Contents

The Grimoire of Programming

Suzune Nisiyama

Chapter 1

Trial of number theory

1.1 Constants and basic functions

```
const double PI = acos (-1);
/*  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.
*/
long long abs (const long long &x) { return x > 0 ? x : -x; }
long long inverse (const long long &x, const long long &mod) {
    if (x == 1) return 1;
    return (mod - mod / x) * inverse (mod % x, mod) % mod;
}
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 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 ans = 0, add = a, k = b;
    while (k) {
        if (k & 1) ans = (ans + add) % mod;
        add = (add + add) % mod;
        k >>= 1;
    }
    return ans;
}
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;
}
```

1.2 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 ());
        }
    }
};
```

1.3 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;
    }
};
```

1.4 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;
    }
};
```

1.5 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;
    }
};
```

1.6 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)
                if (number % i)
                    ++i;
                else {
                    ans.push_back (i);
                    rem %= i;
                }
        }
        return ans;
    }
};
```

1.7 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 2

Trial of geometry

2.1 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; }
```

2.2 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;
    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 rotate (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));
}
```

2.3 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;
    line (const point &s = point (), const point &t = point ()) : s (s), t (t) {}
    double length () const { return dis (s, t); }
};
```

2.4 Interactions between points and lines

```
/*  Point & line interactions :
        bool point_on_line (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_line (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_judgement (const line &a, const line &b) {
    if (point_on_line (b.s, a) || point_on_line (b.t, a)) return true;
    if (point_on_line (a.s, b) || point_on_line (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);
}
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_line (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;
}
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 ());
}
```

2.5 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;
}
```

2.6 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).rotate (sq3)), line (b, c + (a - c).rotate (sq3)));
    else
        mid = line_intersect (line (a, c + (b - c).rotate (sq3)), line (c, b + (a - b).rotate (sq3)));
    return mid;
}
```

2.7 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;
    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);
}
```

2.8 Interactions of circles

```
/*  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);
    return circle_intersect (p, b);
}
```

2.9 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;
}
```

2.10 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;
}
```

2.11 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;
}
```

2.12 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);
    }
};
```

2.13 Union of circles

```
/*  Union of circles :
```

```
        std::vector <double> union_circle::solve (const std::vector <circle> &c) :
            returns the union of circle set c.
            The i-th element is the area covered with at least i circles.
*/
struct union_circle {
    struct cp {
        double x, y, angle;
        int d;
        double r;
        cp (const double &x = 0, const double &y = 0, const double &angle = 0,
            int d = 0, const double &r = 0) : x (x), y (y), angle (angle), d (d), r (r) {}
    };
    double dis (const cp &a, const cp &b) {
        return sqrt (sqr (a.x - b.x) + sqr (a.y - b.y));
    }
    double cross (const cp &p0, const cp &p1, const cp &p2) {
        return (p1.x - p0.x) * (p2.y - p0.y) - (p1.y - p0.y) * (p2.x - p0.x);
    }
    int cir_cross (cp p1, double r1, cp p2, double r2, cp &cp1, cp &cp2) {
        double mx = p2.x - p1.x, sx = p2.x + p1.x, mx2 = mx * mx;
        double my = p2.y - p1.y, sy = p2.y + p1.y, my2 = my * my;
        double sq = mx2 + my2, d = - (sq - sqr (r1 - r2)) * (sq - sqr (r1 + r2));
        if (sgn (d) < 0) return 0;
        if (sgn (d) <= 0) d = 0;
        else d = sqrt (d);
        double x = mx * ((r1 + r2) * (r1 - r2) + mx * sx) + sx * my2;
        double y = my * ((r1 + r2) * (r1 - r2) + my * sy) + sy * mx2;
        double dx = mx * d, dy = my * d;
        sq *= 2;
        cp1.x = (x - dy) / sq;
        cp1.y = (y + dx) / sq;
        cp2.x = (x + dy) / sq;
        cp2.y = (y - dx) / sq;
        if (sgn (d) > 0) return 2;
        else return 1;
    }
    static bool circmp (const cp &u, const cp &v) {
        return sgn (u.r - v.r) < 0;
    }
    static bool cmp (const cp &u, const cp &v) {
        if (sgn (u.angle - v.angle)) return u.angle < v.angle;
        return u.d > v.d;
    }
    double calc (cp cir, cp cp1, cp cp2) {
        double ans = (cp2.angle - cp1.angle) * sqr (cir.r)
                     - cross (cir, cp1, cp2) + cross (cp (0, 0), cp1, cp2);
        return ans / 2;
    }
    std::vector <double> solve (const std::vector <circle> &c) {
        int n = c.size ();
        std::vector <cp> cir, tp;
        std::vector <double> area;
        cir.resize (n);
        tp.resize (2 * n);
        area.resize (n + 1);
        for (int i = 0; i < n; i++)
            cir[i] = cp (c[i].c.x, c[i].c.y, 0, 1, c[i].r);
        cp cp1, cp2;
        std::sort (cir.begin (), cir.end (), circmp);
        for (int i = 0; i < n; ++i)
            for (int j = i + 1; j < n; ++j)
                if (sgn (dis (cir[i], cir[j]) + cir[i].r - cir[j].r) <= 0)
                    cir[i].d++;
        for (int i = 0; i < n; ++i) {
            int tn = 0, cnt = 0;
            for (int j = 0; j < n; ++j) {
                if (i == j) continue;
                if (cir_cross (cir[i], cir[i].r, cir[j], cir[j].r, cp2, cp1) < 2) continue;
                cp1.angle = atan2 (cp1.y - cir[i].y, cp1.x - cir[i].x);
                cp2.angle = atan2 (cp2.y - cir[i].y, cp2.x - cir[i].x);
                cp1.d = 1;
                tp[tn++] = cp1;
                cp2.d = -1;
                tp[tn++] = cp2;
                if (sgn (cp1.angle - cp2.angle) > 0) cnt++;
            }
            tp[tn++] = cp (cir[i].x - cir[i].r, cir[i].y, PI, -cnt);
            tp[tn++] = cp (cir[i].x - cir[i].r, cir[i].y, -PI, cnt);
            std::sort (tp.begin (), tp.begin () + tn, cmp);
            int p, s = cir[i].d + tp[0].d;
            for (int j = 1; j < tn; ++j) {
                p = s;
                s += tp[j].d;
                area[p] += calc (cir[i], tp[j - 1], tp[j]);
            }
        }
        return area;
    }
};
```

Chapter 3

Trial of graph

3.1 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++;
    }
};
```

