % MOD,
				dd = (long long)(D[i].real() / N + 0.5) % MOD;
			a[i] = ((dd << (L * 2)) + ((db + dc) << L) + da) % MOD;
		}
	}
};
```

3.4 Number-theoretic transform

```
/* Number-theoretic transform :
	void ntt::solve (int *a, int n, int f, int mod, int prt) :
		transforms a[n] to its image representation.
		Converts back if f = 1. (n should be 2^k)
		Requries specific mod and corresponding prt to work. (given in MOD and PRT)
	int ntt::crt (int *a, int mod) :
		fixes the results a from module 3 primes to a certain module mod.
*/
```

```
template <int MAXN = 1000000>
struct ntt {
    void solve (int *a, int n, int f, int mod, int prt) {
        for (register int i = 0, j = 0; i < n; i++) {
            if (i > j) std::swap (a[i], a[j]);
            for (register int t = n >> 1; (j ^= t) < t; t >>= 1);
        }
        for (register int i = 2; i <= n; i <<= 1) {
            static int exp[MAXN];
            exp[0] = 1;
            exp[1] = fpm (prt, (mod - 1) / i, mod);
            if (f == 1) exp[1] = fpm (exp[1], mod - 2, mod);
            for (register int k = 2; k < (i >> 1); k++) {
                exp[k] = int (1ll * exp[k - 1] * exp[1] % mod);
            }
            for (register int j = 0; j < n; j += i) {
                for (register int k = 0; k < (i >> 1); k++) {
                    register int &pA = a[j + k], &pB = a[j + k + (i >> 1)];
                    register int A = pA, B = int (1ll * pB * exp[k] % mod);
                    pA = (A + B) % mod;
                    pB = (A - B + mod) % mod;
                }
            }
        }
        if (f == 1) {
            register int rev = fpm (n, mod - 2, mod);
            for (register int i = 0; i < n; i++) {
                a[i] = int (1ll * a[i] * rev % mod);
            }
        }
    }
    int MOD[3] = {1045430273, 1051721729, 1053818881}, PRT[3] = {3, 6, 7};
    int crt (int *a, int mod) {
        static int inv[3][3];
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                inv[i][j] = (int) inverse (MOD[i], MOD[j]);
        static int x[3];
        for (int i = 0; i < 3; i++) {
            x[i] = a[i];
            for (int j = 0; j < i; j++) {
                int t = (x[i] - x[j] + MOD[i]) % MOD[i];
                if (t < 0) t += MOD[i];
                x[i] = int (1LL * t * inv[j][i] % MOD[i]);
            }
        }
        int sum = 1, ret = x[0] % mod;
        for (int i = 1; i < 3; i ++) {
            sum = int (1LL * sum * MOD[i - 1] % mod);
            ret += int (1LL * x[i] * sum % mod);
            if (ret >= mod) ret -= mod;
        }
        return ret;
    }
};
```

3.5 Chinese remainder theorem

```
/*  Chinese remainder theroem :
        bool crt::solve (const std::vector <std::pair<long long, long long> > &input,
                         std::pair<long long, long long> &output) :
            solves for an integer set x = output.first + k * output.second
                that satisfies x % input[i].second = input[i].first.
            Returns whether a solution exists.
*/
struct crt {
    long long fix (const long long &a, const long long &b) {
        return (a % b + b) % b;
    }
    bool solve (const std::vector <std::pair <long long, long long> > &input,
                std::pair <long long, long long> &output) {
        output = std::make_pair (1, 1);
        for (int i = 0; i < (int) input.size (); ++i) {
            long long number, useless;
            euclid (output.second, input[i].second, number, useless);
            long long divisor = gcd (output.second, input[i].second);
            if ((input[i].first - output.first) % divisor) {
                return false;
            }
            number *= (input[i].first - output.first) / divisor;
            number = fix (number, input[i].second);
            output.first += output.second * number;
            output.second *= input[i].second / divisor;
            output.first = fix (output.first, output.second);
        }
        return true;
    }
};
```

3.6 Linear Recurrence

```
/*  Linear recurrence :
    Calculating k-th term of linear recurrence sequence
    Complexity: O(n^2 * log (k)) each operation
    Input (constructor) :
        vector<int> - first n terms
        vector<int> - transition function
    Output (calc (k)) : int - the kth term mod MOD
```

```
    Example :
        In : {2, 1} {2, 1} a1 = 2, a2 = 1, an = 2an-1 + an-2
        Out : calc (3) = 5, calc (10007) = 959155122 (MOD 1E9+7)
*/
struct linear_rec {
    const int LOG = 30, MOD = 1E9 + 7;
    int n;
    std::vector <int> first, trans;
    std::vector <std::vector <int>> bin;
    std::vector <int> add (std::vector <int> &a, std::vector <int> &b) {
        std::vector <int> result(n * 2 + 1, 0);
        for (int i = 0; i <= n; ++i) {
            for (int j = 0; j <= n; ++j) {
                result[i + j] += (long long) a[i] * b[j] % MOD;
                if (result[i + j] >= MOD) {
                    result[i + j] -= MOD;
                }
            }
        }
        for (int i = 2 * n; i > n; --i) {
            for (int j = 0; j < n; ++j) {
                result[i - 1 - j] += (long long) result[i] * trans[j] % MOD;
                if (result[i - 1 - j] >= MOD)
                    result[i - 1 - j] -= MOD;
            }
            result[i] = 0;
        }
        result.erase(result.begin() + n + 1, result.end());
        return result;
    }
    linear_rec (const std::vector <int> &first, const std::vector <int> &trans) :
        first(first), trans(trans) {
        n = first.size();
        std::vector <int> a(n + 1, 0);
        a[1] = 1;
        bin.push_back(a);
        for (int i = 1; i < LOG; ++i)
            bin.push_back(add(bin[i - 1], bin[i - 1]));
    }
    int solve (int k) {
        std::vector <int> a(n + 1, 0);
        a[0] = 1;
        for (int i = 0; i < LOG; ++i)
            if (k >> i & 1)
                a = add(a, bin[i]);
        int ret = 0;
        for (int i = 0; i < n; ++i)
            if ((ret += (long long) a[i + 1] * first[i] % MOD) >= MOD)
                ret -= MOD;
        return ret;
    }
};
```

3.7 Baby step giant step algorithm

```
/*  Baby step giant step algorithm :
        Solves a^x = b (mod c) in O (sqrt (c) * log (sqrt (c))).
        int bsgs::solve (int a, int b, int c) :
            returns -1 when no solution.
*/
struct bsgs {
    int solve (int a, int b, int c) {
        std::map <int, int> bs;
        int m = (int) sqrt ((double) c) + 1, res = 1;
        for (int i = 0; i < m; ++i) {
            if (bs.find (res) == bs.end ()) bs[res] = i;
            res = int (1LL * res * a % c);
        }
        int mul = 1, inv = (int) inverse (a, c);
        for (int i = 0; i < m; ++i)
            mul = int (1LL * mul * inv % c);
        res = b % c;
        for (int i = 0; i < m; ++i) {
            if (bs.find (res) != bs.end ())
                return i * m + bs[res];
            res = int (1LL * res * mul % c);
        }
        return -1;
    }
};
```

3.8 Miller Rabin primality test

```
/*  Miller Rabin :
        bool miller_rabin::solve (const long long &) :
            tests whether a certain integer is prime.
*/
struct miller_rabin {
    int BASE[12] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
    bool check (const long long &prime, const long long &base) {
        long long number = prime - 1;
        for (; ~number & 1; number >>= 1);
        long long result = llfpm (base, number, prime);
        for (; number != prime - 1 && result != 1 && result != prime - 1; number <<= 1)
            result = mul_mod (result, result, prime);
        return result == prime - 1 || (number & 1) == 1;
    }
```

```
    bool solve (const long long &number) {
        if (number < 2) return false;
        if (number < 4) return true;
        if (~number & 1) return false;
        for (int i = 0; i < 12 && BASE[i] < number; ++i)
            if (!check (number, BASE[i]))
                return false;
        return true;
    }
};
```

3.9 Pollard's Rho algorithm

```
/*  Pollard Rho :
        std::vector <long long> pollard_rho::solve (const long long &) :
            factorizes an integer.
*/
struct pollard_rho {
    miller_rabin is_prime;
    const long long threshold = 13E9;
    long long factorize (const long long &number, const long long &seed) {
        long long x = rand() % (number - 1) + 1, y = x;
        for (int head = 1, tail = 2; ; ) {
            x = mul_mod (x, x, number);
            x = (x + seed) % number;
            if (x == y)
                return number;
            long long answer = gcd (abs (x - y), number);
            if (answer > 1 && answer < number)
                return answer;
            if (++head == tail) {
                y = x;
                tail <<= 1;
            }
        }
    }
    void search (const long long &number, std::vector<long long> &divisor) {
        if (number > 1)  {
            if (is_prime.solve (number))
                divisor.push_back (number);
            else {
                long long factor = number;
                for (; factor >= number;
                        factor = factorize (number, rand () % (number - 1) + 1));
                search (number / factor, divisor);
                search (factor, divisor);
            }
        }
    }
    std::vector <long long> solve (const long long &number) {
        std::vector <long long> ans;
        if (number > threshold)
            search (number, ans);
        else {
            long long rem = number;
            for (long long i = 2; i * i <= rem; ++i)
                while (!(rem % i)) {
                    ans.push_back (i);
                    rem /= i;
                }
            if (rem > 1) ans.push_back (rem);
        }
        return ans;
    }
};
```

3.10 Adaptive Simpson's method

```
/*  Adaptive Simpson's method :
        double simpson::solve (double (*f) (double), double l, double r, double eps) :
            integrates f over (l, r) with error eps.
*/
struct simpson {
    double area (double (*f) (double), double l, double r) {
        double m = l + (r - l) / 2;
        return (f (l) + 4 * f (m) + f (r)) * (r - l) / 6;
    }
    double solve (double (*f) (double), double l, double r, double eps, double a) {
        double m = l + (r - l) / 2;
        double left = area (f, l, m), right = area (f, m, r);
        if (fabs (left + right - a) <= 15 * eps) return left + right + (left + right - a) / 15.0;
        return solve (f, l, m, eps / 2, left) + solve (f, m, r, eps / 2, right);
    }
    double solve (double (*f) (double), double l, double r, double eps) {
        return solve (f, l, r, eps, area (f, l, r));
    }
};
```

Chapter 4

Trial of geometry

4.1 Constants and basic functions

```
/*  Constants & basic functions :
        EPS : fixes the possible error of data.
            i.e. x == y iff |x - y| < EPS.
        PI : the value of PI.
        int sgn (const double &x) : returns the sign of x.
        int cmp (const double &x, const double &y) : returns the sign of x - y.
        double sqr (const double &x) : returns x * x.
*/
const double EPS = 1E-8;
const double PI = acos (-1);
int sgn (const double &x) { return x < -EPS ? -1 : x > EPS; }
int cmp (const double &x, const double &y) { return sgn (x - y); }
double sqr (const double &x) { return x * x; }
```

4.2 Point class

```
/*  struct point : defines a point and its various utility.
        point (const double &x, const double &y) gives a point at (x, y).
        It also represents a vector on a 2D plane.
        point unit () const : returns the unit vector of (x, y).
        point rot90 () const :
            returns a point rotated 90 degrees counter-clockwise with respect to the origin.
        point _rot () const : same as above except clockwise.
        point rotate (const double &t) const : returns a point rotated t radian(s) counter-clockwise.
        Operators are mostly vector operations. i.e. vector +, -, *, / and dot/det product.
*/
struct point {
    double x, y;
    explicit point (const double &x = 0, const double &y = 0) : x (x), y (y) {}
    double norm () const { return sqrt (x * x + y * y); }
    double norm2 () const { return x * x + y * y; }
    point unit () const {
        double l = norm ();
        return point (x / l, y / l);
    }
    point rot90 () const {return point (-y, x); }
    point _rot90 () const {return point (y, -x); }
    point rot (const double &t) const {
        double c = cos (t), s = sin (t);
        return point (x * c - y * s, x * s + y * c);
    }
};
bool operator == (const point &a, const point &b) {
    return cmp (a.x, b.x) == 0 && cmp (a.y, b.y) == 0;
}
bool operator != (const point &a, const point &b) {
    return ! (a == b);
}
bool operator < (const point &a, const point &b) {
    if (cmp (a.x, b.x) == 0) return cmp (a.y, b.y) < 0;
    return cmp (a.x, b.x) < 0;
}
point operator - (const point &a) { return point (-a.x, -a.y); }
point operator + (const point &a, const point &b) {
    return point (a.x + b.x, a.y + b.y);
}
point operator - (const point &a, const point &b) {
    return point (a.x - b.x, a.y - b.y);
}
point operator * (const point &a, const double &b) {
    return point (a.x * b, a.y * b);
}
point operator / (const point &a, const double &b) {
    return point (a.x / b, a.y / b);
}
double dot (const point &a, const point &b) {
    return a.x * b.x + a.y * b.y;
}
double det (const point &a, const point &b) {
    return a.x * b.y - a.y * b.x;
```

```
}
double dis (const point &a, const point &b) {
    return sqrt (sqr (a.x - b.x) + sqr (a.y - b.y));
}
```

4.3 Line class

