

# Luna's Magic Reference

Suzune Nisiyama

January 22, 2019

---

MIT License

Copyright (c) 2018 Nisiyama-Suzune

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

# Contents

<b>1 Environment</b>	<b>2</b>		
1.1 Vimrc	2		
<b>2 Data Structure</b>	<b>2</b>		
2.1 Balanced tree	2		
2.1.1 Link-cut tree	2		
2.1.2 Splay operation	2		
2.2 KD tree	2		
2.3 RMQ	3		
<b>3 Geometry</b>	<b>3</b>		
3.1 2D geometry	3		
3.1.1 Convex hull	4		
3.1.2 Delaunay triangulation	4		
3.1.3 Fermat point	5		
3.1.4 Half plane intersection	5		
3.1.5 Intersection of a polygon and a circle	5		
3.1.6 Minimum circle	5		
3.1.7 Nearest pair of points	5		
3.1.8 Triangle center	5		
3.1.9 Union of circles	5		
3.2 3D geometry	6		
3.2.1 3D point	6		
3.2.2 3D line	6		
3.2.3 3D convex hull	6		
<b>4 Graph</b>	<b>6</b>		
4.1 Characteristic	6		
4.1.1 Chordal graph	6		
4.1.2 Euler characteristic	7		
4.2 Clique	7		
4.2.1 DN maximum clique	7		
4.3 Cut	7		
4.3.1 2-SAT	7		
4.3.2 Dominator tree	8		
4.3.3 Stoer Wagner algorithm	8		
4.3.4 Tarjan	8		
4.4 Flow	8		
4.4.1 Maximum flow	8		
4.4.2 Minimum cost flow	9		
4.5 Matching	9		
4.5.1 Blossom algorithm	9		
4.5.2 Blossom algorithm (weighted)	10		
4.5.3 Hopcroft-Karp algorithm	10		
4.5.4 Kuhn-Munkres algorithm	11		
4.6 Path	11		
4.6.1 K-shortest path	11		
4.6.2 Lindström-Gessel-Viennot lemma	11		
4.7 Tree	11		
4.7.1 Optimum branching	11		
4.7.2 Prufer sequence	12		
4.7.3 Spanning tree counting	12		
4.7.4 Tree hash	12		
<b>5 Mathematics</b>	<b>12</b>		
5.1 Computation	12		
5.1.1 Adaptive Simpson's method	12		
5.1.2 Dirichlet convolution	12		
5.1.3 Dirichlet inversion	12		
5.1.4 Euclidean-like algorithm	13		
5.1.5 Extended Eratosthenes sieve	13		
5.1.6 Fast power module	13		
5.1.7 Lucas's theorem	13		
5.1.8 Mobius inversion	13		
5.1.9 Zeller's congruence	14		
5.2 Dynamic programming	14		
5.3 Equality and inequality	14		
5.3.1 Baby step giant step algorithm	14		
5.3.2 Chinese remainder theorem	14		
5.3.3 Extended Euclidean algorithm	14		
5.3.4 Pell equation	14		
5.3.5 Quadric residue	14		
5.3.6 Simplex	14		
5.4 Game theory	14		
5.5 Group theory	15		
5.5.1 Pólya enumeration theorem	15		
5.5.2 Schreier Sims	15		
5.6 Machine learning	15		
5.6.1 Neural network	15		
5.7 Primality	16		
5.7.1 Miller Rabin primality test	16		
5.7.2 Pollard's Rho algorithm	16		
5.7.3 SQUFOF	16		
5.8 Recurrence relation	17		
5.8.1 Berlekamp Massey algorithm	17		
5.8.2 Linear Recurrence	17		
5.9 Sequence manipulation	17		
5.9.1 Discrete Fourier transform	17		
5.9.2 Fast Walsh-Hadamard transform	18		
5.9.3 Number theoretic transform	18		
5.9.4 Polynomial operation	18		
<b>6 String</b>	<b>18</b>		
6.1 Decomposition	18		
6.1.1 Lyndon word	18		
6.2 Matching	19		
6.2.1 Exkmp	19		
6.2.2 Minimal string rotation	19		
6.3 Palindrome	19		
6.3.1 Manacher	19		
6.3.2 Palindromic tree	19		
6.4 Suffix	19		
6.4.1 Suffix array (SAIS)	19		
6.4.2 Suffix automaton	20		
<b>7 System</b>	<b>20</b>		
7.1 Builtin functions	20		
7.2 Container memory release	20		
7.3 Fast IO	20		
7.4 Formatting	20		
7.5 Java	20		
7.6 Random numbers	21		
7.7 Regular expression	21		
7.8 Stack hack	21		
7.9 Time hack	21		
<b>8 Appendix</b>	<b>22</b>		
8.1 Table of formulae	22		
8.2 Table of integrals	23		
8.3 Table of range	24		
8.4 Table of regular expression	24		
8.5 Table of operator precedence	25		

# 1 Environment

## 1.1 Vimrc

```

1 set ru nu ts=4 sts=4 sw=4 si sm hls is ar bs=2 mouse=a
2 syntax on
3 nm <F3> :vsplit %<.in <CR>
4 nm <F4> :!gedit % <CR>
5 au BufEnter *.cpp set cin
6 au BufEnter *.cpp nm <F5> :!time ./%< <CR>|nm <F7> :!
   gdb ./%< <CR>|nm <F8> :!time ./%< <CR>|nm
   <F9> :!g++ % -o %< -g -std=gnu++14 -O2 -DLOCAL -
   Wall -Wconversion && size %< <CR>
7 au BufEnter *.java nm <F5> :!time java %< <CR>|nm <F8>
   :!time java %< <CR>|nm <F9> :!javac % <CR>
   >

```

# 2 Data Structure

## 2.1 Balanced tree

### 2.1.1 Link-cut tree

```

1 struct Node { int son[2], fa, num, pos, rev; } node[
   maxn];
2 int n, m, ans, top, q[maxn];
3 inline bool root (int x) { return node[x].fa == 0; }
4 void update (int x) {
5     int left = node[x].son[0], right = node[x].son[1];
6     node[x].pos = x;
7     if (node[left].pos.num > node[x].pos.num)
8         node[x].pos = node[left].pos;
9     if (node[right].pos.num > node[x].pos.num)
10        node[x].pos = node[right].pos;
11 void down (int x) {
12     int left = node[x].son[0], right = node[x].son[1];
13     if (node[x].rev) { node[x].rev ^= 1; node[left].rev
14         ^= 1; node[right].rev ^= 1;
15         std::swap (node[x].son[0], node[x].son[1]); }
16 void rotate (int x) {
17     int y = node[x].fa, z = node[y].fa, left, right;
18     if (node[y].son[0] == x) left = 0; else left = 1;
19     right = left ^ 1; if (!root (y)) {
20         if (node[z].son[0] == y) node[z].son[0] = x; else
21             node[z].son[1] = x;
22     node[x].fa = z; node[y].fa = x;
23     if (node[x].son[right] != 0) node[node[x].son[right]
24         ].fa = y;
25     node[y].son[left] = node[x].son[right]; node[x].son[
26         right] = y;
27     update (y); update (x); }
28 void splay (int x) {
29     top = 0; q[++top] = x;
30     for (int i = x; !root (i); i = node[i].fa) q[++top] =
31         node[i].fa;
32     for (int i = top; i; i--) down (q[i]);
33     while (!root (x)) { int y = node[x].fa, z = node[y].
34         fa; if (!root (y)) {
35         if (node[y].son[0] == x ^ node[z].son[0] == y)
36             rotate (x);
37         else rotate (y); } rotate (x); } update (x); }
38 void access (int x) { int t = 0; while (x) { splay (x)
39     ; node[x].son[1] = t; t = x; x = node[x].fa; } }
40 void makeroot (int x) { access (x); splay (x); node[x]
41     .rev ^= 1; }
42 void link (int x, int y) { makeroot (x); node[x].fa =
43     y; }
44 void cut (int x, int y) { makeroot (x); access (y);
45     splay (y); node[node[y].son[0]].fa = 0; node[y].
46     son[0] = 0; update (y); }

```

### 2.1.2 Splay operation

```

1 void rotate (int x, int k) {
2     int y = a[x].fa;
3     if (~a[y].fa) a[a[y].fa].son[a[a[y].fa].son[1] == y]
4         = x;
5     a[x].fa = a[y].fa; a[y].son[k] = a[x].son[k ^ 1];
6     if (~a[x].son[k ^ 1]) a[a[x].son[k ^ 1]].fa = y;
7     a[y].fa = x; a[x].son[k ^ 1] = y;
8     update (y); update (x); }
9 void splay (int x, int s = -1) {
10    push_down (x);
11    while (a[x].fa != s) {
12        int y = a[x].fa, z = a[y].fa;
13        if (z == s) {
14            push_down (z = y); push_down (x);
15            rotate (x, a[y].son[1] == x);
16        } else {
17            push_down (z); push_down (y); push_down (x);
18            int t1 = a[y].son[1] == x, t2 = a[z].son[1] == y;
19            if (t1 == t2) rotate (y, t2), rotate (x, t1);
20            else rotate (x, t1), rotate (x, t2); } } }

```

## 2.2 KD tree

Find the  $k$ -th closest/farthest point in  $O(kn^{1-\frac{1}{k}})$ .