3.2 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;
    }
};
```

3.3 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 ();
        }
    }
};
```

3.4 Tarjan

```
/*  Tarjan :
        returns strongly connected components.
        void tarjan::solve (const edge_list &) :
            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], vis[MAXN], s[MAXN], s_s, ins[MAXN], ind;
    void dfs (const edge_list <MAXN, MAXM> &e, int u) {
        dfn[u] = low[u] = ind++;
        vis[u] = ins[u] = 1; s[s_s++] = u;
        for (int i = e.begin[u]; ~i; i = e.next[i]) {
            if (!vis[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]] = 0;
            } while (s[s_s] != u);
            component_size++;
        }
    }
    void solve (const edge_list <MAXN, MAXM> &e) {
        std::fill (vis, vis + MAXN, 0);
        std::fill (ins, ins + MAXN, 0);
        s_s = ind = 0;
        dfs (e, 0);
    }
};
```

3.5 Hopcoft-Carp

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

3.6 Kuhn-Munkres

```
/*  Kuhn Munkres algorithm :
        weighted maximum matching algorithm. Complexity O (N^3).
        struct kuhn_munkres :
            Initialize : pass nx, ny as the size of both sets, w as the weight matrix.
            Usage : solve () for the minimum matching. The exact matching is in link[].
*/
template <int MAXN = 500>
struct kuhn_munkres {
    int nx, ny;
    int w[MAXN][MAXN];
    int lx[MAXN], ly[MAXN], visx[MAXN], visy[MAXN], slack[MAXN], link[MAXN];
    int dfs (int x) {
        visx[x] = 1;
        for (int y = 0; y < ny; y ++) {
            if (visy[y]) continue;
            int t = lx[x] + ly[y] - w[x][y];
            if (t == 0) {
                visy[y] = 1;
                if (link[y] == -1 || dfs (link[y])) {
```

```
                link[y] = x;
                return 1;
            }
        } else slack[y] = std::max (slack[y], t);
    }
    return 0;
}
int solve () {
    int i, j;
    std::fill (link, link + ny, -1);
    std::fill (ly, ly + ny, 0);
    for (i = 0; i < nx; i++)
        for (j = 0, lx[i] = INF; j < ny; j++)
            lx[i] = std::min (lx[i], w[i][j]);
    for (int x = 0; x < nx; x++) {
        for (i = 0; i < ny; i++) slack[i] = -INF;
        while (true) {
            std::fill (visx, visx + nx, 0);
            std::fill (visy, visy + ny, 0);
            if (dfs (x)) break;
            int d = INF;
            for (i = 0; i < ny; i++)
                if (!visy[i] && d < slack[i]) d = slack[i];
            for (i = 0; i < nx; i++)
                if (visx[i]) lx[i] -= d;
            for (i = 0; i < ny; i++)
                if (visy[i]) ly[i] += d;
                else slack[i] -= d;
        }
    }
    int res = 0;
    for (i = 0; i < ny; i ++)
        if (link[i] > -1) res += w[link[i]][i];
    return res;
}
};
```