```
/*  struct line : defines a line (segment) based on two points, s and t.
        line (const point &s, const point &t) gives a basic line from s to t.
        double length () const : returns the length of the segment.
*/
struct line {
    point s, t;
    explicit line (const point &s = point (), const point &t = point ()) : s (s), t (t) {}
    double length () const { return dis (s, t); }
};
/*  Point & line interactions :
        bool point_on_segment (const point &a, const line &b) : checks if a is on b.
        bool intersect_judgement (const line &a, const line &b) : checks if segment a and b intersect.
        point line_intersect (const line &a, const line &b) : returns the intersection of a and b.
            Fails on colinear or parallel situations.
        double point_to_line (const point &a, const line &b) : returns the distance from a to b.
        double point_to_segment (const point &a, const lint &b) : returns the distance from a to b.
            i.e. the minimized length from a to segment b.
        bool in_polygon (const point &p, const std::vector <point> &po) :
            checks if a is in a polygon with vetices po (clockwise or counter-clockwise order).
        double polygon_area (const std::vector <point> &a) :
            returns the signed area of polygon a (positive for counter-clockwise order, and vise-versa).
        point project_to_line (const point &a, const line &b) :
            returns the projection of a on b,
*/
bool point_on_segment (const point &a, const line &b) {
    return sgn (det (a - b.s, b.t - b.s)) == 0 && sgn (dot (b.s - a, b.t - a)) <= 0;
}
bool two_side (const point &a, const point &b, const line &c) {
    return sgn (det (a - c.s, c.t - c.s)) * sgn (det (b - c.s, c.t - c.s)) < 0;
}
bool intersect_judgment (const line &a, const line &b) {
    if (point_on_segment (b.s, a) || point_on_segment (b.t, a)) return true;
    if (point_on_segment (a.s, b) || point_on_segment (a.t, b)) return true;
    return two_side (a.s, a.t, b) && two_side (b.s, b.t, a);
}
point line_intersect (const line &a, const line &b) {
    double s1 = det (a.t - a.s, b.s - a.s);
    double s2 = det (a.t - a.s, b.t - a.s);
    return (b.s * s2 - b.t * s1) / (s2 - s1);
}
double point_to_line (const point &a, const line &b) {
    return fabs (det (b.t - b.s, a - b.s)) / dis (b.s, b.t);
}
point project_to_line (const point &a, const line &b) {
    return b.s + (b.t - b.s) * (dot (a - b.s, b.t - b.s) / (b.t - b.s).norm2 ());
}
double point_to_segment (const point &a, const line &b) {
    if (sgn (dot (b.s - a, b.t - b.s) * dot (b.t - a, b.t - b.s)) <= 0)
        return fabs (det (b.t - b.s, a - b.s)) / dis (b.s, b.t);
    return std::min (dis (a, b.s), dis (a, b.t));
}
bool in_polygon (const point &p, const std::vector <point> & po) {
    int n = (int) po.size ();
    int counter = 0;
    for (int i = 0; i < n; ++i) {
        point a = po[i], b = po[ (i + 1) % n];
        /*          The following statement checks is p is on the border of the polygon.
                    The boolean returned may be changed if necessary.
                        i.e. the algorithm may check if p is strictly in the polygon.
        */
        if (point_on_segment (p, line (a, b))) return true;
        int x = sgn (det (p - a, b - a)), y = sgn (a.y - p.y), z = sgn (b.y - p.y);
        if (x > 0 && y <= 0 && z > 0) counter++;
        if (x < 0 && z <= 0 && y > 0) counter--;
    }
    return counter != 0;
}
double polygon_area (const std::vector <point> &a) {
    double ans = 0.0;
    for (int i = 0; i < (int) a.size (); ++i)
        ans += det (a[i], a[ (i + 1) % a.size ()]) / 2.0;
    return ans;
}
```

4.4 Centers of a triangle

```
/*  Centers of a triangle :
        returns various centers of a triangle with vertices (a, b, c).
*/
point incenter (const point &a, const point &b, const point &c) {
    double p = dis (a, b) + dis (b, c) + dis (c, a);
    return (a * dis (b, c) + b * dis (c, a) + c * dis (a, b)) / p;
}
point circumcenter (const point &a, const point &b, const point &c) {
    point p = b - a, q = c - a, s (dot (p, p) / 2, dot (q, q) / 2);
    double d = det (p, q);
    return a + point (det (s, point (p.y, q.y)), det (point (p.x, q.x), s)) / d;
}
```

```
point orthocenter (const point &a, const point &b, const point &c) {
    return a + b + c - circumcenter (a, b, c) * 2.0;
}
```

4.5 Fermat point

```
/*  Fermat point :
        point fermat_point (const point &a, const point &b, const point &c) :
            returns a point p that minimizes |pa| + |pb| + |pc|.
*/
point fermat_point (const point &a, const point &b, const point &c) {
    if (a == b) return a;
    if (b == c) return b;
    if (c == a) return c;
    double ab = dis (a, b), bc = dis (b, c), ca = dis (c, a);
    double cosa = dot (b - a, c - a) / ab / ca;
    double cosb = dot (a - b, c - b) / ab / bc;
    double cosc = dot (b - c, a - c) / ca / bc;
    double sq3 = PI / 3.0;
    point mid;
    if (sgn (cosa + 0.5) < 0) mid = a;
    else if (sgn (cosb + 0.5) < 0) mid = b;
    else if (sgn (cosc + 0.5) < 0) mid = c;
    else if (sgn (det (b - a, c - a)) < 0)
        mid = line_intersect (line (a, b + (c - b).rot (sq3)), line (b, c + (a - c).rot (sq3)));
    else
        mid = line_intersect (line (a, c + (b - c).rot (sq3)), line (c, b + (a - b).rot (sq3)));
    return mid;
}
```

4.6 Circle class

```
/*  struct circle : defines a circle.
    circle (point c, double r) gives a circle with center c and radius r.
*/
struct circle {
    point c;
    double r;
    explicit circle (point c = point (), double r = 0) : c (c), r (r) {}
};
bool operator == (const circle &a, const circle &b) {
    return a.c == b.c && cmp (a.r, b.r) == 0;
}
bool operator != (const circle &a, const circle &b) {
    return ! (a == b);
}
/*  Circle interaction :
        bool in_circle (const point &a, const circle &b) : checks if a is in or on b.
        circle make_circle (const point &a, const point &b) :
            generates a circle with diameter ab.
        circle make_circle (const point &a, const point &b, const point &c) :
            generates a circle passing a, b and c.
        std::pair <point, point> line_circle_intersect (const line &a, const circle &b) :
            returns the intersections of a and b.
            Fails if a and b do not intersect.
        std::pair <point, point> circle_intersect (const circle &a, const circle &b):
            returns the intersections of a and b.
            Fails if a and b do not intersect.
        std::pair <line, line> tangent (const point &a, const circle &b) :
            returns the tangent lines of b passing through a.
            Fails if a is in b.
*/
bool in_circle (const point &a, const circle &b) {
    return cmp (dis (a, b.c), b.r) <= 0;
}
circle make_circle (const point &a, const point &b) {
    return circle ((a + b) / 2, dis (a, b) / 2);
}
circle make_circle (const point &a, const point &b, const point &c) {
    point p = circumcenter (a, b, c);
    return circle (p, dis (p, a));
}
std::pair <point, point> line_circle_intersect (const line &a, const circle &b) {
    double x = sqrt (sqr (b.r) - sqr (point_to_line (b.c, a)));
    return std::make_pair (project_to_line (b.c, a) + (a.s - a.t).unit () * x,
                           project_to_line (b.c, a) - (a.s - a.t).unit () * x);
}
point __circle_intersect (const circle &a, const circle &b) {
    point r = (b.c - a.c).unit ();
    double d = dis (a.c, b.c);
    double x = .5 * ((sqr (a.r) - sqr (b.r)) / d + d);
    double h = sqrt (sqr (a.r) - sqr (x));
    return a.c + r * x + r.rot90 () * h;
}
std::pair <point, point> circle_intersect (const circle &a, const circle &b) {
    return std::make_pair (__circle_intersect (a, b), __circle_intersect (b, a));
}
std::pair <line, line> tangent (const point &a, const circle &b) {
    circle p = make_circle (a, b.c);
    auto d = circle_intersect (p, b);
    return std::make_pair (line (d.first, a), line (d.second, a));
}
```

4.7 Convex hull

```
/*  Convex hull :
        std::vector <point> convex_hull (std::vector <point> a) :
            returns the convex hull of point set a (counter-clockwise).
*/
bool turn_left (const point &a, const point &b, const point &c) {
    return sgn (det (b - a, c - a)) >= 0;
}
bool turn_right (const point &a, const point &b, const point &c) {
    return sgn (det (b - a, c - a)) <= 0;
}
std::vector <point> convex_hull (std::vector <point> a) {
    int n = (int) a.size (), cnt = 0;
    std::sort (a.begin (), a.end ());
    std::vector <point> ret;
    for (int i = 0; i < n; ++i) {
        while (cnt > 1 && turn_left (ret[cnt - 2], a[i], ret[cnt - 1])) {
            --cnt;
            ret.pop_back ();
        }
        ret.push_back (a[i]);
        ++cnt;
    }
    int fixed = cnt;
    for (int i = n - 1; i >= 0; --i) {
        while (cnt > fixed && turn_left (ret[cnt - 2], a[i], ret[cnt - 1])) {
            --cnt;
            ret.pop_back ();
        }
        ret.push_back (a[i]);
        ++cnt;
    }
    ret.pop_back ();
    return ret;
}
```

4.8 Minimum circle

```
/*  Minimum circle of a point set :
    circle minimum_circle (std::vector <point> p) : returns the minimum circle of point set p.
*/
circle minimum_circle (std::vector <point> p) {
    circle ret;
    std::random_shuffle (p.begin (), p.end ());
    for (int i = 0; i < (int) p.size (); ++i)
        if (!in_circle (p[i], ret)) {
            ret = circle (p[i], 0);
            for (int j = 0; j < i; ++j)
                if (!in_circle (p[j], ret)) {
                    ret = make_circle (p[j], p[i]);
                    for (int k = 0; k < j; ++k)
                        if (!in_circle (p[k], ret)) ret = make_circle (p[i], p[j], p[k]);
                }
        }
    return ret;
}
```

4.9 Half plane intersection

```
/*  Online half plane intersection (complexity = O(c.size ())) :
        std::vector <point> cut (const std::vector<point> &c, line p) :
            returns the convex polygon cutting convex polygon c with half plane p.
                (left hand with respect to vector p)
            If such polygon does not exist, returns an empty set.
            e.g.
                static const double BOUND = 1e5;
                convex.clear ();
                convex.push_back (point (-BOUND, -BOUND));
                convex.push_back (point (BOUND, -BOUND));
                convex.push_back (point (BOUND, BOUND));
                convex.push_back (point (-BOUND, BOUND));
                convex = cut (convex, line(point, point));
                if (convex.empty ()) { ... }
*/
std::vector <point> cut (const std::vector<point> &c, line p) {
    std::vector <point> ret;
    if (c.empty ()) return ret;
    for (int i = 0; i < (int) c.size (); ++i) {
        int j = (i + 1) % (int) c.size ();
        if (!turn_right (p.s, p.t, c[i])) ret.push_back (c[i]);
        if (two_side (c[i], c[j], p))
            ret.push_back (line_intersect (p, line (c[i], c[j])));
    }
    return ret;
}
/*  Offline half plane intersection (complexity = O(nlogn), n = h.size ()) :
        std::vector <point> half_plane_intersect (std::vector <line> h) :
            returns the intersection of half planes h.
                (left hand with respect to the vector)
            If such polygon does not exist, returns an empty set.
*/
bool turn_left (const line &l, const point &p) {
    return turn_left (l.s, l.t, p);
}
std::vector <point> half_plane_intersect (std::vector <line> h) {
    typedef std::pair <double, line> polar;
    std::vector <polar> g;
    g.resize (h.size ());
    for (int i = 0; i < (int) h.size (); ++i) {
```

```
        point v = h[i].t - h[i].s;
        g[i] = std::make_pair (atan2 (v.y, v.x), h[i]);
    }
    sort (g.begin (), g.end (), [] (const polar &a, const polar &b) {
        if (cmp (a.first, b.first) == 0)
            return sgn (det (a.second.t - a.second.s, b.second.t - a.second.s)) < 0;
        else
            return cmp (a.first, b.first) < 0;
    });
    h.resize (std::unique (g.begin (), g.end (), [] (const polar &a, const polar &b) {
        return cmp (a.first, b.first) == 0;
    }) - g.begin ());
    for (int i = 0; i < (int) h.size (); ++i)
        h[i] = g[i].second;
    int fore = 0, rear = -1;
    std::vector <line> ret;
    for (int i = 0; i < (int) h.size (); ++i) {
        while (fore < rear && !turn_left (h[i], line_intersect (ret[rear - 1], ret[rear]))) {
            --rear;
            ret.pop_back ();
        }
        while (fore < rear && !turn_left (h[i], line_intersect (ret[fore], ret[fore + 1])))
            ++fore;
        ++rear;
        ret.push_back (h[i]);
    }
    while (rear - fore > 1 && !turn_left (ret[fore], line_intersect (ret[rear - 1], ret[rear]))) {
        --rear;
        ret.pop_back ();
    }
    while (rear - fore > 1 && !turn_left (ret[rear], line_intersect (ret[fore], ret[fore + 1])))
        ++fore;
    if (rear - fore < 2) return std::vector <point> ();
    std::vector <point> ans;
    ans.resize (ret.size ());
    for (int i = 0; i < (int) ret.size (); ++i)
        ans[i] = line_intersect (ret[i], ret[ (i + 1) % ret.size ()]);
    return ans;
}
```

4.10 Intersection of a polygon and a circle

```
/*  Intersection of a polygon and a circle :
        double polygon_circle_intersect::solve (const std::vector <point> &p, const circle &c) :
            returns the area of intersection of polygon p (vertices in either order) and c.
*/
struct polygon_circle_intersect {
    //  The area of the sector with center (0, 0), radius r and segment ab.
    double sector_area (const point &a, const point &b, const double &r) {
        double c = (2.0 * r * r - (a - b).norm2 ()) / (2.0 * r * r);
        double al = acos (c);
        return r * r * al / 2.0;
    }
    //  The area of triangle (a, b, (0, 0)) intersecting circle (point (), r).
    double area (const point &a, const point &b, const double &r) {
        double dA = dot (a, a), dB = dot (b, b), dC = point_to_segment (point (), line (a, b)), ans = 0.0;
        if (sgn (dA - r * r) <= 0 && sgn (dB - r * r) <= 0) return det (a, b) / 2.0;
        point tA = a.unit () * r;
        point tB = b.unit () * r;
        if (sgn (dC - r) > 0) return sector_area (tA, tB, r);
        std::pair <point, point> ret = line_circle_intersect (line (a, b), circle (point (), r));
        if (sgn (dA - r * r) > 0 && sgn (dB - r * r) > 0) {
            ans += sector_area (tA, ret.first, r);
            ans += det (ret.first, ret.second) / 2.0;
            ans += sector_area (ret.second, tB, r);
            return ans;
        }
        if (sgn (dA - r * r) > 0)
            return det (ret.first, b) / 2.0 + sector_area (tA, ret.first, r);
        else
            return det (a, ret.second) / 2.0 + sector_area (ret.second, tB, r);
    }
    //  Main procedure.
    double solve (const std::vector <point> &p, const circle &c) {
        double ret = 0.0;
        for (int i = 0; i < (int) p.size (); ++i) {
            int s = sgn (det (p[i] - c.c, p[ (i + 1) % p.size ()] - c.c));
            if (s > 0)
                ret += area (p[i] - c.c, p[ (i + 1) % p.size ()] - c.c, c.r);
            else
                ret -= area (p[ (i + 1) % p.size ()] - c.c, p[i] - c.c, c.r);
        }
        return fabs (ret);
    }
};
```

4.11 Union of circles

```
/*  Union of circles :
            returns the union of circle set c.
            The i-th element is the area covered with at least i circles.
*/
template <int MAXN = 500>
struct union_circle {
    int C;
    circle c[MAXN];
    double area[MAXN];
    struct event {
```