Usage:

1. Store the data in p[].

2. Execute init.

3. Execute min\_kth or max\_kth for queries ( $k$  is 1-based).

Note: Switch to the commented code for Manhattan distance.

```

1 template <int MAXN = 200000, int MAXK = 2>
2 struct kd_tree {
3     int k, size;
4     struct point { int data[MAXK], id; } p[MAXN];
5     struct kd_node {
6         int l, r; point p, dmin, dmax;
7         kd_node() {}
8         kd_node (const point &rhs) : l (-1), r (-1), p (rhs)
9             , dmin (rhs), dmax (rhs) {}
10    void merge (const kd_node &rhs, int k) {
11        for (register int i = 0; i < k; ++i) {
12            dmin.data[i] = std::min (dmin.data[i], rhs.dmin.
13                data[i]);
14            dmax.data[i] = std::max (dmax.data[i], rhs.dmax.
15                data[i]); } }
16    long long min_dist (const point &rhs, int k) const {
17        register long long ret = 0;
18        for (register int i = 0; i < k; ++i) {
19            if (dmin.data[i] <= rhs.data[i] && rhs.data[i] <=
20                dmax.data[i]) continue;
21            ret += std::min (1ll * (dmin.data[i] - rhs.data[i]
22                ) * (dmin.data[i] - rhs.data[i]),
23                1ll * (dmax.data[i] - rhs.data[i]) * (dmax.
24                    data[i] - rhs.data[i]));
25            // ret += std::max (0, rhs.data[i] - dmax.data[i])
26                + std::max (0, dmin.data[i] - rhs.data[i]);
27        } return ret; }
28    long long max_dist (const point &rhs, int k) {
29        long long ret = 0;
30        for (int i = 0; i < k; ++i) {
31            int tmp = std::max (std::abs (dmin.data[i] - rhs.
32                data[i]), std::abs (dmax.data[i] - rhs.data[i]
33                    ));
34            ret += 1ll * tmp * tmp; }
35        // ret += std::max (std::abs (rhs.data[i] - dmax.
36            data[i]) + std::abs (rhs.data[i] - dmin.data[i]));
37    } return ret; } } tree[MAXN * 4];
38 struct result {
39     long long dist; point d; result() {}
40     result (const long long &dist, const point &d) :
41         dist (dist), d (d) {}
42     bool operator > (const result &rhs) const { return
43         dist > rhs.dist || (dist == rhs.dist && d.id >
44             rhs.d.id); }
45     bool operator < (const result &rhs) const { return
46         dist < rhs.dist || (dist == rhs.dist && d.id <
47             rhs.d.id); } }
48 long long sqrdist (const point &a, const point &b) {
49     long long ret = 0;
50     for (int i = 0; i < k; ++i) ret += 1ll * (a.data[i]
51         - b.data[i]) * (a.data[i] - b.data[i]);
52     // for (int i = 0; i < k; ++i) ret += std::abs (a.
53         data[i] - b.data[i]);
54     return ret; }
55 int alloc() { tree[size].l = tree[size].r = -1;
56     return size++; }
57 void build (const int &depth, int &rt, const int &l,
58     const int &r) {
59     if (l > r) return;
60     register int middle = (l + r) >> 1;
61     std::nth_element (p + l, p + middle, p + r + 1, [=]
62         (const point &a, const point &b) { return a.
63             data[depth] < b.data[depth]; });
64     tree[rt] = alloc(); kd_node (p[middle]);
65     if (l == r) return;
66     build ((depth + 1) % k, tree[rt].l, l, middle - 1);
67     build ((depth + 1) % k, tree[rt].r, middle + 1, r);
68     if (~tree[rt].l) tree[rt].merge (tree[tree[rt].l], k);
69     if (~tree[rt].r) tree[rt].merge (tree[tree[rt].r], k);
70 }
71 std::priority_queue<result, std::vector<result>, std
72     ::less<result>> heap_l;
73 std::priority_queue<result, std::vector<result>, std
74     ::greater<result>> heap_r;
75 void min_kth (const int &depth, const int &rt, const
76     int &m, const point &d) {
77     result tmp = result (sqrdist (tree[rt].p, d), tree[
78         rt].p);
79     if ((int)heap_l.size() < m) heap_l.push (tmp);
80     else if (tmp < heap_l.top()) {
81         heap_l.pop();
82         heap_l.push (tmp); }
83     int x = tree[rt].l, y = tree[rt].r;
84     if (~x && ~y && sqrdist (d, tree[x].p) > sqrdist (d,
85         tree[y].p)) std::swap (x, y);
86     if (~x && ((int)heap_l.size() < m || tree[x].
87         min_dist (d, k) < heap_l.top().dist))
88         _min_kth ((depth + 1) % k, x, m, d);
89     if (~y && ((int)heap_l.size() < m || tree[y].
90         min_dist (d, k) < heap_l.top().dist))
91         _min_kth ((depth + 1) % k, y, m, d); }
92 void max_kth (const int &depth, const int &rt, const
93     int &m, const point &d) {
94     result tmp = result (sqrdist (tree[rt].p, d), tree[
95         rt].p);
96     if ((int)heap_r.size() < m) heap_r.push (tmp);
97     else if (tmp > heap_r.top()) {
98         heap_r.pop();
99         heap_r.push (tmp); }
100    int x = tree[rt].l, y = tree[rt].r;
101    if (~x && ~y && sqrdist (d, tree[x].p) < sqrdist (d

```

```

    , tree[y].p)) std::swap(x, y);
71 if (~x && ((int)heap_r.size() < m || tree[x].
    max_dist(d, k) >= heap_r.top().dist))
72   _max_kth((depth + 1) % k, x, m, d);
73 if (~y && ((int)heap_r.size() < m || tree[y].
    max_dist(d, k) >= heap_r.top().dist))
74   _max_kth((depth + 1) % k, y, m, d);
75 void init(int n, int k) { this->k = k; size = 0;
    int rt = 0; build(0, rt, 0, n - 1); }
76 result min_kth(const point &d, const int &m) {
    heap_l = decltype(heap_l)(); _min_kth(0, 0, m,
    d); return heap_l.top(); }
77 result max_kth(const point &d, const int &m) {
    heap_r = decltype(heap_r)(); _max_kth(0, 0, m,
    d); return heap_r.top(); }

```

## 2.3 RMQ

```

1 for(int st = 1; st < 20; ++st) for(int i = 0; i < N;
    ++i)
2   if(i + (1 << st - 1) < N) rmq[st][i] = std::min(rmq
    [st - 1][i], rmq[st - 1][i + (1 << st - 1)]);
3
4 int len = 31 - __builtin_clz(r - 1 + 1);
5 return std::min(rmq[len][l], rmq[len][r - (1 << len)
    + 1]);

```

## 3 Geometry

Generally  $\epsilon$  should be less than  $\frac{1}{xy}$ .

```

1 #define cd const double &
2 const double EPS = 1E-8, PI = acos(-1);
3 int sgn(cd x) { return x < -EPS ? -1 : x > EPS; }
4 int cmp(cd x, cd y) { return sgn(x - y); }
5 double sqr(cd x) { return x * x; }
6 double msqrt(cd x) { return sgn(x) <= 0 ? 0 : sqrt(x); }

```

### 3.1 2D geometry

1. point::rot90: Counter-clockwise rotation.
2. line.circle.intersect: In order of the direction of  $a$ .
3. circle.intersect: Counter-clockwise with respect of  $O_a$ .
4. tangent: Counter-clockwise with respect of  $a$ .
5. extangent: Counter-clockwise with respect of  $O_a$ .
6. intangent: Counter-clockwise with respect of  $O_a$ .

```

1 #define cp const point &
2 struct point {
3   double x, y;
4   explicit point(cd x = 0, cd y = 0) : x(x), y(y) {}
5   int dim() const { return sgn(y) == 0 ? sgn(x) > 0
6     : sgn(y) > 0; }
7   point unit() const { double l = msqrt(x * x + y * y)
8     ; return point(x / l, y / l); }
9   point rot90() const { return point(-y, x); }
10  point _rot90() const { return point(y, -x); }
11  point rot(cd t) const {
12    double c = cos(t), s = sin(t);
13    return point(x * c - y * s, x * s + y * c); }
14  bool operator == (cp a, cp b) { return cmp(a.x, b.x)
15    == 0 && cmp(a.y, b.y) == 0; }
16  bool operator != (cp a, cp b) { return cmp(a.x, b.x)
17    != 0 || cmp(a.y, b.y) != 0; }
18  bool operator < (cp a, cp b) { return cmp(a.x, b.x)
19    == 0 ? cmp(a.y, b.y) < 0 : cmp(a.x, b.x) < 0; }
20  point operator - (cp a) { return point(-a.x, -a.y); }
21  point operator + (cp a, cp b) { return point(a.x + b.
22    x, a.y + b.y); }
23  point operator - (cp a, cp b) { return point(a.x - b.
24    x, a.y - b.y); }
25  point operator * (cp a, cd b) { return point(a.x * b,
26    a.y * b); }
27  point operator / (cp a, cd b) { return point(a.x / b,
28    a.y / b); }
29  double dot(cp a, cp b) { return a.x * b.x + a.y * b.y
30    ; }
31  double det(cp a, cp b) { return a.x * b.y - a.y * b.x
32    ; }
33  double dis2(cp a, cp b = point()) { return sqr(a.x
34    - b.x) + sqr(a.y - b.y); }
35  double dis(cp a, cp b = point()) { return msqrt(
36    dis2(a, b)); }
37  #define cl const line &
38  struct line {
39    point s, t;
40    explicit line(cp s = point(), cp t = point()) : s
41      (s), t(t) {}
42    bool point_on_segment(cp a, cl b) { return sgn(det(
43      a - b.s, b.t - b.s)) == 0 && sgn(dot(b.s - a, b.
44      t - a)) <= 0; }
45    bool two_side(cp a, cp b, cl c) { return sgn(det(a
46      - c.s, c.t - c.s)) * sgn(det(b - c.s, c.t - c.s))
47      < 0; }
48    bool intersect_judgment(cl a, cl b) {
49      if(point_on_segment(b.s, a) || point_on_segment(b.
50      t, a)) return true;
51      if(point_on_segment(a.s, b) || point_on_segment(a.
52      t, b)) return true;
53      return two_side(a.s, a.t, b) && two_side(b.s, b.t,
54      a); }