3.7 Stochastic weighted maximum matching

```
/*  Weighted matching algorithm :
        maximum match for graphs. Not stable.
        struct weighted_match :
            Usage : Set k to the size of vertices, w to the weight matrix.
            Note that k has to be even for the algorithm to work.
*/
template <int MAXN = 500>
struct weighted_match {
    int k;
    long long w[MAXN][MAXN];
    int match[MAXN], path[MAXN], p[MAXN], len;
    long long d[MAXN];
    bool v[MAXN];
    bool dfs (int i) {
        path[len++] = i;
        if (v[i]) return true;
        v[i] = true;
        for (int j = 0; j < k; ++j) {
            if (i != j && match[i] != j && !v[j]) {
                int kok = match[j];
                if (d[kok] < d[i] + w[i][j] - w[j][kok]) {
                    d[kok] = d[i] + w[i][j] - w[j][kok];
                    if (dfs (kok)) return true;
                }
            }
        }
        --len;
        v[i] = false;
        return false;
    }
    long long solve () {
        if (k & 1) ++k;
        for (int i = 0; i < k; ++i) p[i] = i, match[i] = i ^ 1;
        int cnt = 0;
        for (;;) {
            len = 0;
            bool flag = false;
            std::fill (d, d + k, 0);
            std::fill (v, v + k, 0);
            for (int i = 0; i < k; ++i) {
                if (dfs (p[i])) {
                    flag = true;
                    int t = match[path[len - 1]], j = len - 2;
                    while (path[j] != path[len - 1]) {
                        match[t] = path[j];
                        std::swap (t, match[path[j]]);
                        --j;
                    }
                    match[t] = path[j];
                    match[path[j]] = t;
                    break;
                }
            }
            if (!flag) {
                if (++cnt >= 2) break;
                std::random_shuffle (p, p + k);
            }
        }
        long long ans = 0;
        for (int i = 0; i < k; ++i)
            ans += w[i][match[i]];
        return ans / 2;
```

```
    }
};
```

3.8 Weighted blossom (vfleaking ver.)

```
/*  Weighted blossom algorithm (vfleaking ver.) :
        maximum match for 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);
    }
};
```

3.9 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);
        }
    }
};
```

3.10 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);
    }
};
```

Chapter 4

Trial of string

4.1 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 = 1E6>
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) {
        for (int i = 0, j = 0; i < (int) str.size (); ++i, ++j) {
            if (j == match.size ()) return i - match.size ();
            while (~j && str[i] != match[j]) j = fail[j];
        }
        return str.size ();
    }
};
```

4.2 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::parent : the parent link.
        state::dest : the automaton link.
*/
template <int MAXN = 1E6, int MAXC = 26>
struct suffix_automaton {
    state *head, *tail;
    struct state {
        int len;
        state *parent, *dest[MAXC];
        state (int len = 0) : len (len), parent (NULL) {
            memset (dest, 0, sizeof (dest));
        }
        state *extend (state *, int token);
    } node_pool[MAXN * 2], *tot_node, *null = new state();
    state *state::extend (state *start, int token) {
        state *p = this;
        state *np = this -> dest[token] ? null : new (tot_node++) state (this -> len + 1);
        while (p && !p -> dest[token])
            p -> dest[token] = np, p = p -> parent;
        if (!p) np -> parent = start;
        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;
                }
```

```
            }
        }
        return np == null ? np -> parent : np;
    }
    void init () {
        tot_node = node_pool;
        head = tail = new (tot_node++) state();
    }
    suffix_automaton () {
        init ();
    }
    void extend (int token) {
        tail = tail -> extend (head, token);
    }
};
```

4.3 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 = 1E6, 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 5