```
        point p; double ang; int delta;
        event (point p = point (), double ang = 0, int delta = 0) : p(p), ang(ang), delta(delta) {}
        bool operator < (const event &a) { return ang < a.ang; }
    };
    void addevent(const circle &a, const circle &b, std::vector<event> &evt, int &cnt) {
        double d2 = (a.c - b.c).norm2(), dRatio = ((a.r - b.r) * (a.r + b.r) / d2 + 1) / 2,
            pRatio = sqrt (std::max (0., -(d2 - sqr(a.r - b.r)) * (d2 - sqr(a.r + b.r)) / (d2 * d2 * 4)));
        point d = b.c - a.c, p = d.rot(PI / 2),
            q0 = a.c + d * dRatio + p * pRatio,
            q1 = a.c + d * dRatio - p * pRatio;
        double ang0 = atan2 ((q0 - a.c).y, (q0 - a.c).x), ang1 = atan2 ((q1 - a.c).x, (q1 - a.c).y);
        evt.emplace_back(q1, ang1, 1); evt.emplace_back(q0, ang0, -1);
        cnt += ang1 > ang0;
    }
    bool issame(const circle &a, const circle &b) {
        return sgn((a.c - b.c).norm()) == 0 && sgn(a.r - b.r) == 0;
    }
    bool overlap(const circle &a, const circle &b) {
        return sgn(a.r - b.r - (a.c - b.c).norm()) >= 0;
    }
    bool intersect(const circle &a, const circle &b) {
        return sgn((a.c - b.c).norm() - a.r - b.r) < 0;
    }
    void solve() {
        std::fill (area, area + C + 2, 0);
        for (int i = 0; i < C; ++i) {
            int cnt = 1;
            std::vector<event> evt;
            for (int j = 0; j < i; ++j) if (issame(c[i], c[j])) ++cnt;
            for (int j = 0; j < C; ++j)
                if (j != i && !issame(c[i], c[j]) && overlap(c[j], c[i]))
                    ++cnt;
            for (int j = 0; j < C; ++j)
                if (j != i && !overlap(c[j], c[i]) && !overlap(c[i], c[j]) && intersect(c[i], c[j]))
                    addevent(c[i], c[j], evt, cnt);
            if (evt.empty()) area[cnt] += PI * c[i].r * c[i].r;
            else {
                std::sort(evt.begin(), evt.end());
                evt.push_back(evt.front());
                for (int j = 0; j + 1 < (int)evt.size(); ++j) {
                    cnt += evt[j].delta;
                    area[cnt] += det(evt[j].p, evt[j + 1].p) / 2;
                    double ang = evt[j + 1].ang - evt[j].ang;
                    if (ang < 0) ang += PI * 2;
                    area[cnt] += ang * c[i].r * c[i].r / 2 - sin(ang) * c[i].r * c[i].r / 2;
                }
            }
        }
    }
};
```

Chapter 5

Trial of graph

5.1 Constants and edge lists

```
const int INF = 1E9;
/*  Edge list:
        Various kinds of edge list.
*/
template <int MAXN = 100000, int MAXM = 100000>
struct edge_list {
    int size;
    int begin[MAXN], dest[MAXM], next[MAXM];
    void clear (int n) {
        size = 0;
        std::fill (begin, begin + n, -1);
    }
    edge_list (int n = MAXN) {
        clear (n);
    }
    void add_edge (int u, int v) {
        dest[size] = v; next[size] = begin[u]; begin[u] = size++;
    }
};
template <int MAXN = 100000, int MAXM = 100000>
struct cost_edge_list {
    int size;
    int begin[MAXN], dest[MAXM], next[MAXM], cost[MAXM];
    void clear (int n) {
        size = 0;
        std::fill (begin, begin + n, -1);
    }
    cost_edge_list (int n = MAXN) {
        clear (n);
    }
    void add_edge (int u, int v, int c) {
        dest[size] = v; next[size] = begin[u]; cost[size] = c; begin[u] = size++;
    }
};
template <int MAXN = 100000, int MAXM = 100000>
struct flow_edge_list {
    int size;
    int begin[MAXN], dest[MAXM], next[MAXM], flow[MAXM], inv[MAXM];
    void clear (int n) {
        size = 0;
        std::fill (begin, begin + n, -1);
    }
    flow_edge_list (int n = MAXN) {
        clear (n);
    }
    void add_edge (int u, int v, int f) {
        dest[size] = v; next[size] = begin[u]; flow[size] = f; inv[size] = size + 1; begin[u] = size++;
        dest[size] = u; next[size] = begin[v]; flow[size] = 0; inv[size] = size - 1; begin[v] = size++;
    }
};
template <int MAXN = 100000, int MAXM = 100000>
struct cost_flow_edge_list {
    int size;
    int begin[MAXN], dest[MAXM], next[MAXM], cost[MAXM], flow[MAXM], inv[MAXM];
    void clear (int n) {
        size = 0;
        std::fill (begin, begin + n, -1);
    }
    cost_flow_edge_list (int n = MAXN) {
        clear (n);
    }
    void add_edge (int u, int v, int c, int f) {
        dest[size] = v; next[size] = begin[u]; cost[size] = c;
        flow[size] = f; inv[size] = size + 1; begin[u] = size++;
        dest[size] = u; next[size] = begin[v]; cost[size] = -c;
        flow[size] = 0; inv[size] = size - 1; begin[v] = size++;
    }
};
```

5.2 SPFA improved

```
/*  SPFA :
```

```
        Shortest path fast algorithm. (with SLF and LLL)
        bool spfa::solve (const cost_edge_list &e, int n, int s) :
            dist[] gives the distance from s.
            last[] gives the previous vertex.
*/
template <int MAXN = 100000, int MAXM = 100000>
struct spfa {
    int dist[MAXN], last[MAXN];
    int queue[MAXN], cnt[MAXN];
    bool inq[MAXN];
    bool solve (const cost_edge_list <MAXN, MAXM> &e, int n, int s) {
        std::fill (dist, dist + MAXN, INF);
        std::fill (last, last + MAXN, -1);
        std::fill (cnt, cnt + MAXN, 0);
        std::fill (inq, inq + MAXN, false);
        int p = 0, q = 1, size = 1;
        long long avg = 0;
        dist[s] = 0; queue[0] = s; inq[s] = true;
        while (p != q) {
            int n = queue[p]; p = (p + 1) % MAXN;
            if (1LL * dist[n] * size > avg) {
                queue[q] = n;
                q = (q + 1) % MAXN;
                continue;
            }
            inq[n] = false; avg -= dist[n]; --size;
            for (int i = e.begin[n]; ~i; i = e.next[i]) {
                int v = e.dest[i];
                if (dist[v] > dist[n] + e.cost[i]) {
                    dist[v] = dist[n] + e.cost[i]; last[v] = n;
                    if (!inq[v]) {
                        if (++cnt[v] > n) return false;
                        inq[v] = true; avg += dist[v]; --size;
                        if (dist[v] < dist[queue[p]])
                            queue[p = (p + MAXN - 1) % MAXN] = v;
                        else {
                            queue[q] = v;
                            q = (q + 1) % MAXN;
                        }
                    }
                }
            }
        }
        return true;
    }
};
```

5.3 Dijkstra's shortest path algorithm

```
/*  Dijkstra :
        Shortest path algorithm.
*/
template <int MAXN = 100000, int MAXM = 100000>
struct dijkstra {
    int dist[MAXN], last[MAXN];
    bool vis[MAXN];
    void solve (const cost_edge_list <MAXN, MAXM> &e, int s) {
        std::priority_queue <std::pair <int, int>, std::vector <std::pair <int, int> >,
            std::greater <std::pair <int, int> > > queue;
        std::fill (dist, dist + MAXN, INF);
        std::fill (last, last + MAXN, -1);
        std::fill (vis, vis + MAXN, false);
        dist[s] = 0;
        queue.push (std::make_pair (0, s));
        while (!queue.empty ()) {
            int n = queue.top ().second; queue.pop (); vis[n] = true;
            for (int i = e.begin[n]; ~i; i = e.next[i]) {
                int v = e.dest[i];
                if (dist[v] > dist[n] + e.cost[i]) {
                    dist[v] = dist[n] + e.cost[i]; last[v] = n;
                    queue.push (std::make_pair (dist[v], v));
                }
            }
            while (!queue.empty () && vis[queue.top ().second]) queue.pop ();
        }
    }
};
```

5.4 Tarjan

```
/*  Tarjan :
        returns strongly connected components.
        void tarjan::solve (const edge_list &, int) :
            component[] gives which component a vertex belongs to.
*/
template <int MAXN = 100000, int MAXM = 100000>
struct tarjan {
    int component[MAXN], component_size;
    int dfn[MAXN], low[MAXN], ins[MAXN], s[MAXN], s_s, ind;
    void dfs (const edge_list <MAXN, MAXM> &e, int u) {
        dfn[u] = low[u] = ind++;
        s[s_s++] = u;
        for (int i = e.begin[u]; ~i; i = e.next[i]) {
            if (!~dfn[e.dest[i]]) {
                dfs (e, e.dest[i]);
                low[u] = std::min (low[u], low[e.dest[i]]);
```

```
            } else if (ins[e.dest[i]])
                low[u] = std::min (low[u], dfn[e.dest[i]]);
        }
        if (dfn[u] == low[u]) {
            do {
                component[s[--s_s]] = component_size; ins[s[s_s]] = false;
            } while (s[s_s] != u);
            ++component_size;
        }
    }
    void solve (const edge_list <MAXN, MAXM> &e, int n) {
        std::fill (dfn, dfn + MAXN, -1);
        std::fill (component, component + MAXN, -1);
        component_size = s_s = ind = 0;
        for (int i = 0; i < n; ++i) if (!~dfn[i]) dfs (e, i);
    }
};
```

5.5 Biconnected component

```
/*  Vertex biconnected component :
        Divides the edges of an undirected graph into several vertex biconnected components.
        vertex_biconnected_component::solve (const edge_list <MAXN, MAXM> &, int n) :
            component[] gives the index of the component each edge belongs to.
*/
template <int MAXN = 100000, int MAXM = 100000>
struct vertex_biconnected_component {
    int component[MAXM], component_size;
    int dfn[MAXN], low[MAXN], s[MAXN], s_s, ind;
    void dfs (const edge_list <MAXN, MAXM> &e, int u, int f) {
        dfn[u] = low[u] = ind++;
        for (int i = e.begin[u]; ~i; i = e.next[i])
            if (e.dest[i] != f && dfn[u] < dfn[e.dest[i]]) {
                s[s_s++] = i;
                if (!~dfn[u]) {
                    dfs (e, e.dest[i], u);
                    low[u] = std::min (low[u], low[e.dest[i]]);
                    if (low[e.dest[i]] >= dfn[u]) {
                        do {
                            component[s[--s_s]] = component_size;
                        } while (component[s[s_s]] != i);
                        component_size++;
                    }
                } else
                    low[u] = std::min (low[u], dfn[e.dest[i]]);
            }
    }
    void solve (const edge_list <MAXN, MAXM> &e, int n) {
        std::fill (dfn, dfn + MAXN, -1);
        std::fill (component, component + MAXM, -1);
        component_size = s_s = ind = 0;
        for (int i = 0; i < n; ++i) if (!~dfn[i]) dfs (e, i, -1);
    }
};
/*  Edge biconnected component :
        Divides the vertices of an undirected graph into several edge biconnected components.
        edge_biconnected_component::solve (const edge_list <MAXN, MAXM> &, int n) :
            component[] gives the index of the component each vertex belongs to.
*/
template <int MAXN = 100000, int MAXM = 100000>
struct edge_biconnected_component {
    int component[MAXN], component_size;
    int dfn[MAXN], low[MAXN], s[MAXN], s_s, ind;
    void dfs (const edge_list <MAXN, MAXM> &e, int u, int f) {
        dfn[u] = low[u] = ind++;
        s[s_s++] = u;
        for (int i = e.begin[u]; ~i; i = e.next[i])
            if (e.dest[i] != f) {
                if (!~dfn[e.dest[i]]) {
                    dfs (e, e.dest[i], u);
                    low[u] = std::min (low[u], low[e.dest[i]]);
                    if (low[e.dest[i]] > dfn[u]) {
                        do {
                            component[s[--s_s]] = component_size;
                        } while (s[s_s] != e.dest[i]);
                        component_size++;
                    }
                } else low[u] = std::min (low[u], dfn[e.dest[i]]);
            }
    }
    void solve (const edge_list <MAXN, MAXM> &e, int n) {
        std::fill (dfn, dfn + MAXN, -1);
        std::fill (component, component + MAXN, -1);
        component_size = s_s = ind = 0;
        for (int i = 0; i < n; ++i) if (!~dfn[i]) dfs (e, i, -1);
    }
};
```

5.6 Hopcoft-Carp

```
/*  Hopcoft-Carp algorithm :
        unweighted maximum matching for bipartition graphs with complexity O (m * n^0.5).
        struct hopcoft_carp :
            Usage : solve () for maximum matching. The matching is in matchx and matchy.
*/
```

```cpp
template <int MAXN = 100000, int MAXM = 100000>
struct hopcoft_carp {
    int n, m;
    int matchx[MAXN], matchy[MAXN], level[MAXN];
    bool dfs (edge_list <MAXN, MAXM> &e, int x) {
        for (int i = e.begin[x]; ~i; i = e.next[i]) {
            int y = e.dest[i];
            int w = matchy[y];
            if (w == -1 || (level[x] + 1 == level[w] && dfs (e, w))) {
                matchx[x] = y;
                matchy[y] = x;
                return true;
            }
        }
        level[x] = -1;
        return false;
    }
    int solve (edge_list <MAXN, MAXM> &e, int n, int m) {
        std::fill (matchx, matchx + n, -1);
        std::fill (matchy, matchy + m, -1);
        for (int answer = 0; ; ) {
            std::vector <int> queue;
            for (int i = 0; i < n; ++i) {
                if (matchx[i] == -1) {
                    level[i] = 0;
                    queue.push_back (i);
                } else {
                    level[i] = -1;
                }
            }
            for (int head = 0; head < (int) queue.size(); ++head) {
                int x = queue[head];
                for (int i = e.begin[x]; ~i; i = e.next[i]) {
                    int y = e.dest[i];
                    int w = matchy[y];
                    if (w != -1 && level[w] < 0) {
                        level[w] = level[x] + 1;
                        queue.push_back (w);
                    }
                }
            }
            int delta = 0;
            for (int i = 0; i < n; ++i)
                if (matchx[i] == -1 && dfs (e, i)) delta++;
            if (delta == 0) return answer;
            else answer += delta;
        }
    }
};
```

5.7 Kuhn-Munkres