```

```

34 point line_intersect(cl a, cl b) {
35   double s1 = det(a.t - a.s, b.s - a.s), s2 = det(a.t
36     - a.s, b.t - a.s);
37   return (b.s * s2 - b.t * s1) / (s2 - s1); }
38 double point_to_line(cp a, cl b) { return std::abs(
39   det(b.t - b.s, a - b.s)) / dis(b.s, b.t); }
40 point project_to_line(cp a, cl b) { return b.s + (b.t
41   - b.s) * (dot(a - b.s, b.t - b.s) / dis2(b.t, b.
42   .s)); }
43 double point_to_segment(cp a, cl b) {
44   if(sgn(dot(b.s - a, b.t - b.s)) * dot(b.t - a, b.t
45     - b.s)) <= 0) return std::abs(dot(b.t - b.s, a
46     - b.s)) / dis(b.s, b.t);
47   return std::min(dis(a, b.s), dis(a, b.t)); }
48 bool in_polygon(cp p, const std::vector<point> &po)
49   {
50     int n = (int)po.size(), counter = 0;
51     for(int i = 0; i < n; ++i) {
52       point a = po[i], b = po[(i + 1) % n];
53       // Modify the next line if necessary.
54       if(point_on_segment(p, line(a, b))) return true;
55       int x = sgn(det(p - a, b - a)), y = sgn(a.y - p.y)
56       , z = sgn(b.y - p.y);
57       if(x > 0 && y <= 0 && z > 0) counter++;
58       if(x < 0 && z <= 0 && y > 0) counter--; }
59     return counter != 0; }
60 double polygon_area(const std::vector<point> &a) {
61   double ans = 0.0;
62   for(int i = 0; i < (int)a.size(); ++i) ans += det
63     (a[i], a[(i + 1) % a.size()]) / 2.0;
64   return ans; }
65 #define cc const circle &
66 struct circle {
67   point c; double r;
68   explicit circle(point c = point(), double r = 0) :
69     c(c), r(r) {}
70   bool operator == (cc a, cc b) { return a.c == b.c &&
71     cmp(a.r, b.r) == 0; }
72   bool operator != (cc a, cc b) { return !(a == b); }
73   bool in_circle(cp a, cc b) { return cmp(dis(a, b.c)
74     , b.r) <= 0; }
75   circle make_circle(cp a, cp b) { return circle((a +
76     b) / 2, dis(a, b) / 2); }
77   circle make_circle(cp a, cp b, cp c) { point p =
78     circumcenter(a, b, c); return circle(p, dis(p,
79     a)); }
80   std::vector<point> line_circle_intersect(cl a, cc b)
81     {
82       if(cmp(point_to_line(b.c, a), b.r) > 0) return std
83         ::vector<point>();
84       double x = msqrt(sqr(b.r) - sqr(point_to_line(b.c
85         , a)));
86       point s = project_to_line(b.c, a), u = (a.t - a.s).
87         unit();
88       if(sgn(x) == 0) return std::vector<point>({s});
89       return std::vector<point>({s - u * x, s + u * x});
90     }
91   double circle_intersect_area(cc a, cc b) {
92     double d = dis(a.c, b.c);
93     if(sgn(d - (a.r + b.r)) >= 0) return 0;
94     if(sgn(d - std::abs(a.r - b.r)) <= 0) {
95       double r = std::min(a.r, b.r); return r * r * PI; }
96     double x = (d * d + a.r * a.r - b.r * b.r) / (2 * d),
97     t1 = acos(std::min(1., std::max(-1., x / a.r)
98     )), t2 = acos(std::min(1., std::max(-1., (d -
99     x) / b.r)));
100    return a.r * a.r * t1 + b.r * b.r * t2 - d * a.r *
101    sin(t1); }
102   std::vector<point> circle_intersect(cc a, cc b) {
103     if(a.c == b.c || cmp(dis(a.c, b.c), a.r + b.r) > 0
104     || cmp(dis(a.c, b.c), std::abs(a.r - b.r)) <
105     0) return std::vector<point>();
106     point r = (b.c - a.c).unit(); double d = dis(a.c, b
107     .c);
108     double x = ((sqr(a.r) - sqr(b.r)) / d + d) / 2, h =
109     msqrt(sqr(a.r) - sqr(x));
110     if(sgn(h) == 0) return std::vector<point>({a.c +
111     r * x});
112     return std::vector<point>({a.c + r * x - r.rot90()
113     * h, a.c + r * x + r.rot90() * h}); }
114   std::vector<point> tangent(cp a, cc b) { circle p =
115     make_circle(a, b.c); return circle_intersect(p,
116     b); }
117   std::vector<line> extangent(cc a, cc b) {
118     std::vector<line> ret;
119     if(cmp(dis(a.c, b.c), std::abs(a.r - b.r)) <= 0)
120       return ret;
121     if(sgn(a.r - b.r) == 0) {
122       point dir = b.c - a.c; dir = (dir * a.r / dis(dir))
123       .rot90();
124       ret.push_back(line(a.c - dir, b.c - dir));
125       ret.push_back(line(a.c + dir, b.c + dir));
126     } else {
127       point p = (b.c * a.r - a.c * b.r) / (a.r - b.r);
128       std::vector<point> pp = tangent(p, a), qq =
129       tangent(p, b);
130       if(pp.size() == 2 && qq.size() == 2) {
131         if(cmp(a.r, b.r) < 0) std::swap(pp[0], pp[1]),
132         std::swap(qq[0], qq[1]);
133         ret.push_back(line(pp[0], qq[0]));
134         ret.push_back(line(pp[1], qq[1]));
135       }
136       return ret; }
137   std::vector<line> intangent(cc a, cc b) {
138     std::vector<line> ret;

```



```

102 point p = (b.c * a.r + a.c * b.r) / (a.r + b.r);
103 std::vector<point> pp = tangent(p, a), qq = tangent
    (p, b);
104 if (pp.size() == 2 && qq.size() == 2) {
105     ret.push_back(line(pp[0], qq[0]));
106     ret.push_back(line(pp[1], qq[1]));
107 return ret; }

```

### 3.1.1 Convex hull

Counter-clockwise, starting with the smallest point, and with the minimum number of points. Modify `!s` to `s` in turn to conserve all points on the hull.

`convex_tan` finds the covering `[s..t]` of a certain point.

```

1 bool turn (cp a, cp b, cp c, int s) { return sgn (det
    (b - a, c - a)) != -s; }
2 std::pair<std::vector<point>, int> convex_hull (std
    ::vector<point> a) {
3     int cnt = 0; std::sort (a.begin(), a.end());
4     static std::vector<point> ret; ret.resize (a.size()
        << 1);
5     for (int i = 0; i < (int) a.size(); ++i) {
6         while (cnt > 1 && turn (ret[cnt - 2], a[i], ret[cnt
            - 1], 1)) --cnt;
7         ret[cnt++] = a[i];
8     int fixed = cnt;
9     for (int i = (int) a.size() - 1; i >= 0; --i) {
10        while (cnt > fixed && turn (ret[cnt - 2], a[i], ret[
            cnt - 1], 1)) --cnt;
11        ret[cnt++] = a[i];
12    return std::make_pair (std::vector<point> (ret.begin
        (), ret.begin() + cnt - 1), fixed - 1); }
13
14 int lb (cp x, const std::vector<point> &v, int l, int
    r, int s) {
15     if (l > r) l = r;
16     while (l != r) {
17         int m = (l + r) / 2;
18         if (sgn (det (v[m % v.size()] - x, v[(m + 1) % v.
            size()] - x)) == s) r = m; else l = m + 1; }
19     return r % v.size(); }
20 std::pair<int, int> convex_tan (cp x, const std::
    vector<point> &v, int rp) {
21     if (cmp (x.x, v[0].x) < 0) return std::make_pair (lb
        (x, v, rp, v.size(), -1), lb (x, v, 0, rp, 1));
22     else if (cmp (x.x, v[rp].x) > 0) return std::
        make_pair (lb (x, v, 0, rp, -1), lb (x, v, rp, v.
            size(), 1));
23     else {
24         int id = std::lower_bound (v.begin(), v.begin() +
            rp, x) - v.begin();
25         if (id == 0 || sgn (det (v[id - 1] - x, v[id] - x))
            < 0)
26             return std::make_pair (lb (x, v, 0, id, -1), lb (x,
                v, id, rp, 1));
27         id = std::lower_bound (v.begin() + rp, v.end(), x,
            std::greater<point>()) - v.begin();
28         if (id == rp || sgn (det (v[id - 1] - x, v[id] % v.
            size()) - x)) < 0)
29             return std::make_pair (lb (x, v, rp, id, -1), lb (x,
                v, id, v.size(), 1));
30     return std::make_pair (-1, -1); } }

```

### 3.1.2 Delaunay triangulation

In mathematics and computational geometry, a Delaunay triangulation (also known as a Delone triangulation) for a given set  $P$  of discrete points in a plane is a triangulation  $DT(P)$  such that no point in  $P$  is inside the circumcircle of any triangle in  $DT(P)$ . Delaunay triangulations maximize the minimum angle of all the angles of the triangles in the triangulation; they tend to avoid sliver triangles.