Reference

5.1 Vimrc

```
set ruler
set number
set tabstop=4
set softtabstop=4
set shiftwidth=4
set smartindent
set showmatch
set hlsearch
set incsearch
set autoread
set backspace=2
set mouse=a
syntax on
nmap <C-A> ggVG
vmap <C-C> "+y
nmap <C-P> "+p
autocmd FileType cpp set cindent
autocmd FileType cpp map <F3> :vsplit %<.in <CR>
autocmd FileType cpp map <F5> :!time ./%<.exe <CR>
autocmd FileType cpp map <F7> :!gdb ./%<.exe <CR>
autocmd FileType cpp map <F8> :!time ./%<.exe < %<.in <CR>
autocmd FileType cpp map <F9> :!g++ % -o %< -g -std=c++11 -Wall -Wextra -Wconversion && size %<.exe <CR>
autocmd FileType java map <F3> :vsplit %<.in <CR>
autocmd FileType java map <F5> :!time java %< <CR>
autocmd FileType java map <F8> :!time java %< < %<.in <CR>
autocmd FileType java map <F9> :!javac % <CR>
```

5.2 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.
    }
}
```

5.3 Operator precedence

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`a++ a--`	Suffix/postfix increment and decrement	
	`type()` `type{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`. ->`	Member access	
3	`++a --a`	Prefix increment and decrement	Right-to-left
	`+a -a`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(type)`	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of	
	`new new[]`	Dynamic memory allocation	
	`delete delete[]`	Dynamic memory deallocation	
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`	Ternary conditional	Right-to-left
	`throw`	throw operator	
	`=`	Direct assignment	
	`+= -= *= /= %=`	Compound assignment by arithmetic operation	
	`<<= >>=`	Compound assignment by bitwise shift	
	`&= ^= |=`	Compound assignment by bitwise AND, XOR, and OR	
16	`,`	Comma	Left-to-right

5.4 Hacks

5.4.1 Formating long long in scanf & printf

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

5.4.2 Optimizing

```
#pragma GCC optimize ("O3")
#pragma GCC optimize ("whole-program")
```

5.4.3 Larger stack

5.4.3.1 C++

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

5.4.3.2 G++

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

5.5 Math reference

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

5.5.2 Dynamic programming optimization

5.5.2.1 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.

5.5.2.2 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.

5.5.2.3 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 $c[a][c] + c[b][d] < c[a][d] + c[b][c]$.

5.5.3 Integration table

5.5.3.1 $ax^2 + bx + c$ $(a > 0)$

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. $\int \frac{x}{ax^2+bx+c} \,\mathrm{d}x = \frac{1}{2a} \ln \left| ax^2 + bx + c \right| - \frac{b}{2a} \int \frac{\mathrm{d}x}{ax^2+bx+c}$

5.5.3.2 $\sqrt{\pm ax^2 + bx + c}$ $(a > 0)$

1. $\int \frac{\mathrm{d}x}{ax^2+bx+c} = \frac{1}{\sqrt{a}} \ln \left| 2ax + b + 2\sqrt{a}\sqrt{ax^2 + bx + c} \right| + C$

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

3. $\int \frac{x}{\sqrt{ax^2+bx+c}} \,\mathrm{d}x = \frac{1}{a} \sqrt{ax^2 + bx + c} - \frac{b}{2\sqrt{a^3}} \ln \left| 2ax + b + 2\sqrt{a}\sqrt{ax^2 + bx + c} \right| + C$

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

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

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

5.5.3.3 Trigonometric

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. $\int \cos^m x \sin^n x \,\mathrm{d}x = \frac{1}{m+n} \cos^{m-1} x \sin^{n+1} x + \frac{m-1}{m+n} \int \cos^{m-2} x \sin^n x \,\mathrm{d}x = -\frac{1}{m+n} \cos^{m+1} x \sin^{n-1} x + \frac{n-1}{m+1} \int \cos^m x \sin^{n-2} x \,\mathrm{d}x$

5.5.3.4 Inverse trigonometric $(a > 0)$

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

5.5.3.5 Exponential

1. $\int a^x \,\mathrm{d}x = \frac{1}{\ln a} a^x + C$
2. $\int e^{ax} \,\mathrm{d}x = \frac{1}{a} a^{ax} + C$

5.5.3.6 Logistic

1. $\int \ln x \, \mathrm{d}x = x \ln x - x + C$
2. $\frac{\mathrm{d}x}{x \ln x} = \ln |\ln x| + C$

5.5.4 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 $\varphi * I(n) = id(n).(I(n) = 1, id(n) = n)$ 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 $S(n) = 1 - \sum_{i=2}^{n} S(\left\lfloor \frac{n}{i} \right\rfloor)$.

5.5.5 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.

5.5.6 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.

5.6 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;
}
```

5.6.1 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).

5.6.2 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.

5.6.3 Groups

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

5.6.4 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.

5.6.5 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.

5.6.6 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.