```cpp
/*  Kuhn Munkres algorithm :
        weighted maximum matching algorithm for bipartition graphs (1-base). Complexity O (N^3).
        struct kuhn_munkres :
            Initialize : pass n as the size of both sets, w as the weight matrix.
            Usage : solve () for the maximum matching. The exact matching is in match[].
*/
template <int MAXN = 500>
struct kuhn_munkres {
    int n, w[MAXN][MAXN];
    int lx[MAXN], ly[MAXN], match[MAXN], way[MAXN], slack[MAXN];
    bool used[MAXN];
    void hungary(int x) {
        match[0] = x;
        int j0 = 0;
        std::fill (slack, slack + n + 1, INF);
        std::fill (used, used + n + 1, false);
        do {
            used[j0] = true;
            int i0 = match[j0], delta = INF, j1 = 0;
            for (int j = 1; j <= n; ++j)
                if (used[j] == false) {
                    int cur = -w[i0][j] - lx[i0] - ly[j];
                    if (cur < slack[j]) {
                        slack[j] = cur;
                        way[j] = j0;
                    }
                    if (slack[j] < delta) {
                        delta = slack[j];
                        j1 = j;
                    }
                }
            for (int j = 0; j <= n; ++j) {
                if (used[j]) {
                    lx[match[j]] += delta;
                    ly[j] -= delta;
                }
                else slack[j] -= delta;
            }
            j0 = j1;
        } while (match[j0] != 0);
        do {
            int j1 = way[j0];
            match[j0] = match[j1];
            j0 = j1;
        } while (j0);
    }
    int solve() {
        for (int i = 1; i <= n; ++i)
```

```
            match[i] = lx[i] = ly[i] = way[i] = 0;
        for (int i = 1; i <= n; ++i) hungary (i);
        int sum = 0;
        for (int i = 1; i <= n; ++i) sum += w[match[i]][i];
        return sum;
    }
};
```

5.8 Blossom algorithm

```
/*  Blossom algorithm :
        Maximum match for general graph.
        Usage : int blossom::solve (int n, const edge_list &e).
        The matching is in match[].
*/
template <int MAXN = 500, int MAXM = 250000>
struct blossom {
    int match[MAXN], d[MAXN], fa[MAXN], c1[MAXN], c2[MAXN], v[MAXN], q[MAXN];
    int *qhead, *qtail;
    struct {
        int fa[MAXN];
        void init (int n) {
            for(int i = 1; i <= n; i++)
                fa[i] = i;
        }
        int find (int x) {
            if(fa[x] != x) fa[x] = find(fa[x]);
            return fa[x];
        }
        void merge (int x, int y) {
            x = find(x);
            y = find(y);
            fa[x] = y;
        }
    } ufs;
    void solve(int x, int y) {
        if(x == y) return;
        if(d[y] == 0) {
            solve(x, fa[fa[y]]);
            match[fa[y]] = fa[fa[y]];
            match[fa[fa[y]]] = fa[y];
        }
        else if(d[y] == 1) {
            solve(match[y], c1[y]);
            solve(x, c2[y]);
            match[c1[y]] = c2[y];
            match[c2[y]] = c1[y];
        }
    }
    int lca (int x, int y, int root) {
        x = ufs.find(x); y = ufs.find(y);
        while (x != y && v[x] != 1 && v[y] != 0) {
            v[x] = 0; v[y] = 1;
            if (x != root) x = ufs.find (fa[x]);
            if (y != root) y = ufs.find (fa[y]);
        }
        if (v[y] == 0) std::swap(x, y);
        for (int i = x; i != y; i = ufs.find (fa[i])) v[i] = -1;
        v[y] = -1;
        return x;
    }
    void contract(int x, int y, int b) {
        for(int i = ufs.find(x); i != b; i = ufs.find(fa[i])) {
            ufs.merge (i, b);
            if(d[i] == 1) {
                c1[i] = x; c2[i] = y;
                *qtail++ = i;
            }
        }
    }
    bool bfs (int root, int n, const edge_list <MAXN, MAXM> &e) {
        ufs.init (n);
        std::fill (d, d + MAXN, -1);
        std::fill (v, v + MAXN, -1);
        qhead = qtail = q;
        d[root] = 0;
        *qtail++ = root;
        while(qhead < qtail) {
            for (int loc = *qhead++, i = e.begin[loc]; ~i; i = e.next[i]) {
                int dest = e.dest[i];
                if(match[dest] == -2 || ufs.find(loc) == ufs.find(dest)) continue;
                if(d[dest] == -1)
                    if(match[dest] == -1)
                    {
                        solve(root, loc);
                        match[loc] = dest;
                        match[dest] = loc;
                        return 1;
                    } else {
                        fa[dest] = loc; fa[match[dest]] = dest;
                        d[dest] = 1; d[match[dest]] = 0;
                        *qtail++ = match[dest];
                    }
                else if (d[ufs.find(dest)] == 0) {
                    int b = lca(loc, dest, root);
                    contract(loc, dest, b);
                    contract(dest, loc, b);
                }
            }
        }
        return 0;
    }
```

```
    int solve (int n, const edge_list <MAXN, MAXM> &e)
    {
        std::fill (fa, fa + n, 0);
        std::fill (c1, c1 + n, 0);
        std::fill (c2, c2 + n, 0);
        std::fill (match, match + n, -1);
        int re = 0;
        for(int i = 0; i < n; i++)
            if(match[i] == -1)
                if (bfs (i, n, e)) ++re;
                else match[i] = -2;
        return re;
    }
};
```

5.9 Weighted blossom (vfleaking ver.)

```
/*  Weighted blossom algorithm (vfleaking ver.) :
        maximum matching for general weighted graphs. Complexity O (n^3).
        Note that the base index is 1.
        struct weighted_blossom :
            Usage :
                Set n to the size of the vertices.
                Run init ().
                Set g[][].w to the weight of the edge.
                Run solve ().
                The first result is the answer, the second one is the number of matching pairs.
                Obtain the matching with match[].
*/
template <int MAXN = 500>
struct weighted_blossom {
    struct edge {
        int u, v, w;
        edge (int u = 0, int v = 0, int w = 0): u (u), v (v), w (w) {}
    };
    int n, n_x;
    edge g[MAXN * 2 + 1][MAXN * 2 + 1];
    int lab[MAXN * 2 + 1];
    int match[MAXN * 2 + 1], slack[MAXN * 2 + 1], st[MAXN * 2 + 1], pa[MAXN * 2 + 1];
    int flower_from[MAXN * 2 + 1][MAXN + 1], S[MAXN * 2 + 1], vis[MAXN * 2 + 1];
    std::vector<int> flower[MAXN * 2 + 1];
    std::queue<int> q;
    int e_delta (const edge &e) {
        return lab[e.u] + lab[e.v] - g[e.u][e.v].w * 2;
    }
    void update_slack (int u, int x) {
        if (!slack[x] || e_delta (g[u][x]) < e_delta (g[slack[x]][x]))slack[x] = u;
    }
    void set_slack (int x) {
        slack[x] = 0;
        for (int u = 1; u <= n; ++u)
            if (g[u][x].w > 0 && st[u] != x && S[st[u]] == 0)update_slack (u, x);
    }
    void q_push (int x) {
        if (x <= n)q.push (x);
        else for (size_t i = 0; i < flower[x].size(); i++)q_push (flower[x][i]);
    }
    void set_st (int x, int b) {
        st[x] = b;
        if (x > n)for (size_t i = 0; i < flower[x].size(); ++i)
                set_st (flower[x][i], b);
    }
    int get_pr (int b, int xr) {
        int pr = find (flower[b].begin(), flower[b].end(), xr) - flower[b].begin();
        if (pr % 2 == 1) {
            reverse (flower[b].begin() + 1, flower[b].end());
            return (int)flower[b].size() - pr;
        } else return pr;
    }
    void set_match (int u, int v) {
        match[u] = g[u][v].v;
        if (u > n) {
            edge e = g[u][v];
            int xr = flower_from[u][e.u], pr = get_pr (u, xr);
            for (int i = 0; i < pr; ++i)set_match (flower[u][i], flower[u][i ^ 1]);
            set_match (xr, v);
            rotate (flower[u].begin(), flower[u].begin() + pr, flower[u].end());
        }
    }
    void augment (int u, int v) {
        for (;;) {
            int xnv = st[match[u]];
            set_match (u, v);
            if (!xnv)return;
            set_match (xnv, st[pa[xnv]]);
            u = st[pa[xnv]], v = xnv;
        }
    }
    int get_lca (int u, int v) {
        static int t = 0;
        for (++t; u || v; std::swap (u, v)) {
            if (u == 0)continue;
            if (vis[u] == t)return u;
            vis[u] = t;
            u = st[match[u]];
            if (u)u = st[pa[u]];
        }
        return 0;
    }
    void add_blossom (int u, int lca, int v) {
        int b = n + 1;
```

```
    while (b <= n_x && st[b])++b;
    if (b > n_x)++n_x;
    lab[b] = 0, S[b] = 0;
    match[b] = match[lca];
    flower[b].clear();
    flower[b].push_back (lca);
    for (int x = u, y; x != lca; x = st[pa[y]])
        flower[b].push_back (x), flower[b].push_back (y = st[match[x]]), q_push (y);
    reverse (flower[b].begin() + 1, flower[b].end());
    for (int x = v, y; x != lca; x = st[pa[y]])
        flower[b].push_back (x), flower[b].push_back (y = st[match[x]]), q_push (y);
    set_st (b, b);
    for (int x = 1; x <= n_x; ++x)g[b][x].w = g[x][b].w = 0;
    for (int x = 1; x <= n; ++x)flower_from[b][x] = 0;
    for (size_t i = 0; i < flower[b].size(); ++i) {
        int xs = flower[b][i];
        for (int x = 1; x <= n_x; ++x)
            if (g[b][x].w == 0 || e_delta (g[xs][x]) < e_delta (g[b][x]))
                g[b][x] = g[xs][x], g[x][b] = g[x][xs];
        for (int x = 1; x <= n; ++x)
            if (flower_from[xs][x])flower_from[b][x] = xs;
    }
    set_slack (b);
}
void expand_blossom (int b) {
    for (size_t i = 0; i < flower[b].size(); ++i)
        set_st (flower[b][i], flower[b][i]);
    int xr = flower_from[b][g[b][pa[b]].u], pr = get_pr (b, xr);
    for (int i = 0; i < pr; i += 2) {
        int xs = flower[b][i], xns = flower[b][i + 1];
        pa[xs] = g[xns][xs].u;
        S[xs] = 1, S[xns] = 0;
        slack[xs] = 0, set_slack (xns);
        q_push (xns);
    }
    S[xr] = 1, pa[xr] = pa[b];
    for (size_t i = pr + 1; i < flower[b].size(); ++i) {
        int xs = flower[b][i];
        S[xs] = -1, set_slack (xs);
    }
    st[b] = 0;
}
bool on_found_edge (const edge &e) {
    int u = st[e.u], v = st[e.v];
    if (S[v] == -1) {
        pa[v] = e.u, S[v] = 1;
        int nu = st[match[v]];
        slack[v] = slack[nu] = 0;
        S[nu] = 0, q_push (nu);
    } else if (S[v] == 0) {
        int lca = get_lca (u, v);
        if (!lca)return augment (u, v), augment (v, u), true;
        else add_blossom (u, lca, v);
    }
    return false;
}
bool matching() {
    std::fill (S + 1, S + 1 + n_x, -1);
    std::fill (slack + 1, slack + 1 + n_x, -1);
    q = std::queue<int>();
    for (int x = 1; x <= n_x; ++x)
        if (st[x] == x && !match[x])pa[x] = 0, S[x] = 0, q_push (x);
    if (q.empty())return false;
    for (;;) {
        while (q.size()) {
            int u = q.front();
            q.pop();
            if (S[st[u]] == 1) continue;
            for (int v = 1; v <= n; ++v)
                if (g[u][v].w > 0 && st[u] != st[v]) {
                    if (e_delta (g[u][v]) == 0) {
                        if (on_found_edge (g[u][v])) return true;
                    } else update_slack (u, st[v]);
                }
        }
        int d = INF;
        for (int b = n + 1; b <= n_x; ++b)
            if (st[b] == b && S[b] == 1)d = std::min (d, lab[b] / 2);
        for (int x = 1; x <= n_x; ++x)
            if (st[x] == x && slack[x]) {
                if (S[x] == -1)d = std::min (d, e_delta (g[slack[x]][x]));
                else if (S[x] == 0)d = std::min (d, e_delta (g[slack[x]][x]) / 2);
            }
        for (int u = 1; u <= n; ++u) {
            if (S[st[u]] == 0) {
                if (lab[u] <= d)return 0;
                lab[u] -= d;
            } else if (S[st[u]] == 1)lab[u] += d;
        }
        for (int b = n + 1; b <= n_x; ++b)
            if (st[b] == b) {
                if (S[st[b]] == 0)lab[b] += d * 2;
                else if (S[st[b]] == 1)lab[b] -= d * 2;
            }
        q = std::queue<int>();
        for (int x = 1; x <= n_x; ++x)
            if (st[x] == x && slack[x] && st[slack[x]] != x && e_delta (g[slack[x]][x]) == 0)
                if (on_found_edge (g[slack[x]][x]))return true;
        for (int b = n + 1; b <= n_x; ++b)
            if (st[b] == b && S[b] == 1 && lab[b] == 0)expand_blossom (b);
    }
    return false;
}
std::pair <long long, int> solve () {
    std::fill (match + 1, match + n + 1, 0);
```

```
        n_x = n;
        int n_matches = 0;
        long long tot_weight = 0;
        for (int u = 0; u <= n; ++u)st[u] = u, flower[u].clear();
        int w_max = 0;
        for (int u = 1; u <= n; ++u)
            for (int v = 1; v <= n; ++v) {
                flower_from[u][v] = (u == v ? u : 0);
                w_max = std::max (w_max, g[u][v].w);
            }
        for (int u = 1; u <= n; ++u)lab[u] = w_max;
        while (matching())++n_matches;
        for (int u = 1; u <= n; ++u)
            if (match[u] && match[u] < u)
                tot_weight += g[u][match[u]].w;
        return std::make_pair (tot_weight, n_matches);
    }
    void init () {
        for (int u = 1; u <= n; ++u)
            for (int v = 1; v <= n; ++v)
                g[u][v] = edge (u, v, 0);
    }
};
```

5.10 Maximum flow