The Delaunay triangulation of a discrete point set  $P$  in general position corresponds to the dual graph of the Voronoi diagram for  $P$ . Special cases include the existence of three points on a line and four points on circle.

Properties: Let  $n$  be the number of points.

1. The union of all triangles in the triangulation is the convex hull of the points.
2. The Delaunay triangulation contains  $O(n)$  triangles.
3. If there are  $b$  vertices on the convex hull, then any triangulation of the points has at most  $2n - 2 - b$  triangles, plus one exterior face.
4. If points are distributed according to a Poisson process in the plane with constant intensity, then each vertex has on average six surrounding triangles.
5. In the plane, the Delaunay triangulation maximizes the minimum angle. Compared to any other triangulation of the points, the smallest angle in the Delaunay triangulation is at least as large as the smallest angle in any other. However, the Delaunay triangulation does not necessarily minimize the maximum angle. The Delaunay triangulation also does not necessarily minimize the length of the edges.
6. A circle circumscribing any Delaunay triangle does not contain any other input points in its interior.
7. If a circle passing through two of the input points doesn't contain any other of them in its interior, then the segment connecting the two points is an edge of a Delaunay triangulation of the given points.
8. Each triangle of the Delaunay triangulation of a set of points in  $d$ -dimensional spaces corresponds to a facet of convex hull of the projection of the points onto a  $(d + 1)$ -dimensional paraboloid, and vice versa.

9. The closest neighbor  $b$  to any point  $p$  is on an edge  $bp$  in the Delaunay triangulation since the nearest neighbor graph is a sub-graph of the Delaunay triangulation.
10. The Delaunay triangulation is a geometric spanner: the shortest path between two vertices, along Delaunay edges, is known to be no longer than  $\frac{4\pi}{3\sqrt{3}} \approx 2.418$  times the Euclidean distance between them.
11. The Euclidean minimum spanning tree of a set of points is a subset of the Delaunay triangulation of the same points, and this can be exploited to compute it efficiently.

Usage:

1. Initialize the coordinate range with `trig::LOTS`.
2. `trig::find`: Find the triangle that contains the given point.
3. `trig::add_point`: Add the point to the triangulation.
4. One certain triangle is in the triangulation if `tri::has_child()` == 0.
5. To find the neighbouring triangles of  $u$ , check `u.e[i].tri`, with vertex of the corresponding edge `u.p[(i + 1) % 3]` and `u.p[(i + 2) % 3]`.

```

1 const int N = 100000 + 5, MAX_TRIS = N * 6;
2 bool in_circumcircle (cp p1, cp p2, cp p3, cp p4) {
3     double u11 = p1.x - p4.x, u21 = p2.x - p4.x, u31 = p3
        .x - p4.x;
4     double u12 = p1.y - p4.y, u22 = p2.y - p4.y, u32 = p3
        .y - p4.y;
5     double u13 = sqr (p1.x) - sqr (p4.x) + sqr (p1.y) -
        sqr (p4.y);
6     double u23 = sqr (p2.x) - sqr (p4.x) + sqr (p2.y) -
        sqr (p4.y);
7     double u33 = sqr (p3.x) - sqr (p4.x) + sqr (p3.y) -
        sqr (p4.y);
8     double det = -u13 * u22 * u31 + u12 * u23 * u31 + u13
        * u21 * u32 - u11 * u23 * u32 - u12 * u21 * u33
        + u11 * u22 * u33;
9     return sgn (det) > 0; }
10 double side (cp a, cp b, cp p) { return (b.x - a.x) *
    (p.y - a.y) - (b.y - a.y) * (p.x - a.x); }
11 typedef int side_t; struct tri; typedef tri* tri_r;
12 struct edge {
13     tri_r t; side_t side;
14     edge (tri_r t = 0, side_t side = 0) : t(t), side(side)
        {} } ;
15 struct tri {
16     point p[3]; edge e[3]; tri_r child[3]; tri () {}
17     tri (cp p0, cp p1, cp p2) { p[0] = p0; p[1] = p1; p
        [2] = p2;
18     child[0] = child[1] = child[2] = 0; }
19     bool has_child() const { return child[0] != 0; }
20     int num_child() const { return child[0] == 0 ? 0 :
        child[1] == 0 ? 1 : child[2] == 0 ? 2 : 3; }
21     bool contains (cp q) const {
22         double a = side (p[0], p[1], q), b = side (p[1], p
            [2], q), c = side (p[2], p[0], q);
23         return sgn (a) >= 0 && sgn (b) >= 0 && sgn (c) >= 0;
        } } ;
24 void set_edge (edge a, edge b) {
25     if (a.t) a.t -> e[a.side] = b;
26     if (b.t) b.t -> e[b.side] = a; }
27 class trig {
28 public:
29     tri tpool[MAX_TRIS], *tot;
30     trig() { const double LOTS = 1E6;
31         the_root = new (tot++) tri (point (-LOTS, -LOTS),
32             point (LOTS, -LOTS), point (0, LOTS)); }
33     tri_r find (cp p) const { return find (the_root, p);
        }
34     void add_point (cp p) { add_point (find (the_root, p)
        , p); }
35 private:
36     tri_r the_root;
37     static tri_r find (tri_r root, cp p) {
38         for(;;) { if (!root -> has_child()) return root;
39             else for (int i = 0; i < 3 && root -> child[i]; ++
                i)
40                 if (root -> child[i] -> contains (p))
41                     { root = root->child[i]; break; } } }
42     void add_point (tri_r root, cp p) {
43         tri_r tab, tbc, tca;
44         tab = new (tot++) tri (root -> p[0], root -> p[1],
            p);
45         tbc = new (tot++) tri (root -> p[1], root -> p[2],
            p);
46         tca = new (tot++) tri (root -> p[2], root -> p[0],
            p);
47         set_edge (edge (tab, 0), edge (tbc, 1)); set_edge (
            edge (tbc, 0), edge (tca, 1));
48         set_edge (edge (tca, 0), edge (tab, 1)); set_edge (
            edge (tab, 2), root -> e[2]);
49         set_edge (edge (tbc, 2), root -> e[0]); set_edge (
            edge (tca, 2), root -> e[1]);
50         root -> child[0] = tab; root -> child[1] = tbc;
51         root -> child[2] = tca;
52         flip (tab, 2); flip (tbc, 2); flip (tca, 2); }
53     void flip (tri_r t, side_t pi) {
54         tri_r trj = t -> e[pi].t; int pj = t -> e[pi].side;
55         if (!trj || !in_circumcircle (t -> p[0], t -> p[1],
            t -> p[2], trj -> p[pj])) return;
56         tri_r trk = new (tot++) tri (t -> p[(pi + 1) % 3],
            trj -> p[pj], t -> p[pi]);
57         tri_r trl = new (tot++) tri (trj -> p[(pj + 1) %
            3], t -> p[pi], trj -> p[pj]);
58         set_edge (edge (trk, 0), edge (trl, 0));

```

```

57 set_edge (edge (trk, 1), t -> e[(pi + 2) % 3]);
   set_edge (edge (trk, 2), trj -> e[(pj + 1) %
58 3]);
   set_edge (edge (trl, 1), trj -> e[(pj + 2) % 3]);
   set_edge (edge (trl, 2), t -> e[(pi + 1) % 3]);
59 t -> child[0] = trk; t -> child[1] = trl; t ->
   child[2] = 0;
60 trj -> child[0] = trk; trj -> child[1] = trl; trj
   -> child[2] = 0;
61 flip (trk, 1); flip (trk, 2); flip (trl, 1); flip (
   trl, 2); } };
62 void build (std::vector<point> ps, trig &t) {
63 t.tot = t.tpool; std::random_shuffle (ps.begin (), ps
   .end ());
64 for (point &p : ps) t.add_point (p); }

```

### 3.1.3 Fermat point

Find a point  $P$  that minimizes  $|PA| + |PB| + |PC|$ .

```

1 point fermat_point (cp a, cp b, cp c) {
2 if (a == b) return a; if (b == c) return b; if (c ==
   a) return c;
3 double ab = dis (a, b), bc = dis (b, c), ca = dis (c,
   a);
4 double cosa = dot (b - a, c - a) / ab / ca;
5 double cosb = dot (a - b, c - b) / ab / bc;
6 double cosc = dot (b - c, a - c) / ca / bc;
7 double sq3 = PI / 3.0; point mid;
8 if (sgn (cosa + 0.5) < 0) mid = a;
9 else if (sgn (cosb + 0.5) < 0) mid = b;
10 else if (sgn (cosc + 0.5) < 0) mid = c;
11 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)));
12 else mid = line_intersect (line (a, c + (b - c).rot (
   sq3)), line (c, b + (a - b).rot (sq3)));
13 return mid; }