```
/*  Sparse graph maximum flow :
        int isap::solve (flow_edge_list &e, int n, int s, int t) :
            e : edge list.
            n : vertex size.
            s : source.
            t : sink.
*/
template <int MAXN = 1000, int MAXM = 100000>
struct isap {
    int pre[MAXN], d[MAXN], gap[MAXN], cur[MAXN];
    int solve (flow_edge_list <MAXN, MAXM> &e, int n, int s, int t) {
        std::fill (pre, pre + n + 1, 0);
        std::fill (d, d + n + 1, 0);
        std::fill (gap, gap + n + 1, 0);
        for (int i = 0; i < n; i++) cur[i] = e.begin[i];
        gap[0] = n;
        int u = pre[s] = s, v, maxflow = 0;
        while (d[s] < n) {
            v = n;
            for (int i = cur[u]; ~i; i = e.next[i])
                if (e.flow[i] && d[u] == d[e.dest[i]] + 1) {
                    v = e.dest[i];
                    cur[u] = i;
                    break;
                }
            if (v < n) {
                pre[v] = u;
                u = v;
                if (v == t) {
                    int dflow = INF, p = t;
                    u = s;
                    while (p != s) {
                        p = pre[p];
                        dflow = std::min (dflow, e.flow[cur[p]]);
                    }
                    maxflow += dflow;
                    p = t;
                    while (p != s) {
                        p = pre[p];
                        e.flow[cur[p]] -= dflow;
                        e.flow[e.inv[cur[p]]] += dflow;
                    }
                }
            } else {
                int mindist = n + 1;
                for (int i = e.begin[u]; ~i; i = e.next[i])
                    if (e.flow[i] && mindist > d[e.dest[i]]) {
                        mindist = d[e.dest[i]];
                        cur[u] = i;
                    }
                if (!--gap[d[u]]) return maxflow;
                gap[d[u] = mindist + 1]++;
                u = pre[u];
            }
        }
        return maxflow;
    }
};
/*  Dense graph maximum flow :
        int dinic::solve (flow_edge_list &e, int n, int s, int t) :
            e : edge list.
            n : vertex size.
            s : source.
            t : sink.
*/
template <int MAXN = 1000, int MAXM = 100000>
struct dinic {
    int n, s, t;
    int d[MAXN], w[MAXN], q[MAXN];
    int bfs (flow_edge_list <MAXN, MAXM> &e) {
        for (int i = 0; i < n; i ++) d[i] = -1;
        int l, r;
        q[l = r = 0] = s, d[s] = 0;
        for (; l <= r; l ++)
            for (int k = e.begin[q[l]]; k > -1; k = e.next[k])
                if (d[e.dest[k]] == -1 && e.flow[k] > 0) d[e.dest[k]] = d[q[l]] + 1, q[++r] = e.dest[k];
```

```
        return d[t] > -1 ? 1 : 0;
    }
    int dfs (flow_edge_list <MAXN, MAXM> &e, int u, int ext) {
        if (u == t) return ext;
        int k = w[u], ret = 0;
        for (; k > -1; k = e.next[k], w[u] = k) {
            if (ext == 0) break;
            if (d[e.dest[k]] == d[u] + 1 && e.flow[k] > 0) {
                int flow = dfs (e, e.dest[k], std::min (e.flow[k], ext));
                if (flow > 0) {
                    e.flow[k] -= flow, e.flow[e.inv[k]] += flow;
                    ret += flow, ext -= flow;
                }
            }
        }
        if (k == -1) d[u] = -1;
        return ret;
    }
    void solve (flow_edge_list <MAXN, MAXM> &e, int n, int s, int t) {
        dinic::n = n; dinic::s = s; dinic::t = t;
        while (bfs (e)) {
            for (int i = 0; i < n; i ++) w[i] = e.begin[i];
            dfs (e, s, INF);
        }
    }
};
```

5.11 Minimum cost flow

```
/*  Sparse graph minimum cost flow :
        std::pair <int, int> minimum_cost_flow::solve (cost_flow_edge_list &e,
                                                        int n, int s, int t) :
            e : edge list.
            n : vertex size.
            s : source.
            t : sink.
            returns the flow and the cost respectively.
*/
template <int MAXN = 1000, int MAXM = 100000>
struct minimum_cost_flow {
    int n, source, target;
    int prev[MAXN];
    int dist[MAXN], occur[MAXN];
    bool augment (cost_flow_edge_list <MAXN, MAXM> &e) {
        std::vector <int> queue;
        std::fill (dist, dist + n, INF);
        std::fill (occur, occur + n, 0);
        dist[source] = 0;
        occur[source] = true;
        queue.push_back (source);
        for (int head = 0; head < (int)queue.size(); ++head) {
            int x = queue[head];
            for (int i = e.begin[x]; ~i; i = e.next[i]) {
                int y = e.dest[i];
                if (e.flow[i] && dist[y] > dist[x] + e.cost[i]) {
                    dist[y] = dist[x] + e.cost[i];
                    prev[y] = i;
                    if (!occur[y]) {
                        occur[y] = true;
                        queue.push_back (y);
                    }
                }
            }
            occur[x] = false;
        }
        return dist[target] < INF;
    }
    std::pair <int, int> solve (cost_flow_edge_list <MAXN, MAXM> &e, int n, int s, int t) {
        minimum_cost_flow::n = n;
        source = s; target = t;
        std::pair <int, int> answer = std::make_pair (0, 0);
        while (augment (e)) {
            int number = INF;
            for (int i = target; i != source; i = e.dest[e.inv[prev[i]]]) {
                number = std::min (number, e.flow[prev[i]]);
            }
            answer.first += number;
            for (int i = target; i != source; i = e.dest[e.inv[prev[i]]]) {
                e.flow[prev[i]] -= number;
                e.flow[e.inv[prev[i]]] += number;
                answer.second += number * e.cost[prev[i]];
            }
        }
        return answer;
    }
};
/*  Dense graph minimum cost flow :
        std::pair <int, int> zkw_flow::solve (cost_flow_edge_list &e,
                                              int n, int s, int t) :
            e : edge list.
            n : vertex size.
            s : source.
            t : sink.
            returns the flow and the cost respectively.
*/
template <int MAXN = 1000, int MAXM = 100000>
struct zkw_flow {
    int n, s, t, totFlow, totCost;
    int dis[MAXN], slack[MAXN], visit[MAXN];
    int modlable() {
        int delta = INF;
```

```
        for (int i = 0; i < n; i++) {
            if (!visit[i] && slack[i] < delta) delta = slack[i];
            slack[i] = INF;
        }
        if (delta == INF) return 1;
        for (int i = 0; i < n; i++) if (visit[i]) dis[i] += delta;
        return 0;
    }
    int dfs (cost_flow_edge_list <MAXN, MAXM> &e, int x, int flow) {
        if (x == t) {
            totFlow += flow;
            totCost += flow * (dis[s] - dis[t]);
            return flow;
        }
        visit[x] = 1;
        int left = flow;
        for (int i = e.begin[x]; ~i; i = e.next[i])
            if (e.flow[i] > 0 && !visit[e.dest[i]]) {
                int y = e.dest[i];
                if (dis[y] + e.cost[i] == dis[x]) {
                    int delta = dfs (e, y, std::min (left, e.flow[i]));
                    e.flow[i] -= delta;
                    e.flow[e.inv[i]] += delta;
                    left -= delta;
                    if (!left) { visit[x] = false; return flow; }
                } else
                    slack[y] = std::min (slack[y], dis[y] + e.cost[i] - dis[x]);
            }
        return flow - left;
    }
    std::pair <int, int> solve (cost_flow_edge_list <MAXN, MAXM> &e, int n, int s, int t) {
        zkw_flow::n = n; zkw_flow::s = s; zkw_flow::t = t;
        totFlow = 0; totCost = 0;
        std::fill (dis + 1, dis + t + 1, 0);
        do {
            do {
                std::fill (visit + 1, visit + t + 1, 0);
            } while (dfs (e, s, INF));
        } while (!modlable ());
        return std::make_pair (totFlow, totCost);
    }
};
```

5.12 Dominator tree

```
/*  Dominator tree :
        void dominator_tree::solve (int s, int n, const edge_list <MAXN, MAXM> &succ) :
            solves for the immediate dominator (idom[]) of each node,
                idom[x] will be x if x does not have a dominator,
                and will be -1 if x is not reachable from s.
*/
template <int MAXN = 100000, int MAXM = 100000>
struct dominator_tree {
    int dfn[MAXN], sdom[MAXN], idom[MAXN], id[MAXN], f[MAXN], fa[MAXN], smin[MAXN], stamp;
    void predfs (int x, const edge_list <MAXN, MAXM> &succ) {
        id[dfn[x] = stamp++] = x;
        for (int i = succ.begin[x]; ~i; i = succ.next[i]) {
            int y = succ.dest[i];
            if (dfn[y] < 0) {
                f[y] = x;
                predfs (y, succ);
            }
        }
    }
    int getfa (int x) {
        if (fa[x] == x) return x;
        int ret = getfa (fa[x]);
        if (dfn[sdom[smin[fa[x]]]] < dfn[sdom[smin[x]]])
            smin[x] = smin[fa[x]];
        return fa[x] = ret;
    }
    void solve (int s, int n, const edge_list <MAXN, MAXM> &succ) {
        std::fill (dfn, dfn + n, -1);
        std::fill (idom, idom + n, -1);
        static edge_list <MAXN, MAXM> pred, tmp;
        pred.clear (n);
        for (int i = 0; i < n; ++i)
            for (int j = succ.begin[i]; ~j; j = succ.next[j])
                pred.add_edge (succ.dest[j], i);
        stamp = 0; tmp.clear (n); predfs (s, succ);
        for (int i = 0; i < stamp; ++i)
            fa[id[i]] = smin[id[i]] = id[i];
        for (int o = stamp - 1; o >= 0; --o) {
            int x = id[o];
            if (o) {
                sdom[x] = f[x];
                for (int i = pred.begin[x]; ~i; i = pred.next[i]) {
                    int p = pred.dest[i];
                    if (dfn[p] < 0) continue;
                    if (dfn[p] > dfn[x]) {
                        getfa (p);
                        p = sdom[smin[p]];
                    }
                    if (dfn[sdom[x]] > dfn[p])
                        sdom[x] = p;
                }
                tmp.add_edge (sdom[x], x);
            }
            while (~tmp.begin[x]) {
                int y = tmp.dest[tmp.begin[x]];
                tmp.begin[x] = tmp.next[tmp.begin[x]];
```

```
                getfa (y);
                if (x != sdom[smin[y]])
                    idom[y] = smin[y];
                else
                    idom[y] = x;
            }
            for (int i = succ.begin[x]; ~i; i = succ.next[i])
                if (f[succ.dest[i]] == x)
                    fa[succ.dest[i]] = x;
        }
        idom[s] = s;
        for (int i = 1; i < stamp; ++i) {
            int x = id[i];
            if (idom[x] != sdom[x])
                idom[x] = idom[idom[x]];
        }
    }
};
```

5.13 Stoer Wagner

```
/*  Stoer Wagner algorithm :
        Finds the minimum cut of an undirected graph.
        Usage :
            input : n, edge[][] : the size and weights of the graph.
            int stoer_wagner::solve () : returns the minimum cut.
*/
template <int MAXN = 500>
struct stoer_wagner {
    int n, edge[MAXN][MAXN];
    int dist[MAXN];
    bool vis[MAXN], bin[MAXN];
    stoer_wagner () {
        memset (edge, 0, sizeof (edge));
        memset (bin, false, sizeof (bin));
    }
    int contract (int &s, int &t)  {
        memset (dist, 0, sizeof (dist));
        memset (vis, false, sizeof (vis));
        int i, j, k, mincut, maxc;
        for (i = 1; i <= n; i++) {
            k = -1; maxc = -1;
            for (j = 1; j <= n; j++)
                if (!bin[j] && !vis[j] && dist[j] > maxc) {
                    k = j;  maxc = dist[j];
                }
            if (k == -1) return mincut;
            s = t; t = k;
            mincut = maxc;
            vis[k] = true;
            for (j = 1; j <= n; j++)
                if (!bin[j] && !vis[j])
                    dist[j] += edge[k][j];
        }
        return mincut;
    }
    int solve () {
        int mincut, i, j, s, t, ans;
        for (mincut = INF, i = 1; i < n; i++) {
            ans = contract ( s, t );
            bin[t] = true;
            if (mincut > ans) mincut = ans;
            if (mincut == 0) return 0;
            for (j = 1; j <= n; j++)
                if (!bin[j])
                    edge[s][j] = (edge[j][s] += edge[j][t]);
        }
        return mincut;
    }
};
```

Chapter 6

Trial of string

6.1 KMP

```
/*  KMP algorithm :
        void kmp::build (const std::string & str) :
            initializes and builds the failure array. Complexity O (n).
        int kmp::find (const std::string &str) :
            finds the first occurence of match in str. Complexity O (n).
        Note : match is cylic when L % (L - 1 - fail[L - 1]) == 0 &&
            L / (L - 1 - fail[L - 1]) > 1, where L = match.size ().
*/
template <int MAXN = 1000000>
struct kmp {
    std::string match;
    int fail[MAXN];
    void build (const std::string &str) {
        match = str; fail[0] = -1;
        for (int i = 1; i < (int) str.size (); ++i) {
            int j = fail[i - 1];
            while (~j && str[i] != str[j + 1]) j = fail[j];
            fail[i] = str[i] == str[j + 1] ? j + 1 : -1;
        }
    }
    int find (const std::string &str) {
        int i, j;
        for (i = 0, j = -1; i < (int) str.size () && j < match.size () - 1;
                j += str[i] == match[j + 1], ++i) {
            while (~j && str[i] != match[j + 1]) j = fail[j];
        }
        if (j == match.size () - 1) return i - match.size ();
        return str.size ();
    }
};
```

6.2 Suffix array

```
/*  Suffix Array :
        Suffix array implementation.
        void suffix_array::solve (int *a, int n) :
            sa[i] : the beginning position of the ith smallest suffix.
            rk[i] : the rank of the suffix beginning at position i.
            height[i] : the longest common prefix of sa[i] and sa[i - 1].
*/
template <int MAXN = 1000000, int MAXC = 26>
struct suffix_array {
    int rk[MAXN], height[MAXN], sa[MAXN];
    int cmp (int *x, int a, int b, int d) {
        return x[a] == x[b] && x[a + d] == x[b + d];
    }
    void doubling (int *a, int n) {
        static int sRank[MAXN], tmpA[MAXN], tmpB[MAXN];
        int m = MAXC;
        int *x = tmpA, *y = tmpB;
        for (int i = 0; i < m; ++i) sRank[i] = 0;
        for (int i = 0; i < n; ++i) ++sRank[x[i] = a[i]];
        for (int i = 1; i < m; ++i) sRank[i] += sRank[i - 1];
        for (int i = n - 1; i >= 0; --i) sa[--sRank[x[i]]] = i;
        for (int d = 1, p = 0; p < n; m = p, d <<= 1) {
            p = 0; for (int i = n - d; i < n; ++i) y[p++] = i;
            for (int i = 0; i < n; ++i) if (sa[i] >= d) y[p++] = sa[i] - d;
            for (int i = 0; i < m; ++i) sRank[i] = 0;
            for (int i = 0; i < n; ++i) ++sRank[x[i]];
            for (int i = 1; i < m; ++i) sRank[i] += sRank[i - 1];
            for (int i = n - 1; i >= 0; --i) sa[--sRank[x[y[i]]]] = y[i];
            std::swap (x, y); x[sa[0]] = 0; p = 1;
            y[n] = -1;
            for (int i = 1; i < n; ++i)
                x[sa[i]] = cmp (y, sa[i], sa[i - 1], d) ? p - 1 : p++;
        }
    }
    void solve (int *a, int n) {
        a[n] = -1; doubling (a, n);
        for (int i = 0; i < n; ++i) rk[sa[i]] = i;
```