```

### 3.1.4 Half plane intersection

1. cut: Online in  $O(n^2)$ .
2. half\_plane\_intersect: Offline in  $O(m \log m)$ .

```

1 std::vector<point> cut (const std::vector<point> &c,
   line p) {
2 std::vector<point> ret;
3 if (c.empty ()) return ret;
4 for (int i = 0; i < (int) c.size (); ++i) {
5 int j = (i + 1) % (int) c.size ();
6 if (turn_left (p.s, p.t, c[i])) ret.push_back (c[i])
   ;
7 if (two_side (c[i], c[j], p)) ret.push_back (
   line_intersect (p, line (c[i], c[j]))); }
8 return ret; }
9 bool turn_left (cl l, cp p) { return sgn (det (l.t - l
   .s, p - l.s)) >= 0; }
10 int cmp (cp a, cp b) { return a.dim () != b.dim () ? (
   a.dim () < b.dim () ? -1 : 1) : -sgn (det (a, b)); }
11 std::vector<point> half_plane_intersect (std::vector
   <line> h) {
12 typedef std::pair<point, line> polar;
13 std::vector<polar> g; g.resize (h.size ());
14 for (int i = 0; i < (int) h.size (); ++i) g[i] = std
   ::make_pair (h[i].t - h[i].s, h[i]);
15 sort (g.begin (), g.end (), [&] (const polar &a,
   const polar &b) {
16 if (cmp (a.first, b.first) == 0) return sgn (det (a.
   second.t - a.second.s, b.second.t - a.second.s))
   < 0;
17 else return cmp (a.first, b.first) < 0; });
18 h.resize (std::unique (g.begin (), g.end (), [] (
   const polar &a, const polar &b) { return cmp (a.
   first, b.first) == 0; }) - g.begin ());
19 for (int i = 0; i < (int) h.size (); ++i) h[i] = g[i
   ].second;
20 int fore = 0, rear = -1; std::vector<line> ret (h.
   size (), line ());
21 for (int i = 0; i < (int) h.size (); ++i) {
22 while (fore < rear && !turn_left (h[i],
   line_intersect (ret[rear - 1], ret[rear]))) --
   rear;
23 while (fore < rear && !turn_left (h[i],
   line_intersect (ret[fore], ret[fore + 1]))) ++
   fore;
24 ret[++rear] = h[i]; }
25 while (rear - fore > 1 && !turn_left (ret[fore],
   line_intersect (ret[rear - 1], ret[rear]))) --
   rear;
26 while (rear - fore > 1 && !turn_left (ret[rear],
   line_intersect (ret[fore], ret[fore + 1]))) ++
   fore;
27 if (rear - fore < 2) return std::vector<point> ();
28 std::vector<point> ans; ans.resize (rear + 1);
29 for (int i = 0; i < rear + 1; ++i) ans[i] =
   line_intersect (ret[i], ret[(i + 1) % (rear + 1)
   ]);
30 return ans; }

```

### 3.1.5 Intersection of a polygon and a circle

```

1 struct polygon_circle_intersect {
2 double sector_area (cp a, cp b, const double &r) {

```

```

3 double c = (2.0 * r * r - dis2 (a, b)) / (2.0 * r *
   r);
4 return r * r * acos (c) / 2.0; }
5 double area (cp a, cp b, const double &r) {
6 double dA = dot (a, a), dB = dot (b, b), dC =
   point_to_segment (point (), line (a, b));
7 if (sgn (dA - r * r) <= 0 && sgn (dB - r * r) <= 0)
   return det (a, b) / 2.0;
8 point tA = a.unit () * r, tB = b.unit () * r;
9 if (sgn (dC - r) > 0) return sector_area (tA, tB, r)
   ;
10 std::vector<point> ret = line_circle_intersect (
   line (a, b), circle (point (), r));
11 if (sgn (dA - r * r) > 0 && sgn (dB - r * r) > 0)
   return sector_area (tA, ret[0], r) + det (ret[0],
   ret[1]) / 2.0 + sector_area (ret[1], tB, r);
12 if (sgn (dA - r * r) > 0) return det (ret[0], b) /
   2.0 + sector_area (tA, ret[0], r);
13 else return det (a, ret[1]) / 2.0 + sector_area (ret
   [1], tB, r); }
14 double solve (const std::vector<point> &p, cc c) {
15 double ret = 0.0;
16 for (int i = 0; i < (int) p.size (); ++i) {
17 int s = sgn (det (p[i] - c.c, p[(i + 1) % p.size ()
   ] - c.c));
18 if (s > 0) ret += area (p[i] - c.c, p[(i + 1) % p.
   size ()] - c.c, c.r);
19 else ret -= area (p[(i + 1) % p.size ()] - c.c, p[i
   ] - c.c, c.r); }
20 return std::abs (ret); } }

```

### 3.1.6 Minimum circle

```

1 circle minimum_circle (std::vector<point> p) {
2 circle ret; std::random_shuffle (p.begin (), p.end ()
   );
3 for (int i = 0; i < (int) p.size (); ++i) if (!
   in_circle (p[i], ret)) {
4 ret = circle (p[i], 0); for (int j = 0; j < i; ++j)
   if (!in_circle (p[j], ret)) {
5 ret = make_circle (p[j], p[i]); for (int k = 0; k <
   j; ++k)
6 if (!in_circle (p[k], ret)) ret = make_circle (p[i
   ], p[j], p[k]); } }
7 return ret; }

```

### 3.1.7 Nearest pair of points

Solve in range  $[l, r]$ . Necessary to sort  $p[]$  first. Complexity  $O(n \log n)$ .

```

1 double solve (std::vector<point> &p, int l, int r) {
2 if (l + 1 >= r) return INF;
3 int m = (l + r) / 2; double mx = p[m].x; std::vector
   <point> v;
4 double ret = std::min (solve(p, l, m), solve(p, m, r)
   );
5 for (int i = l; i < r; ++i)
6 if (sqr (p[i].x - mx) < ret) v.push_back (p[i]);
7 sort (v.begin (), v.end (), [&] (cp a, cp b) { return
   a.y < b.y; });
8 for (int i = 0; i < v.size (); ++i)
9 for (int j = i + 1; j < v.size (); ++j) {
10 if (sqr (v[i].y - v[j].y) > ret) break;
11 ret = min (ret, dis2 (v[i] - v[j])); }
12 return ret; }

```

### 3.1.8 Triangle center

Trilinear coordinates:

1. incenter:  $1:1:1$ .
2. centroid:  $bc:ca:ab$ .
3. circumcenter:  $\cos A:\cos B:\cos C$ .
4. orthocenter:  $\sec A:\sec B:\sec C$ .
5. Non-trivial Fermat point:  $\csc(A + \pi/3):\csc(B + \pi/3):\csc(C + \pi/3)$ .

```

1 point incenter (cp a, cp b, cp c) {
2 double p = dis (a, b) + dis (b, c) + dis (c, a);
3 return (a * dis (b, c) + b * dis (c, a) + c * dis (a,
   b)) / p; }
4 point circumcenter (cp a, cp b, cp c) {
5 point p = b - a, q = c - a, s (dot (p, p) / 2, dot (q
   , q) / 2);
6 return a + point (det (s, point (p.y, q.y)), det (
   point (p.x, q.x), s)) / det (p, q); }
7 point orthocenter (cp a, cp b, cp c) { return a + b +
   c - circumcenter (a, b, c) * 2; }

```

### 3.1.9 Union of circles

```

1 template<int MAXN = 500> struct union_circle {
2 int C; circle c[MAXN]; double area[MAXN];
3 struct event {
4 point p; double ang; int delta;
5 event (cp p = point (), double ang = 0, int delta =
   0) : p(p), ang(ang), delta(delta) {}
6 bool operator < (const event &a) { return ang < a.
   ang; }
7 };
8 void addevent(cc a, cc b, std::vector<event> &evt,
   int &cnt) {
9 double d2 = dis2 (a.c, b.c), d_ratio = ((a.r - b.r)
   * (a.r + b.r) / d2 + 1) / 2,

```

```

10 p_ratio = msqrt (std::max (0., -(d2 - sqr(a.r - b.r
11 )) * (d2 - sqr(a.r + b.r)) / (d2 * d2 * 4)));
12 point d = b.c - a.c, p = d.rot(PI / 2), q0 = a.c + d
13 * d_ratio + p * p_ratio, q1 = a.c + d * d_ratio
14 - p * p_ratio;
15 double ang0 = atan2 ((q0 - a.c).y, (q0 - a.c).x),
16 ang1 = atan2 ((q1 - a.c).x, (q1 - a.c).y);
17 evt.emplace_back(q1, ang1, 1); evt.emplace_back(q0,
18 ang0, -1); cnt += ang1 > ang0; }
19 bool same(cc a, cc b) { return sgn (dis (a.c, b.c))
20 == 0 && sgn (a.r - b.r) == 0; }
21 bool overlap(cc a, cc b) { return sgn (a.r - b.r -
22 dis (a.c, b.c)) >= 0; }
23 bool intersect(cc a, cc b) { return sgn (dis (a.c, b.
24 c) - a.r - b.r) < 0; }
25 void solve() {
26 std::fill (area, area + C + 2, 0);
27 for (int i = 0; i < C; ++i) {
28 int cnt = 1; std::vector <event> evt;
29 for (int j = 0; j < i; ++j) if (same (c[i], c[j]))
30 ++cnt;
31 for (int j = 0; j < C; ++j) if (j != i && !same (c[
32 i], c[j]) && overlap (c[j], c[i])) ++cnt;
33 for (int j = 0; j < C; ++j) if (j != i && !overlap
34 (c[j], c[i]) && !overlap (c[i], c[j]) &&
35 intersect (c[i], c[j]))
36 addevent (c[i], c[j], evt, cnt);
37 if (evt.empty ()) area[cnt] += PI * c[i].r * c[i].r
38 ;
39 else {
40 std::sort (evt.begin (), evt.end ());
41 evt.push_back (evt.front ());
42 for (int j = 0; j + 1 < (int) evt.size (); ++j) {
43 cnt += evt[j].delta; area[cnt] += det (evt[j].p,
44 evt[j + 1].p) / 2;
45 double ang = evt[j + 1].ang - evt[j].ang; if (ang
46 < 0) ang += PI * 2;
47 area[cnt] += ang * c[i].r * c[i].r / 2 - sin(ang)
48 * c[i].r * c[i].r / 2; } } } } }