```
        int cur = 0;
        for (int i = 0; i < n; ++i)
            if (rk[i]) {
                if (cur) cur--;
                for (; a[i + cur] == a[sa[rk[i] - 1] + cur]; ++cur);
                height[rk[i]] = cur;
            }
    }
};
```

6.3 Suffix automaton

```
/*  Suffix automaton :
        void suffix_automaton::init () :
            initializes the automaton with an empty string.
        void suffix_automaton::extend (int token) :
            extends the string with token. Complexity O (1).
        head : the first state.
        tail : the last state.
            Terminating states can be reached via visiting the ancestors of tail.
        state::len : the longest length of the string in the state.
        state::right - 1 : the first place where the state can be reached.
        state::parent : the parent link.
        state::dest : the automaton link.
*/
template <int MAXN = 1000000, int MAXC = 26>
struct suffix_automaton {
    struct state {
        int len, right;
        state *parent, *dest[MAXC];
        state (int len = 0, int right = 0) : len (len), right (right), parent (NULL) {
            memset (dest, 0, sizeof (dest));
        }
    } node_pool[MAXN * 2], *tot_node, *null = new state();
    state *head, *tail;
    void extend (int token) {
        state *p = tail;
        state *np = tail -> dest[token] ? null : new (tot_node++) state (tail -> len + 1, tail -> len + 1);
        while (p && !p -> dest[token])
            p -> dest[token] = np, p = p -> parent;
        if (!p) np -> parent = head;
        else {
            state *q = p -> dest[token];
            if (p -> len + 1 == q -> len) {
                np -> parent = q;
            } else {
                state *nq = new (tot_node++) state (*q);
                nq -> len = p -> len + 1;
                np -> parent = q -> parent = nq;
                while (p && p -> dest[token] == q) {
                    p -> dest[token] = nq, p = p -> parent;
                }
            }
        }
        tail = np == null ? np -> parent : np;
    }
    void init () {
        tot_node = node_pool;
        head = tail = new (tot_node++) state();
    }
    suffix_automaton () {
        init ();
    }
};
```

6.4 Palindromic tree

```
/*  Palindromic tree :
        void palindromic_tree::init () : initializes the tree.
        bool palindromic_tree::extend (int) : extends the string with token.
            returns whether the tree has generated a new node.
            Complexity O (log MAXC).
        odd, even : the root of two trees.
        last : the node representing the last char.
        node::len : the palindromic string length of the node.
*/
template <int MAXN = 1000000, int MAXC = 26>
struct palindromic_tree {
    struct node {
        node *child[MAXC], *fail;
        int len;
        node (int len) : fail (NULL), len (len) {
            memset (child, NULL, sizeof (child));
        }
    } node_pool[MAXN * 2], *tot_node;
    int size, text[MAXN];
    node *odd, *even, *last;
    node *match (node *now) {
        for (; text[size - now -> len - 1] != text[size]; now = now -> fail);
        return now;
    }
    bool extend (int token) {
        text[++size] = token;
        node *now = match (last);
        if (now -> child[token])
            return last = now -> child[token], false;
        last = now -> child[token] = new (tot_node++) node (now -> len + 2);
```

```
        if (now == odd) last -> fail = even;
        else {
            now = match (now -> fail);
            last -> fail = now -> child[token];
        }
        return true;
    }
    void init() {
        text[size = 0] = -1;
        tot_node = node_pool;
        last = even = new (tot_node++) node (0); odd = new (tot_node++) node (-1);
        even -> fail = odd;
    }
    palindromic_tree () {
        init ();
    }
};
```

Chapter 7

Reference

7.1 Vimrc

```
set ru nu ts=4 sts=4 sw=4 si sm hls is ar bs=2 mouse=a
syntax on
nm <F3> :vsplit %<.in <CR>
nm <F4> :!gedit % <CR>
au BufEnter *.cpp set cin
au BufEnter *.cpp nm <F5> :!time ./%< <CR>|nm <F7> :!gdb ./%< <CR>|nm <F8> :!time ./%< < %<.in <CR>|
\nm <F9> :!g++ % -o %< -g -std=c++11 -O2 && size %< <CR>
au BufEnter *.java nm <F5> :!time java %< <CR>|nm <F8> :!time java %< < %<.in <CR>|nm <F9> :!javac % <CR>
```

7.2 Java reference

```java
/*  Java reference :
        References on Java IO, structures, etc.
*/
import java.io.*;
import java.lang.*;
import java.math.*;
import java.util.*;
/*  Regular usage:
        Slower IO :
            Scanner in = new Scanner (System.in);
            Scanner in = new Scanner (new BufferedInputStream (System.in));
            Input :
                in.nextInt () / in.nextBigInteger () / in.nextBigDecimal () / in.nextDouble ()
                in.nextLine () / in.hasNext ()
            Output :
                System.out.print (...);
                System.out.println (...);
                System.out.printf (...);
        Faster IO :
            Shown below.
        BigInteger :
            BigInteger.valueOf (int) : convert to BigInteger.
            abs / negate () / max / min / add / subtract / multiply /
                divide / remainder (BigInteger) : BigInteger algebraic.
            gcd (BigInteger) / modInverse (BigInteger mod) /
                modPow (BigInteger ex, BigInteger mod) / pow (int ex) : Number Theory.
            not () / and / or / xor (BigInteger) / shiftLeft / shiftRight (int) : Bit operation.
            compareTo (BigInteger) : comparation.
            intValue () / longValue () / toString (int radix) : converts to other types.
            isProbablePrime (int certainty) / nextProbablePrime () : checks primitive.
        BigDecimal :
            consists of a BigInteger value and a scale.
            The scale is the number of digits to the right of the decimal point.
            divide (BigDecimal) : exact divide.
            divide (BigDecimal, int scale, RoundingMode roundingMode) :
                divide with roundingMode, which may be:
                    CEILING / DOWN / FLOOR / HALF_DOWN / HALF_EVEN / HALF_UP / UNNECESSARY / UP.
            BigDecimal setScale (int newScale, RoundingMode roundingMode) :
                returns a BigDecimal with newScale.
            doubleValue () / toPlainString () : converts to other types.
        Arrays :
            Arrays.sort (T [] a);
            Arrays.sort (T [] a, int fromIndex, int toIndex);
            Arrays.sort (T [] a, int fromIndex, int toIndex, Comperator <? super T> comperator);
        LinkedList <E> :
            addFirst / addLast (E) / getFirst / getLast / removeFirst / removeLast () :
                deque implementation.
            clear () / add (int, E) / remove (int) : clear, add & remove.
            size () / contains / removeFirstOccurrence / removeLastOccurrence (E) :
                deque methods.
            ListIterator <E> listIterator (int index) : returns an iterator :
                E next / previous () : accesses and iterates.
                hasNext / hasPrevious () : checks availablity.
                nextIndex / previousIndex () : returns the index of a subsequent call.
                add / set (E) / remove () : changes element.
        PriorityQueue <E> (int initcap, Comparator <? super E> comparator) :
            add (E) / clear () / iterator () / peek () / poll () / size () :
                priority queue implementations.
        TreeMap <K, V> (Comparator <? super K> comparator) :
            Map.Entry <K, V> ceilingEntry / floorEntry / higherEntry / lowerEntry (K):
                getKey / getValue () / setValue (V) : entries.
```

```
            clear () / put (K, V) / get (K) / remove (K) : basic operation.
            size () : size.
        StringBuilder :
            Mutable string.
            StringBuilder (string) : generates a builder.
            append (int, string, ...) / insert (int offset, ...) : adds objects.
            charAt (int) / setCharAt (int, char) : accesses a char.
            delete (int, int) : removes a substring.
            reverse () : reverses itself.
            length () : returns the length.
            toString () : converts to string.
        String :
            Immutable string.
            String.format (String, ...) : formats a string. i.e. sprintf.
            toLowerCase / toUpperCase () : changes the case of letters.
*/
/* Examples on Comparator :
public class Main {
    public static class Point {
        public int x;
        public int y;
        public Point () {
            x = 0;
            y = 0;
        }
        public Point (int xx, int yy) {
            x = xx;
            y = yy;
        }
    };
    public static class Cmp implements Comparator <Point> {
        public int compare (Point a, Point b) {
            if (a.x < b.x) return -1;
            if (a.x == b.x) {
                if (a.y < b.y) return -1;
                if (a.y == b.y) return 0;
            }
            return 1;
        }
    };
    public static void main (String [] args) {
        Cmp c = new Cmp ();
        TreeMap <Point, Point> t = new TreeMap <Point, Point> (c);
        return;
    }
};
*/
/*  Another way to implement is to use Comparable.
    However, equalTo and hashCode must be rewritten.
    Otherwise, containers may fail.
    Example :
    public static class Point implements Comparable <Point> {
        public int x;
        public int y;
        public Point () {
            x = 0;
            y = 0;
        }
        public Point (int xx, int yy) {
            x = xx;
            y = yy;
        }
        public int compareTo (Point p) {
            if (x < p.x) return -1;
            if (x == p.x) {
                if (y < p.y) return -1;
                if (y == p.y) return 0;
            }
            return 1;
        }
        public boolean equalTo (Point p) {
            return (x == p.x && y == p.y);
        }
        public int hashCode () {
            return x + y;
        }
    };
*/
//Faster IO :
public class Main {
    static class InputReader {
        public BufferedReader reader;
        public StringTokenizer tokenizer;
        public InputReader (InputStream stream) {
            reader = new BufferedReader (new InputStreamReader (stream), 32768);
            tokenizer = null;
        }
        public String next() {
            while (tokenizer == null || !tokenizer.hasMoreTokens()) {
                try {
                    String line = reader.readLine();
                    tokenizer = new StringTokenizer (line);
                } catch (IOException e) {
                    throw new RuntimeException (e);
                }
            }
            return tokenizer.nextToken();
        }
        public BigInteger nextBigInteger() {
            return new BigInteger (next (), 10);    //  customize the radix here.
        }
        public int nextInt() {
            return Integer.parseInt (next());
```

```
        }
        public double nextDouble() {
            return Double.parseDouble(next());
        }
    }
    public static void main(String[] args) {
        InputReader in = new InputReader(System.in);
        // Put your code here.
    }
}
```

7.3 Operator precedence

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right

2	`a++ a--`
	`type()`
	`type{}`
	`a()`
	`a[]`
	`. ->`
		Suffix/postfix increment and decrement
		Functional cast
		Function call
		Subscript
		Member access	

3	`++a --a`
	`+a -a`
	`! ~`
	`(type)`
	`*a`
	`&a`
	`sizeof`
	`new new[]`
	`delete`
	`delete[]`
		Prefix increment and decrement
		Unary plus and minus
		Logical NOT and bitwise NOT
		C-style cast
		Indirection (dereference)
		Address-of
		Size-of
		Dynamic memory allocation
		Dynamic memory deallocation
			Right-to-left

4	`.* ->*`	Pointer-to-member	Left-to-right
5	`a*b a/b` `a%b`	Multiplication, division, and remainder	
6	`a+b a-b`	Addition and subtraction	
7	`<< >>`	Bitwise left shift and right shift	
8	`< <=` `> >=`	For relational operators $<$ and $\leq$ respectively For relational operators $>$ and $\geq$ respectively	
9	`== !=`	For relational operators $=$ and $\neq$ respectively	
10	`a&b`	Bitwise AND	
11	`^`	Bitwise XOR (exclusive or)	
12	`|`	Bitwise OR (inclusive or)	
13	`&&`	Logical AND	
14	`||`	Logical OR	

15	`a?b:c`
	`throw`
	`=`
	`+= -= *=`
	`/= %=`
	`<<= >>=`
	`&= ^= |=`
		Ternary conditional
		throw operator
		Direct assignment
		Compound assignment by arithmetic operation
		Compound assignment by bitwise shift
		Compound assignment by bitwise AND, XOR, and OR
			Right-to-left

16	`,`	Comma	Left-to-right

7.4 Hacks

7.4.1 Ultra fast functions

```
// Ultra fast functions :
__inline void make_min(int &a, int &b) {
    asm (
        "cmpl␣%2,␣%0\n\t"
        "jle␣DONE\n\t"
        "movl␣%2,␣%0\n\t"
        "DONE:"
        : "=r" (a)
        : "0" (a), "r" (b)
    );
}
__inline void make_max(int &a, int &b) {
    asm (
```

```
        "cmpl %2, %0\n\t"
        "jge DONE\n\t"
        "movl %2, %0\n\t"
        "DONE:"
        : "=r" (a)
        : "0" (a), "r" (b)
    );
}
__inline int cmp (int a) {
    return (a >> 31) + (-a >> 31 & 1);
}
__inline int abs (int x) {
    int y = x >> 31;
    return (x + y) ^ y;
}
__inline int mul_mod (int a, int b) {
    int ret;
    asm (
        "mull %%ebx\n\t"
        "divl %%ecx\n\t"
        : "=d" (ret)
        : "a" (a), "b" (b), "c" (MO)
    );
    return ret;
}
__inline int next_uint () {
    const int SIZE = 110000; static char buf[SIZE]; static int p = SIZE;
    register int ans = 0, f = 1;
    while ((p < SIZE || fread (buf, 1, SIZE, stdin) && (p = 0, 1))
        && (isdigit (buf[p]) && (ans = ans * 10 + buf[p] - '0', f = 0, 1) || f)) ++p;
    return ans;
}
__inline int next_int () {
    const int SIZE = 110000; static char buf[SIZE]; static int p = SIZE;
    register int ans = 0, f = 1, sgn = 1;
    while ((p < SIZE || fread (buf, 1, SIZE, stdin) && (p = 0, 1)) &&
        (isdigit (buf[p]) && (ans = ans * 10 + buf[p] - '0', f = 0, 1) ||
        f && (buf[p] == '-' && (sgn = 0), 1))) ++p;
    return sgn ? ans : -ans;
}
```

7.4.2 Formating long long in scanf & printf

```
#ifdef WIN32
    #define LL "%I64d"
#else
    #define LL "%lld"
#endif
```

7.4.3 Optimizing