```

## 3.2 3D geometry

### 3.2.1 3D point

rotate: Right-hand rule with right-handed coordinates.

```

1 #define cp3 const point3 &
2 struct point3 {
3 double x, y, z;
4 explicit point3 (cd x = 0, cd y = 0, cd z = 0) : x (x
5 ), y (y), z (z) {} }
6 point3 operator + (cp3 a, cp3 b) { return point3 (a.x
7 + b.x, a.y + b.y, a.z + b.z); }
8 point3 operator - (cp3 a, cp3 b) { return point3 (a.x
9 - b.x, a.y - b.y, a.z - b.z); }
10 point3 operator * (cp3 a, cd b) { return point3 (a.x *
11 b, a.y * b, a.z * b); }
12 point3 operator / (cp3 a, cd b) { return point3 (a.x /
13 b, a.y / b, a.z / b); }
14 double dot (cp3 a, cp3 b) { return a.x * b.x + a.y * b
15 .y + a.z * b.z; }
16 point3 det (cp3 a, cp3 b) { return point3 (a.y * b.z -
17 a.z * b.y, -a.x * b.z + a.z * b.x, a.x * b.y - a.
18 y * b.x); }
19 double dis2 (cp3 a, cp3 b = point3 ()) { return sqr (a
20 .x - b.x) + sqr (a.y - b.y) + sqr (a.z - b.z); }
21 double dis (cp3 a, cp3 b = point3 ()) { return msqrt (
22 dis2 (a, b)); }
23 point3 rotate(cp3 p, cp3 axis, double w) {
24 double x = axis.x, y = axis.y, z = axis.z;
25 double s = x * x + y * y + z * z, ss = msqrt(s), cosw
26 = cos(w), sinw = sin(w);
27 double a[4][4]; memset (a, 0, sizeof (a));
28 a[3][3] = 1;
29 a[0][0] = ((y * y + z * z) * cosw + x * x) / s;
30 a[0][1] = x * y * (1 - cosw) / s + z * sinw / ss;
31 a[0][2] = x * z * (1 - cosw) / s - y * sinw / ss;
32 a[1][0] = x * y * (1 - cosw) / s - z * sinw / ss;
33 a[1][1] = ((x * x + z * z) * cosw + y * y) / s;
34 a[1][2] = y * z * (1 - cosw) / s + x * sinw / ss;
35 a[2][0] = x * z * (1 - cosw) / s + y * sinw / ss;
36 a[2][1] = y * z * (1 - cosw) / s - x * sinw / ss;
37 a[2][2] = ((x * x + y * y) * cosw + z * z) / s;
38 double ans[4] = {0, 0, 0, 0}, c[4] = {p.x, p.y, p.z,
39 1};
40 for (int i = 0; i < 4; ++i) for (int j = 0; j < 4; ++
41 j)
42 ans[i] += a[j][i] * c[j];
43 return point3 (ans[0], ans[1], ans[2]);
44 }

```

### 3.2.2 3D line

```

1 #define cl3 const line3 &
2 struct line3 {
3 point3 s, t;
4 explicit line3 (cp3 s = point3 (), cp3 t = point3 ())
5 : s (s), t (t) {} }
6 point3 line_plane_intersection (cl3 a, cl3 b) { return
7 a.s + (a.t - a.s) * dot (b.s - a.s, b.t - b.s) /
8 dot (a.t - a.s, b.t - b.s); }
9 line3 plane_intersection (cl3 a, cl3 b) {
10 point3 p = det (a.t - a.s, b.t - b.s), q = det (a.t -
11 a.s, p), s = line_plane_intersection (line3 (a.s
12 , a.s + q), b);

```

```

8 return line3 (s, s + p); }
9 point3 project_to_plane (cp3 a, cl3 b) { return a + (b
10 .t - b.s) * dot (b.t - b.s, b.s - a) / dis2 (b.t -
11 b.s); }

```

### 3.2.3 3D convex hull

Input  $n$  and  $p$ . Return *face*.

Note: Remove coincide points first.

```

1 template <int MAXN = 500>
2 struct convex_hull3 {
3 double mix (cp3 a, cp3 b, cp3 c) { return dot (det (a
4 , b), c); }
5 double volume (cp3 a, cp3 b, cp3 c, cp3 d) { return
6 mix (b - a, c - a, d - a); }
7 struct tri {
8 int a, b, c; tri() {}
9 tri(int _a, int _b, int _c): a(_a), b(_b), c(_c) {}
10 double area() const { return dis (det (p[b] - p[a],
11 p[c] - p[a])) / 2; }
12 point3 normal() const { return det (p[b] - p[a], p[c]
13 - p[a]).unit (); }
14 double dis (cp3 p0) const { return dot (normal (),
15 p0 - p[a]); } }
16 int n; std::vector <point3> p;
17 std::vector <tri> face; tmp;
18 int mark[MAXN][MAXN], time;
19 void add (int v) {
20 ++time; tmp.clear ();
21 for (int i = 0; i < (int) face.size (); ++i) {
22 int a = face[i].a, b = face[i].b, c = face[i].c;
23 if (sgn (volume (p[v], p[a], p[b], p[c])) > 0)
24 mark[a][b] = mark[b][a] = mark[a][c] = mark[c][a]
25 = mark[b][c] = mark[c][b] = time;
26 else tmp.push_back (face[i]); }
27 face.clear (); face = tmp;
28 for (int i = 0; i < (int) tmp.size (); ++i) {
29 int a = face[i].a, b = face[i].b, c = face[i].c;
30 if (mark[a][b] == time) face.emplace_back (v, b, a)
31 ;
32 if (mark[b][c] == time) face.emplace_back (v, c, b)
33 ;
34 if (mark[c][a] == time) face.emplace_back (v, a, c)
35 ; } }
36 void reorder () {
37 for (int i = 2; i < n; ++i) {
38 point3 tmp = det (p[i] - p[0], p[i] - p[1]);
39 if (sgn (dis (tmp))) {
40 std::swap (p[i], p[2]);
41 for (int j = 3; j < n; ++j)
42 if (sgn (volume (p[0], p[1], p[2], p[j]))) {
43 std::swap (p[j], p[3]); return; } } } }
44 void build_convex () {
45 reorder (); face.clear ();
46 face.emplace_back (0, 1, 2);
47 face.emplace_back (0, 2, 1);
48 for (int i = 3; i < n; ++i) add(i); } }

```

## 4 Graph

```

1 template <int MAXN = 100000, int MAXM = 100000>
2 struct edge_list {
3 int size, begin[MAXN], dest[MAXM], next[MAXM];
4 void clear (int n) { size = 0; std::fill (begin,
5 begin + n, -1); }
6 edge_list (int n = MAXN) { clear (n); }
7 void add_edge (int u, int v) { dest[size] = v; next[
8 size] = begin[u]; begin[u] = size++; } }
9 template <int MAXN = 100000, int MAXM = 100000>
10 struct cost_edge_list {
11 int size, begin[MAXN], dest[MAXM], next[MAXM], cost[
12 MAXM];
13 void clear (int n) { size = 0; std::fill (begin,
14 begin + n, -1); }
15 cost_edge_list (int n = MAXN) { clear (n); }
16 void add_edge (int u, int v, int c) { dest[size] = v;
17 next[size] = begin[u]; cost[size] = c; begin[u]
18 = size++; } }

```

### 4.1 Characteristic

#### 4.1.1 Chordal graph

A chordal graph is one in which all cycles of four or more vertices have a chord, which is an edge that is not part of the cycle but connects two vertices of the cycle.

A perfect elimination ordering in a graph is an ordering of the vertices of the graph such that, for each vertex  $v$ ,  $v$  and the neighbors of  $v$  that occur after  $v$  in the order form a clique. A graph is chordal if and only if it has a perfect elimination ordering. One application of perfect elimination orderings is finding a maximum clique of a chordal graph in polynomial-time, while the same problem for general graphs is NP-complete. More generally, a chordal graph can have only linearly many maximal cliques, while non-chordal graphs may have exponentially many. To list all maximal cliques of a chordal graph, simply find a perfect elimination ordering, form a clique for each vertex  $v$  together with the neighbors of  $v$  that are later than  $v$  in the perfect elimination ordering, and test whether each of the resulting cliques is maximal.

The largest maximal clique is a maximum clique, and, as chordal graphs are perfect, the size of this clique equals the chromatic number of the chordal graph. Chordal graphs are perfectly orderable: an optimal coloring may be obtained by applying a greedy coloring algorithm to the vertices in the reverse of a perfect elimination ordering.



In any graph, a vertex separator is a set of vertices the removal of which leaves the remaining graph disconnected; a separator is minimal if it has no proper subset that is also a separator. Chordal graphs are graphs in which each minimal separator is a clique.

Usage:

1. Set  $n$  and  $e$ .
2. Call `init` to obtain the perfect elimination ordering in seq.
3. Use `is_chordal` to test whether the graph is chordal.
4. Use `min_color` to obtain the size of the maximum clique (and the chromatic number).

```
1 template <int MAXN = 100000, int MAXM = 100000>
2 struct chordal_graph {
3     int n; edge_list <MAXN, MAXM> e;
4     int id[MAXN], seq[MAXN];
5     void init () {
6         struct point {
7             int lab, u;
8             point (int lab = 0, int u = 0) : lab (lab), u (u)
9             {}
10        bool operator < (const point &a) const { return lab
11            < a.lab; } };
12        std::fill (id, id + n, -1);
13        static int label[MAXN]; std::fill (label, label + n,
14            0);
15        std::priority_queue <point> q;
16        for (int i = 0; i < n; ++i) q.push (point (0, i));
17        for (int i = n - 1; i >= 0; --i) {
18            for (; ~id[q.top ().u];) q.pop ();
19            int u = q.top ().u; q.pop (); id[u] = i;
20            for (int j = e.begin[u], v; ~j; j = e.next[j])
21                if (v = e.dest[j], !~id[v]) ++label[v], q.push (
22                    point (label[v], v)); }
23        for (int i = 0; i < n; ++i) seq[id[i]] = i; }
24 bool is_chordal () {
25     static int vis[MAXN], q[MAXN]; std::fill (vis, vis +
26         n, -1);
27     for (int i = n - 1; i >= 0; --i) {
28         int u = seq[i], t = 0, v;
29         for (int j = e.begin[u]; ~j; j = e.next[j])
30             if (v = e.dest[j], id[v] > id[u]) q[t++] = v;
31         if (!t) continue; int w = q[0];
32         for (int j = 0; j < t; ++j) if (id[q[j]] < id[w]) w
33             = q[j];
34         for (int j = e.begin[w]; ~j; j = e.next[j]) vis[e.
35             dest[j]] = i;
36         for (int j = 0; j < t; ++j) if (q[j] != w && vis[q[
37             j]] != i) return 0;
38     }
39     return 1; }
40 int min_color () {
41     int res = 0;
42     static int vis[MAXN], c[MAXN];
43     std::fill (vis, vis + n, -1);
44     std::fill (c, c + n, n);
45     for (int i = n - 1; i >= 0; --i) {
46         int u = seq[i];
47         for (int j = e.begin[u]; ~j; j = e.next[j]) vis[c[e.
48             dest[j]]] = i;
49         int k; for (k = 0; vis[k] == i; ++k);
50         c[u] = k; res = std::max (res, k + 1);
51     }
52     return res; } };
```

#### 4.1.2 Euler characteristic

The Euler characteristic  $\chi$  was classically defined for the surfaces of polyhedra, according to the formula

$$\chi = V - E + F$$

where  $V$ ,  $E$ , and  $F$  are respectively the numbers of vertices (corners), edges and faces in the given polyhedron. Any convex polyhedron's surface has Euler characteristic

$$V - E + F = 2.$$

This equation is known as Euler's polyhedron formula. It corresponds to the Euler characteristic of the sphere (i.e.  $\chi = 2$ ), and applies identically to spherical polyhedra.

The Euler characteristic of a closed orientable surface can be calculated from its genus  $g$  (the number of tori in a connected sum decomposition of the surface; intuitively, the number of "handles") as

$$\chi = 2 - 2g.$$

The Euler characteristic of a closed non-orientable surface can be calculated from its non-orientable genus  $k$  (the number of real projective planes in a connected sum decomposition of the surface) as

$$\chi = 2 - k.$$

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

$$v - e + f = 2.$$

In a finite, connected, simple, planar graph, any face (except possibly the outer one) is bounded by at least three edges and every edge touches at most two faces; using Euler's formula, one can then show that these graphs are sparse in the sense that if  $v \geq 3$ :

$$e \leq 3v - 6.$$

## 4.2 Clique

### 4.2.1 DN maximum clique

Find the maximum clique ( $n \leq 150$ ).

Example:

```
1 BB e[N]; int ans, sol[N]; for (...) e[x][y] = e[y][x]
2 = true;
3 max_clique mc (e, n); mc.mcqdyn (sol, ans); //0-based.
4 for (int i = 0; i < ans; ++i) std::cout << sol[i] <<
5 std::endl;
```

```
1 typedef bool BB[N]; struct max_clique {
2     const BB *e; int pk, level; const float Tlimit;
3     struct vertex { int i, d; vertex (int i) : i(i), d(0)
4         {} };
5     typedef std::vector <vertex> vertices; vertices V;
6     typedef std::vector <int> colors; colors QMAX, Q;
7     std::vector <colors> C;
8     static bool desc_degree (const vertex &vi, const vertex
9         &vj) { return vi.d > vj.d; }
10    void init_colors (vertices &v) {
11        const int max_degree = v[0].d;
12        for (int i = 0; i < (int) v.size(); ++i) v[i].d = std
13            ::min (i, max_degree) + 1; }
14    void set_degrees (vertices &v) {
15        for (int i = 0, j; i < (int) v.size(); ++i)
16            for (v[i].d = j = 0; j < (int) v.size(); ++j)
17                v[i].d += e[v[i].i][v[j].i]; }
18    struct steps { int i1, i2; steps () : i1 (0), i2 (0) {}
19        };
20    std::vector <steps> S;
21    bool cut1 (const int pi, const colors &A) {
22        for (int i = 0; i < (int) A.size(); ++i)
23            if (e[pi][A[i]]) return true; return false; }
24    void cut2 (const vertices &A, vertices &B) {
25        for (int i = 0; i < (int) A.size(); ++i)
26            if (e[A.back ().i][A[i].i]) B.push_back (A[i].i); }
27    void color_sort (vertices &R) {
28        int j = 0, maxno = 1, min_k = std::max ((int) QMAX.
29            size () - (int) Q.size () + 1, 1);
30        C[1].clear (); C[2].clear ();
31        for (int i = 0; i < (int) R.size(); ++i) {
32            int pi = R[i].i, k = 1; while (cut1(pi, C[k])) ++k;
33            if (k > maxno) maxno = k, C[maxno + 1].clear ();
34            C[k].push_back (pi); if (k < min_k) R[j++] = pi; }
35        if (j > 0) R[j - 1].d = 0;
36        for (int k = min_k; k <= maxno; ++k)
37            for (int i = 0; i < (int) C[k].size(); ++i)
38                R[j].i = C[k][i], R[j++].d = k; }
39    void expand_dyn (vertices &R) {
40        S[level].i1 = S[level].i1 + S[level - 1].i1 - S[level
41            ].i2;
42        S[level].i2 = S[level - 1].i1;
43        while ((int) R.size () > 0) {
44            if ((int) Q.size () + R.back ().d > (int) QMAX.size
45                ()) {
46                Q.push_back (R.back ().i); vertices Rp; cut2 (R, Rp
47                    );
48                if ((int) Rp.size ()) {
49                    if ((float) S[level].i1 / ++pk < Tlimit)
50                        degree_sort (Rp);
51                        color_sort (Rp); ++S[level].i1, ++level;
52                        expand_dyn (Rp); --level;
53                    } else if ((int) Q.size () > (int) QMAX.size ()
54                        )
55                        QMAX = Q;
56                Q.pop_back (); } else return; R.pop_back (); } }
57    void mcqdyn (int *maxclique, int &sz) {
58        set_degrees (V); std::sort (V.begin (), V.end (),
59            desc_degree); init_colors (V);
60        for (int i = 0; i < (int) V.size (); ++i) S[i].i1
61            = S[i].i2 = 0;
62        expand_dyn (V); sz = (int) QMAX.size ();
63        for (int i = 0; i < (int) QMAX.size (); ++i)
64            maxclique[i] = QMAX[i]; }
65    void degree_sort (vertices &R) {
66        set_degrees (R); std::sort (R.begin (), R.end (),
67            desc_degree); }
68    max_clique (const BB *conn, const int sz, const float
69        tt = .025) : pk (0), level (1), Tlimit (tt) {
70        for (int i = 0; i < sz; ++i) V.push_back (vertex (i));
71        e = conn, C.resize (sz + 1), S.resize (sz + 1); } }
72 BB e[N]; int ans, sol[N]; for (...) e[x][y] = e[y][x]
73 = true;
74 max_clique mc (e, n); mc.mcqdyn (sol, ans); //0-based.
75 for (int i = 0; i < ans; ++i) std::cout << sol[i] <<
76 std::endl;
```

## 4.3 Cut

### 4.3.1 2-SAT

In terms of the implication graph, two literals belong to the same strongly connected component whenever there exist chains of implications from one literal to the other and vice versa. Therefore, the two literals must have the same value in any satisfying assignment to the given 2-satisfiability instance. In particular, if a variable and its negation both belong to the same strongly connected component, the instance cannot be satisfied, because it is impossible to assign both of these literals the same value. As Aspvall et al. showed, this is a necessary and sufficient condition: a 2-CNF formula is satisfiable if and only if there is no variable that belongs to the same strongly connected component as its negation.



This immediately leads to a linear time algorithm for testing satisfiability of 2-CNF formulae: simply perform a strong connectivity analysis on the implication graph and check that each variable and its negation belong to different components. However, as Aspvall et al. also showed, it also leads to a linear time algorithm for finding a satisfying assignment, when one exists. Their algorithm performs the following steps:

Construct the implication graph of the instance, and find its strongly connected components using any of the known linear-time algorithms for strong connectivity analysis.

Check whether any strongly connected component contains both a variable and its negation. If so, report that the instance is not satisfiable and halt.

Construct the condensation of the implication graph, a smaller graph that has one vertex for each strongly connected component, and an edge from component  $i$  to component  $j$  whenever the implication graph contains an edge  $uv$  such that  $u$  belongs to component  $i$  and  $v$  belongs to component  $j$ . The condensation is automatically a directed acyclic graph and, like the implication graph from which it was formed, it is skew-symmetric.

Topologically order the vertices of the condensation. In practice this may be efficiently achieved as a side effect of the previous step, as components are generated by Kosaraju's algorithm in topological order and by Tarjan's algorithm in reverse topological order.