```
#define __ __attribute__ ((optimize ("-O3")))
#define _ __ __inline __attribute__ ((__gnu_inline__, __always_inline__, __artificial__))
#define NDEBUG
```

7.4.4 Larger stack

7.4.4.1 C++

```
#pragma comment(linker, "/STACK:36777216")
```

7.4.4.2 G++

```
int __size__ = 256 << 20;   // 256MB
char *__p__ = (char*)malloc(__size__) + __size__;
__asm__ ("movl %0, %%esp\n" :: "r"(__p__));
```

7.5 Math reference

7.5.1 Catalan number

For

$$\begin{aligned} f(0) &= 1 \\ f(1) &= 1 \\ f(n) &= f(n-1)f(0) + f(n-2)f(1) + ... + f(1)f(n-2) + f(0)f(n-1) \end{aligned}$$

We have $f(n) = \frac{(2n)!}{n!(n+1)!}$.

7.5.2 Dynamic programming optimization

7.5.2.1 Convex hull optimization

Generally, in dynamic programming with recurrence

$$f(i) = \min_{k<i}\{a[i]b(j) + c(j)\}$$

all decisions k can be treated as a set of segments on a convex hull. By applying Graham's scanning, it is possible to maintain such hull in a monotone queue or a `std::tuple <slope, intercept, x_min>`. Hence, $k(i)$ can be obtained by performing a binary search in the hull.

7.5.2.2 Divide & conquer optimization

For recurrence

$$f(i) = \min_{k<i}\{b(k) + c[k][i]\}$$

$k(i) \leq k(i+1)$ holds true if $c[a][c] + c[b][d] < c[a][d] + c[b][c]$. Thus, $k(i)$ can be maintained in a monotone queue.

7.5.2.3 Knuth optimization

For recurrence

$$f(i,j) = \min_{i<k<j}\{f(i,k) + f(k,j)\} + c[i][j]$$

$k(i,j-1) \leq k(i,j) \leq k(i+1,j)$ holds true if $c[a][c] + c[b][d] < c[a][d] + c[b][c]$.

7.5.3 Integration table

7.5.3.1 $ax^2 + bx + c(a > 0)$

1. $\int \frac{\mathrm{d}x}{ax^2+bx+c} = \begin{cases} \frac{2}{\sqrt{4ac-b^2}} \arctan \frac{2ax+b}{\sqrt{4ac-b^2}} + C & (b^2 < 4ac) \\ \frac{1}{\sqrt{b^2-4ac}} \ln \left| \frac{2ax+b-\sqrt{b^2-4ac}}{2ax+b+\sqrt{b^2-4ac}} \right| + C & (b^2 > 4ac) \end{cases}$
2. $\int \frac{x}{ax^2+bx+c}\mathrm{d}x = \frac{1}{2a} \ln |ax^2 + bx + c| - \frac{b}{2a} \int \frac{\mathrm{d}x}{ax^2+bx+c}$

7.5.3.2 $\sqrt{\pm ax^2 + bx + c}(a > 0)$

1. $\int \frac{\mathrm{d}x}{\sqrt{ax^2+bx+c}} = \frac{1}{\sqrt{a}} \ln |2ax + b + 2\sqrt{a}\sqrt{ax^2 + bx + c}| + C$
2. $\int \sqrt{ax^2 + bx + c}\mathrm{d}x = \frac{2ax+b}{4a}\sqrt{ax^2 + bx + c} + \frac{4ac-b^2}{8\sqrt{a^3}} \ln |2ax + b + 2\sqrt{a}\sqrt{ax^2 + bx + c}| + C$
3. $\int \frac{x}{\sqrt{ax^2+bx+c}}\mathrm{d}x = \frac{1}{a}\sqrt{ax^2 + bx + c} - \frac{b}{2\sqrt{a^3}} \ln |2ax + b + 2\sqrt{a}\sqrt{ax^2 + bx + c}| + C$
4. $\int \frac{\mathrm{d}x}{\sqrt{c+bx-ax^2}} = -\frac{1}{\sqrt{a}} \arcsin \frac{2ax-b}{\sqrt{b^2+4ac}} + C$
5. $\int \sqrt{c + bx - ax^2}\mathrm{d}x = \frac{2ax-b}{4a}\sqrt{c + bx - ax^2} + \frac{b^2+4ac}{8\sqrt{a^3}} \arcsin \frac{2ax-b}{\sqrt{b^2+4ac}} + C$
6. $\int \frac{x}{\sqrt{c+bx-ax^2}}\mathrm{d}x = -\frac{1}{a}\sqrt{c + bx - ax^2} + \frac{b}{2\sqrt{a^3}} \arcsin \frac{2ax-b}{\sqrt{b^2+4ac}} + C$

7.5.3.3 $\sqrt{\pm\frac{x-a}{x-b}}$ and $\sqrt{(x-a)(x-b)}$

1. $\int \frac{\mathrm{d}x}{\sqrt{(x-a)(b-x)}} = 2\arcsin\sqrt{\frac{x-a}{b-x}} + C \ (a < b)$
2. $\int \sqrt{(x-a)(b-x)}\mathrm{d}x = \frac{2x-a-b}{4}\sqrt{(x-a)(b-x)} + \frac{(b-a)^2}{4} \arcsin\sqrt{\frac{x-a}{b-x}} + C \ (a < b)$

7.5.3.4 Integration of trigonometric functions

1. $\int \tan x\mathrm{d}x = -\ln|\cos x| + C$
2. $\int \cot x\mathrm{d}x = \ln|\sin x| + C$
3. $\int \sec x\mathrm{d}x = \ln\left|\tan\left(\frac{\pi}{4} + \frac{x}{2}\right)\right| + C = \ln|\sec x + \tan x| + C$
4. $\int \csc x\mathrm{d}x = \ln\left|\tan\frac{x}{2}\right| + C = \ln|\csc x - \cot x| + C$
5. $\int \sec^2 x\mathrm{d}x = \tan x + C$
6. $\int \csc^2 x\mathrm{d}x = -\cot x + C$
7. $\int \sec x \tan x\mathrm{d}x = \sec x + C$

8. $\int \csc x \cot x \mathrm{d}x = -\csc x + C$
9. $\int \sin^2 x \mathrm{d}x = \frac{x}{2} - \frac{1}{4}\sin 2x + C$
10. $\int \cos^2 x \mathrm{d}x = \frac{x}{2} + \frac{1}{4}\sin 2x + C$
11. $\int \sin^n x \mathrm{d}x = -\frac{1}{n}\sin^{n-1} x \cos x + \frac{n-1}{n}\int \sin^{n-2} x \mathrm{d}x$
12. $\int \cos^n x \mathrm{d}x = \frac{1}{n}\cos^{n-1} x \sin x + \frac{n-1}{n}\int \cos^{n-2} x \mathrm{d}x$
13. $\int \frac{\mathrm{d}x}{\sin^n x} = -\frac{1}{n-1}\frac{\cos x}{\sin^{n-1} x} + \frac{n-2}{n-1}\int \frac{\mathrm{d}x}{\sin^{n-2} x}$
14. $\int \frac{\mathrm{d}x}{\cos^n x} = \frac{1}{n-1}\frac{\sin x}{\cos^{n-1} x} + \frac{n-2}{n-1}\int \frac{\mathrm{d}x}{\cos^{n-2} x}$
15. $\int \cos^m x \sin^n x \mathrm{d}x$
$= \frac{1}{m+n}\cos^{m-1} x \sin^{n+1} x + \frac{m-1}{m+n}\int \cos^{m-2} x \sin^n x \mathrm{d}x$
$= -\frac{1}{m+n}\cos^{m+1} x \sin^{n-1} x + \frac{n-1}{m+1}\int \cos^m x \sin^{n-2} x \mathrm{d}x$
16. $\int \frac{\mathrm{d}x}{a+b\sin x} = \begin{cases} \frac{2}{\sqrt{a^2-b^2}}\arctan\frac{a\tan\frac{x}{2}+b}{\sqrt{a^2-b^2}} + C & (a^2 > b^2) \\ \frac{1}{\sqrt{b^2-a^2}}\ln\left|\frac{a\tan\frac{x}{2}+b-\sqrt{b^2-a^2}}{a\tan\frac{x}{2}+b+\sqrt{b^2-a^2}}\right| + C & (a^2 < b^2) \end{cases}$
17. $\int \frac{\mathrm{d}x}{a+b\cos x} = \begin{cases} \frac{2}{a+b}\sqrt{\frac{a+b}{a-b}}\arctan\left(\sqrt{\frac{a-b}{a+b}}\tan\frac{x}{2}\right) + C & (a^2 > b^2) \\ \frac{1}{a+b}\sqrt{\frac{a+b}{a-b}}\ln\left|\frac{\tan\frac{x}{2}+\sqrt{\frac{a+b}{b-a}}}{\tan\frac{x}{2}-\sqrt{\frac{a+b}{b-a}}}\right| + C & (a^2 < b^2) \end{cases}$
18. $\int \frac{\mathrm{d}x}{a^2\cos^2 x + b^2\sin^2 x} = \frac{1}{ab}\arctan\left(\frac{b}{a}\tan x\right) + C$
19. $\int \frac{\mathrm{d}x}{a^2\cos^2 x - b^2\sin^2 x} = \frac{1}{2ab}\ln\left|\frac{b\tan x+a}{b\tan x-a}\right| + C$
20. $\int x\sin ax \mathrm{d}x = \frac{1}{a^2}\sin ax - \frac{1}{a}x\cos ax + C$
21. $\int x^2\sin ax \mathrm{d}x = -\frac{1}{a}x^2\cos ax + \frac{2}{a^2}x\sin ax + \frac{2}{a^3}\cos ax + C$
22. $\int x\cos ax \mathrm{d}x = \frac{1}{a^2}\cos ax + \frac{1}{a}x\sin ax + C$
23. $\int x^2\cos ax \mathrm{d}x = \frac{1}{a}x^2\sin ax + \frac{2}{a^2}x\cos ax - \frac{2}{a^3}\sin ax + C$

7.5.3.5 Integration of inverse trigonometric functions ($a > 0$)

1. $\int \arcsin\frac{x}{a}\mathrm{d}x = x\arcsin\frac{x}{a} + \sqrt{a^2-x^2} + C$
2. $\int x\arcsin\frac{x}{a}\mathrm{d}x = (\frac{x^2}{2} - \frac{a^2}{4})\arcsin\frac{x}{a} + \frac{x}{4}\sqrt{x^2-x^2} + C$
3. $\int x^2\arcsin\frac{x}{a}\mathrm{d}x = \frac{x^3}{3}\arcsin\frac{x}{a} + \frac{1}{9}(x^2+2a^2)\sqrt{a^2-x^2} + C$
4. $\int \arccos\frac{x}{a}\mathrm{d}x = x\ arccos\frac{x}{a} - \sqrt{a^2-x^2} + C$
5. $\int x\arccos\frac{x}{a}\mathrm{d}x = (\frac{x^2}{2} - \frac{a^2}{4})\arccos\frac{x}{a} - \frac{x}{4}\sqrt{a^2-x^2} + C$
6. $\int x^2\arccos\frac{x}{a}\mathrm{d}x = \frac{x^3}{3}\arccos\frac{x}{a} - \frac{1}{9}(x^2+2a^2)\sqrt{a^2-x^2} + C$
7. $\int \arctan\frac{x}{a}\mathrm{d}x = x\arctan\frac{x}{a} - \frac{a}{2}\ln\left(a^2+x^2\right) + C$
8. $\int x\arctan\frac{x}{a}\mathrm{d}x = \frac{1}{2}(a^2+x^2)\arctan\frac{x}{a} - \frac{a}{2}x + C$
9. $\int x^2\arctan\frac{x}{a}\mathrm{d}x = \frac{x^3}{3}\arctan\frac{x}{a} - \frac{a}{6}x^2 + \frac{a^3}{6}\ln\left(a^2+x^2\right) + C$

7.5.3.6 Integration of exponential functions

1. $\int a^x \mathrm{d}x = \frac{1}{\ln a}a^x + C$
2. $\int \mathrm{e}^{ax}\mathrm{d}x = \frac{1}{a}a^{ax} + C$
3. $\int x\mathrm{e}^{ax}\mathrm{d}x = \frac{1}{a^2}(ax-1)a^{ax} + C$
4. $\int x^n\mathrm{e}^{ax}\mathrm{d}x = \frac{1}{a}x^n\mathrm{e}^{ax} - \frac{n}{a}\int x^{n-1}\mathrm{e}^{ax}\mathrm{d}x$
5. $\int xa^x\mathrm{d}x = \frac{x}{\ln a}a^x - \frac{1}{(\ln a)^2}a^x + C$
6. $\int x^n a^x\mathrm{d}x = \frac{1}{\ln a}x^n a^x - \frac{n}{\ln a}\int x^{n-1}a^x\mathrm{d}x$
7. $\int \mathrm{e}^{ax}\sin bx\mathrm{d}x = \frac{1}{a^2+b^2}\mathrm{e}^{ax}(a\sin bx - b\cos bx) + C$
8. $\int \mathrm{e}^{ax}\cos bx\mathrm{d}x = \frac{1}{a^2+b^2}\mathrm{e}^{ax}(b\sin bx + a\cos bx) + C$
9. $\int \mathrm{e}^{ax}\sin^n bx\mathrm{d}x = \frac{1}{a^2+b^2n^2}\mathrm{e}^{ax}\sin^{n-1} bx(a\sin bx - nb\cos bx) + \frac{n(n-1)b^2}{a^2+b^2n^2}\int \mathrm{e}^{ax}\sin^{n-2} bx\mathrm{d}x$
10. $\int \mathrm{e}^{ax}\cos^n bx\mathrm{d}x = \frac{1}{a^2+b^2n^2}\mathrm{e}^{ax}\cos^{n-1} bx(a\cos bx + nb\sin bx) + \frac{n(n-1)b^2}{a^2+b^2n^2}\int \mathrm{e}^{ax}\cos^{n-2} bx\mathrm{d}x$

7.5.3.7 Integration of logistic functions

1. $\int \ln x\mathrm{d}x = x\ln x - x + C$
2. $\int \frac{\mathrm{d}x}{x\ln x} = \ln\left|\ln x\right| + C$
3. $\int x^n\ln x\mathrm{d}x = \frac{1}{n+1}x^{n+1}(\ln x - \frac{1}{n+1}) + C$
4. $\int (\ln x)^n\mathrm{d}x = x(\ln x)^n - n\int (\ln x)^{n-1}\mathrm{d}x$
5. $\int x^m(\ln x)^n\mathrm{d}x = \frac{1}{m+1}x^{m+1}(\ln x)^n - \frac{n}{m+1}\int x^m(\ln x)^{n-1}\mathrm{d}x$

7.5.4 Prefix sum of multiplicative functions

Define the Dirichlet convolution $f * g(n)$ as:

$$f * g(n) = \sum_{d=1}^{n} [d|n] f(n) g(\frac{n}{d})$$

Assume we are going to calculate some function $S(n) = \sum_{i=1}^{n} f(i)$, where $f(n)$ is a multiplicative function. Say we find some $g(n)$ that is simple to calculate, and $\sum_{i=1}^{n} f * g(i)$ can be figured out in $O(1)$ complexity. Then we have

$$\begin{aligned}
\sum_{i=1}^{n} f * g(i) &= \sum_{i=1}^{n} \sum_{d} [d|i] g(\frac{i}{d}) f(d) \\
&= \sum_{\frac{i}{d}=1}^{n} \sum_{d=1}^{\left\lfloor \frac{n}{\frac{i}{d}} \right\rfloor} g(\frac{i}{d}) f(d) \\
&= \sum_{i=1}^{n} \sum_{d=1}^{\left\lfloor \frac{n}{i} \right\rfloor} g(i) f(d) \\
&= g(1) S(n) + \sum_{i=2}^{n} g(i) S(\left\lfloor \frac{n}{i} \right\rfloor) \\
S(n) &= \frac{\sum_{i=1}^{n} f * g(i) - \sum_{i=2}^{n} g(i) S(\left\lfloor \frac{n}{i} \right\rfloor)}{g(1)}
\end{aligned}$$

It can be proven that $\left\lfloor \frac{n}{i} \right\rfloor$ has at most $O(\sqrt{n})$ possible values. Therefore, the calculation of $S(n)$ can be reduced to $O(\sqrt{n})$ calculations of $S(\left\lfloor \frac{n}{i} \right\rfloor)$. By applying the master theorem, it can be shown that the complexity of such method is $O(n^{\frac{3}{4}})$.

Moreover, since $f(n)$ is multiplicative, we can process the first $n^{\frac{2}{3}}$ elements via linear sieve, and for the rest of the elements, we apply the method shown above. The complexity can thus be enhaced to $O(n^{\frac{2}{3}})$.

For the prefix sum of Euler's function $S(n) = \sum_{i=1}^{n} \varphi(i)$, notice that $\sum_{d|n} \varphi(d) = n$. Hence $\varphi * I = id$. $(I(n) = 1, id(n) = n)$ Now let $g(n) = I(n)$, and we have $S(n) = \sum_{i=1}^{n} i - \sum_{i=2}^{n} S(\left\lfloor \frac{n}{i} \right\rfloor)$.

For the prefix sum of Mobius function $S(n) = \sum_{i=1}^{n} \mu(i)$, notice that $\mu * I = (n)\{[n=1]\}$. Hence $S(n) = 1 - \sum_{i=2}^{n} S(\left\lfloor \frac{n}{i} \right\rfloor)$.

Some other convolutions include $(p^k)\{1-p\} * id = I$, $(p^k)\{p^k - p^{k+1}\} * id^2 = id$ and $(p^k)\{p^{2k} - p^{2k-2}\} * I = id^2$.

Example code :

```
/*  Prefix sum of multiplicative functions :
        CUBEN : N ^ (1 / 3).
        p_f : the prefix sum of f (x) (1 <= x <= th).
        p_g : the prefix sum of g (x) (0 <= x <= N).
        p_c : the prefix sum of f (x) * g (x) (0 <= x <= N).
        th : the thereshold, generally should be x ^ (2 / 3).
        REMEMBER THAT x IN p_g (x) AND p_c (x) MAY BE LARGER THAN MOD!!
*/
template <int CUBEN = 11000>
struct prefix_mul {
    typedef long long (*func) (long long);
    func p_f, p_g, p_c;
    long long n, mod, th, inv;
    std::unordered_map <long long, long long> mem;
    prefix_mul (func p_f, func p_g, func p_c, long long mod) : p_f (p_f), p_g (p_g), p_c (p_c), mod (mod) {}
    void euclid (long long a, long long b, long long &x, long long &y) {
        if (b == 0) x = 1, y = 0;
        else euclid (b, a % b, y, x), y -= a / b * x;
    }
    long long inverse (long long x, long long m) {
        long long a, b;
        euclid (x, m, a, b);
        return (a % m + m) % m;
    }
    long long calc (long long x) {
        if (x <= th) return p_f (x);
        auto d = mem.find (x);
        if (d != mem.end ()) return d -> second;
        long long ans = 0;
        for (long long i = 2, la; i <= x; i = la + 1) {
            la = x / (x / i);
            ans = (ans + (p_g (la) - p_g (i - 1) + mod) * calc (x / i)) % mod;
        }
        ans = p_c (x) - ans; if (ans < 0) ans += mod; ans = ans * inv % mod;
        return mem[x] = ans;
    }
    long long solve (long long n, long long th) {
        if (n <= 0) return 0;
        prefix_mul::n = n; prefix_mul::th = th;
        inv = inverse (p_g (1), mod);
        return calc (n);
    }
};
```

7.5.5 Prufer sequence

In combinatorial mathematics, the Prufer sequence of a labeled tree is a unique sequence associated with the tree. The sequence for a tree on n vertices has length $n-2$.

One can generate a labeled tree's Prufer sequence by iteratively removing vertices from the tree until only two vertices remain. Specifically, consider a labeled tree T with vertices $1, 2, ..., n$. At step i, remove the leaf with the smallest label and set the ith element of the Prufer sequence to be the label of this leaf's neighbour.

One can generate a labeled tree from a sequence in three steps. The tree will have $n+2$ nodes, numbered from 1 to $n+2$. For each node set its degree to the number of times it appears in the sequence plus 1. Next, for each number in the sequence $a[i]$, find the first (lowest-numbered) node, j, with degree equal to 1, add the edge $(j, a[i])$ to the tree, and decrement the degrees of j and $a[i]$. At the end of this loop two nodes with degree 1 will remain (call them u, v). Lastly, add the edge (u, v) to the tree.

The Prufer sequence of a labeled tree on n vertices is a unique sequence of length $n-2$ on the labels 1 to n - this much is clear. Somewhat less obvious is the fact that for a given sequence S of length $n-2$ on the labels 1 to n, there is a unique labeled tree whose Prufer sequence is S.

7.5.6 Mobius inversion

7.5.6.1 Mobius inversion formula

$$[x=1] = \sum_{d|x} \mu(d)$$

$$x = \sum_{d|x} \mu(d)$$

7.5.6.2 Gcd inversion

$$\begin{aligned}
\sum_{a=1}^{n}\sum_{b=1}^{n} gcd^2(a,b) &= \sum_{d=1}^{n} d^2 \sum_{i=1}^{\lfloor \frac{n}{d} \rfloor}\sum_{j=1}^{\lfloor \frac{n}{d} \rfloor} [gcd(i,j)=1] \\
&= \sum_{d=1}^{n} d^2 \sum_{i=1}^{\lfloor \frac{n}{d} \rfloor}\sum_{j=1}^{\lfloor \frac{n}{d} \rfloor} \sum_{t|gcd(i,j)} \mu(t) \\
&= \sum_{d=1}^{n} d^2 \sum_{t=1}^{\lfloor \frac{n}{d} \rfloor} \mu(t) \sum_{i=1}^{\lfloor \frac{n}{d} \rfloor} [t|i] \sum_{j=1}^{\lfloor \frac{n}{d} \rfloor} [t|j] \\
&= \sum_{d=1}^{n} d^2 \sum_{t=1}^{\lfloor \frac{n}{d} \rfloor} \mu(t) \lfloor \frac{n}{dt} \rfloor^2
\end{aligned}$$

The formula can be calculated in $O(nlogn)$ complexity.

Moreover, let $l = dt$, then

$$\sum_{d=1}^{n} d^2 \sum_{t=1}^{\lfloor \frac{n}{d} \rfloor} \mu(t) \lfloor \frac{n}{dt} \rfloor^2 = \sum_{l=1}^{n} \lfloor \frac{n}{l} \rfloor^2 \sum_{d|l} d^2 \mu(\frac{l}{d})$$

Let $f(l) = \sum_{d|l} d^2 \mu(\frac{l}{d})$. It can be proven that $f(l)$ is multiplicative. Besides, $f(p^k) = p^{2k} - p^{2k-2}$.

Therefore, with linear sieve the formula can be solved in $O(n)$ complexity.

7.5.7 Spanning tree counting

Kirchhoff's Theorem: the number of spanning trees in a graph G is equal to *any* cofactor of the Laplacian matrix of G, which is equal to the difference between the graph's degree matrix (a diagonal matrix with vertex degrees on the diagonals) and its adjacency matrix (a (0,1)-matrix with 1's at places corresponding to entries where the vertices are adjacent and 0's otherwise).

The number of edges with a certain weight in a minimum spanning tree is fixed given a graph. Moreover, the number of its arrangements can be obtained by finding a minimum spannig tree, compressing connected components of other edges in that tree into a point, and then applying Kirrchoff's theorem with only edges of the certain weight in the graph. Therefore, the number of minimum spanning trees in a graph can be solved by multiplying all numbers of arrangements of edges of different weight together.

7.6 Regular expression

```
/*  C++11 supports regular expressions, see below for an example.   */
char str[] = "The thirty-three thieves thought that they thrilled the throne throughout Thursday.";
std::regex pattern ("(th|Th)[\\w]*", std::regex_constants::optimize | std::regex_constants::ECMAScript);
std::match_results <char *> match;
std::regex_constants::match_flag_type flag = std::regex_constants::match_default;
int begin = 0, end = strlen (str);
while (std::regex_search (str + begin, str + end, match, pattern, flag)) {
    std::cout << match[0] << " " << match[1] << std::endl;
    begin += match.position (0) + 1;
    flag |= std::regex_constants::match_prev_avail;
}
```

7.6.1 Special pattern characters

Characters	Description	Matches
`.`	Not newline	Any character except line terminators (LF, CR, LS, PS).
`\t`	Tab (HT)	A horizontal tab character (same as `\u0009`).
`\n`	Newline (LF)	A newline (line feed) character (same as `\u000A`).
`\v`	Vertical tab (VT)	A vertical tab character (same as `\u000B`).
`\f`	Form feed (FF)	A form feed character (same as `\u000C`).
`\r`	Carriage return (CR)	A carriage return character (same as `\u000D`).
`\cletter`	Control code	A control code character whose code unit value is the same as the remainder of dividing the code unit value of letter by 32. For example: `\ca` is the same as `\u0001`, `\cb` the same as `\u0002`, and so on...
`\xhh`	ASCII character	A character whose code unit value has an hex value equivalent to the two hex digits hh. For example: `\x4c` is the same as `L`, or `\x23` the same as `#`.
`\uhhhh`	Unicode character	A character whose code unit value has an hex value equivalent to the four hex digits hhhh.
`\0`	Null	A null character (same as `\u0000`).
`\int`	Backreference	The result of the submatch whose opening parenthesis is the int-th (int shall begin by a digit other than 0). See groups below for more info.
`\d`	Digit	A decimal digit character (same as `[[:digit:]]`).
`\D`	Not digit	Any character that is not a decimal digit character (same as `[^[:digit:]]`).
`\s`	Whitespace	A whitespace character (same as `[[:space:]]`).
`\S`	Not whitespace	Any character that is not a whitespace character (same as `[^[:space:]]`).
`\w`	Word	An alphanumeric or underscore character (same as `[_[:alnum:]]`).
`\W`	Not word	Any character that is not an alphanumeric or underscore character (same as `[^_[:alnum:]]`).
`\character`	Character	The character character as it is, without interpreting its special meaning within a regex expression. Any character can be escaped except those which form any of the special character sequences above. Needed for: `^ $ \ . * + ? ( ) [ ] { } |`.
`[class]`	Character class	The target character is part of the class (see character classes below).
`[^class]`	Negated character class	The target character is not part of the class (see character classes below).

7.6.2 Quantifiers

Characters	Times	Effects
`*`	0 or more	The preceding atom is matched 0 or more times.
`+`	1 or more	The preceding atom is matched 1 or more times.
`?`	0 or 1	The preceding atom is optional (matched either 0 times or once).
`{int}`	`int`	The preceding atom is matched exactly `int` times.
`{int,}`	`int` or more	The preceding atom is matched `int` or more times.
`{min,max}`	Between `min` and `max`	The preceding atom is matched at least `min` times, but not more than `max`.

By default, all these quantifiers are greedy (i.e., they take as many characters that meet the condition as possible). This behavior can be overridden to ungreedy (i.e., take as few characters that meet the condition as possible) by adding a question mark (`?`) after the quantifier.

7.6.3 Groups

Characters	Description	Effects
`(subpattern)`	Group	Creates a backreference.
`(?:subpattern)`	Passive group	Does not create a backreference.

7.6.4 Assertions

Characters	Description	Condition for match
`^`	Beginning of line	Either it is the beginning of the target sequence, or follows a line terminator.
`$`	End of line	Either it is the end of the target sequence, or precedes a line terminator.
`\b`	Word boundary	The previous character is a word character and the next is a non-word character (or vice-versa). Note: The beginning and the end of the target sequence are considered here as non-word characters.
`\B`	Not a word boundary	The previous and next characters are both word characters or both are non-word characters. Note: The beginning and the end of the target sequence are considered here as non-word characters.
`(?=subpattern)`	Positive lookahead	The characters following the assertion must match subpattern, but no characters are consumed.
`(?!subpattern)`	Negative lookahead	The characters following the assertion must not match subpattern, but no characters are consumed.

7.6.5 Alternative

A regular expression can contain multiple alternative patterns simply by separating them with the separator operator (`|`): The regular expression will match if any of the alternatives match, and as soon as one does.

7.6.6 Character classes

Class	Description	Equivalent (with regex_traits, default locale)
`[:alnum:]`	Alpha-numerical character	`isalnum`
`[:alpha:]`	Alphabetic character	`isalpha`
`[:blank:]`	Blank character	`isblank`
`[:cntrl:]`	Control character	`iscntrl`
`[:digit:]`	Decimal digit character	`isdigit`
`[:graph:]`	Character with graphical representation	`isgraph`
`[:lower:]`	Lowercase letter	`islower`
`[:print:]`	Printable character	`isprint`
`[:punct:]`	Punctuation mark character	`ispunct`
`[:space:]`	Whitespace character	`isspace`
`[:upper:]`	Uppercase letter	`isupper`
`[:xdigit:]`	Hexadecimal digit character	`isxdigit`
`[:d:]`	Decimal digit character	`isdigit`
`[:w:]`	Word character	`isalnum`
`[:s:]`	Whitespace character	`isspace`

Please note that the brackets in the class names are additional to those opening and closing the class definition. For example:

`[[:alpha:]]` is a character class that matches any alphabetic character.
`[abc[:digit:]]` is a character class that matches a, b, c, or a digit.
`[^[:space:]]` is a character class that matches any character except a whitespace.