For each component in the reverse topological order, if its variables do not already have truth assignments, set all the literals in the component to be true. This also causes all of the literals in the complementary component to be set to false.

### 4.3.2 Dominator tree

Find the immediate dominator ( $\text{idom}[]$ ) of each node,  $\text{idom}[x]$  will be  $x$  if  $x$  does not have a dominator, and will be  $-1$  if  $x$  is not reachable from  $s$ .

```
1 template <int MAXN = 100000, int MAXM = 100000>
2 struct dominator_tree {
3     int dfn[MAXN], sdom[MAXN], idom[MAXN], id[MAXN], f[
4         MAXN], fa[MAXN], smin[MAXN], stamp;
5     void predfs (int x, const edge_list <MAXN, MAXM> &
6         succ) {
7         id[dfn[x] = stamp++] = x;
8         for (int i = succ.begin[x]; ~i; i = succ.next[i]) {
9             int y = succ.dest[i];
10            if (dfn[y] < 0) { f[y] = x; predfs (y, succ); } } }
11 int getfa (int x) {
12     if (fa[x] == x) return x;
13     int ret = getfa (fa[x]);
14     if (dfn[sdom[smin[fa[x]]]] < dfn[sdom[smin[x]]])
15         smin[x] = smin[fa[x]];
16     return fa[x] = ret; }
17 void solve (int s, int n, const edge_list <MAXN, MAXM>
18     & succ) {
19     std::fill (dfn, dfn + n, -1); std::fill (idom, idom
20         + n, -1);
21     static edge_list <MAXN, MAXM> pred, tmp; pred.clear
22         (n);
23     for (int i = 0; i < n; ++i) for (int j = succ.begin[
24         i]; ~j; j = succ.next[j]) {
25         pred.add_edge (succ.dest[j], i);
26         stamp = 0; tmp.clear (n); predfs (s, succ);
27         for (int i = 0; i < stamp; ++i) fa[id[i]] = smin[id[
28             i]] = id[i];
29         for (int o = stamp - 1; o >= 0; --o) {
30             int x = id[o];
31             if (o) {
32                 sdom[x] = f[x];
33                 for (int i = pred.begin[x]; ~i; i = pred.next[i]) {
34                     int p = pred.dest[i];
35                     if (dfn[p] < 0) continue;
36                     if (dfn[p] > dfn[x]) { getfa (p); p = sdom[smin[p]
37                         ]; }
38                     if (dfn[sdom[x]] > dfn[p]) sdom[x] = p; }
39                 tmp.add_edge (sdom[x], x); }
40             while (~tmp.begin[x]) {
41                 int y = tmp.dest[tmp.begin[x]];
42                 tmp.begin[x] = tmp.next[tmp.begin[x]]; getfa (y);
43                 if (x != sdom[smin[y]]) idom[y] = smin[y];
44                 else idom[y] = x; }
45             for (int v : succ[x]) if (f[v] == x) fa[v] = x; }
46         idom[s] = s; for (int i = 1; i < stamp; ++i) {
47             int x = id[i]; if (idom[x] != sdom[x]) idom[x] =
48                 idom[idom[x]]; } } }
```

### 4.3.3 Stoer Wagner algorithm

Find the minimum cut of an undirected graph (1-based).

```
1 template <int MAXN = 500>
2 struct stoer_wagner {
3     int n, edge[MAXN][MAXN];
4     int dist[MAXN];
5     bool vis[MAXN], bin[MAXN];
6     stoer_wagner () {
7         memset (edge, 0, sizeof (edge));
8         memset (bin, false, sizeof (bin)); }
9     int contract (int &s, int &t) {
10        memset (dist, 0, sizeof (dist));
11        memset (vis, false, sizeof (vis));
12        int i, j, k, mincut, maxc;
13        for (i = 1; i <= n; ++i) {
14            k = -1; maxc = -1;
15            for (j = 1; j <= n; ++j)
16                if (!bin[j] && !vis[j] && dist[j] > maxc) {
17                    k = j; maxc = dist[j]; }
18            for (j = 1; j <= n; ++j) if (!bin[j] && !vis[j])
19                dist[j] += maxc;
20            bin[k] = true;
21            if (k != t) {
22                for (j = 1; j <= n; ++j) if (!bin[j] && !vis[j])
23                    edge[k][j] += edge[j][k];
24                edge[k][t] += edge[t][k];
25                n--;
26                if (k != s) s = k;
27            }
28        }
29        return mincut;
30    }
```

```
18     if (k == -1) return mincut;
19     s = t; t = k; mincut = maxc; vis[k] = true;
20     for (j = 1; j <= n; ++j) if (!bin[j] && !vis[j])
21         dist[j] += edge[k][j]; }
22     return mincut; }
23 int solve () {
24     int mincut, i, j, s, t, ans;
25     for (mincut = INF, i = 1; i < n; ++i) {
26         ans = contract (s, t); bin[t] = true;
27         if (mincut > ans) mincut = ans;
28         if (mincut == 0) return 0;
29         for (j = 1; j <= n; ++j) if (!bin[j])
30             edge[s][j] = (edge[j][s] += edge[j][t]); }
31     return mincut; }
```

### 4.3.4 Tarjan

Find strongly-connected components on directed graphs, or edge/vertex-biconnected components on undirected graphs.

```
1 template <int MAXN = 1000000, int MAXM = 1000000>
2 struct tarjan {
3     int comp[MAXN], size;
4     int dfn[MAXN], ind, low[MAXN], ins[MAXN], stk[MAXN],
5         stks;
6     void dfs (const edge_list <MAXN, MAXM> &e, int i) {
7         dfn[i] = low[i] = ind++;
8         ins[i] = 1; stk[stks++] = i;
9         for (int x = e.begin[i]; ~x; x = e.next[x]) {
10            int j = e.dest[x]; if (!dfn[j]) {
11                dfs (e, j);
12                if (low[i] > low[j]) low[i] = low[j];
13                if (low[j] > dfn[i]); //edge-biconnected
14                if (low[j] >= dfn[i]); //vertex-biconnected
15            } else if (ins[j] && low[i] > dfn[j])
16                low[i] = dfn[j];
17            if (dfn[i] == low[i]) { //strongly-connected
18                for (int j = -1; j != i;
19                    j = stk[--stks], ins[j] = 0, comp[j] = size);
20                ++size; } }
21     void solve (const edge_list <MAXN, MAXM> &e, int n) {
22         size = ind = stks = 0;
23         std::fill (dfn, dfn + n, -1);
24         for (int i = 0; i < n; ++i) if (!dfn[i])
25             dfs (e, i); }
```

## 4.4 Flow

### 4.4.1 Maximum flow

ISAP is better for sparse graphs, while Dinic is better for dense graphs.

```
1 template <int MAXN = 1000, int MAXM = 100000>
2 struct isap {
3     struct flow_edge_list {
4         int size, begin[MAXN], dest[MAXN], next[MAXN], flow[
5             MAXM];
6         void clear (int n) { size = 0; std::fill (begin,
7             begin + n, -1); }
8         flow_edge_list (int n = MAXN) { clear (n); }
9         void add_edge (int u, int v, int f) {
10            dest[size] = v; next[size] = begin[u]; flow[size] =
11                f; begin[u] = size++;
12            dest[size] = u; next[size] = begin[v]; flow[size] =
13                0; begin[v] = size++; } }
14     int pre[MAXN], d[MAXN], gap[MAXN], cur[MAXN], que[
15         MAXN], vis[MAXN];
16     int solve (flow_edge_list &e, int s, int t) {
17         for (int i = 0; i < n; ++i) { pre[i] = d[i] = gap[i]
18             = vis[i] = 0; cur[i] = e.begin[i]; }
19         int l = 0, r = 0; que[0] = t; gap[0] = 1; vis[t] =
20             true;
21         while (l <= r) { int u = que[l++];
22             for (int i = e.begin[u]; ~i; i = e.next[i])
23                 if (e.flow[i] == 0 && !vis[e.dest[i]]) {
24                     que[++r] = e.dest[i];
25                     vis[e.dest[i]] = true;
26                     d[e.dest[i]] = d[u] + 1;
27                     ++gap[d[e.dest[i]]]; } }
28         for (int i = 0; i < n; ++i) if (!vis[i]) d[i] = n,
29             ++gap[n];
30         int u = pre[s] = s, v, maxflow = 0;
31         while (d[s] < n) {
32             v = n; for (int i = cur[u]; ~i; i = e.next[i])
33                 if (e.flow[i] && d[u] == d[e.dest[i]] + 1) {
34                     v = e.dest[i]; cur[u] = i; break; }
35             if (v < n) {
36                 pre[v] = u; u = v;
37                 if (v == t) {
38                     int dflow, p = t; u = s;
39                     while (p != s) { p = pre[p]; dflow = std::min (
40                         dflow, e.flow[cur[p]]); }
41                     maxflow += dflow; p = t;
42                     while (p != s) { p = pre[p]; e.flow[cur[p]] -=
43                         dflow; e.flow[cur[p] ^ 1] += dflow; } }
44             } else {
45                 int mindist = n + 1;
46                 for (int i = e.begin[u]; ~i; i = e.next[i])
47                     if (e.flow[i] && mindist > d[e.dest[i]]) {
48                         mindist = d[e.dest[i]]; cur[u] = i; }
49                 if (!--gap[d[u]]) return maxflow;
50                 gap[d[u] = mindist + 1]++; u = pre[u]; } }
51         return maxflow; } }
52 template <int MAXN = 1000, int MAXM = 100000>
```

```

43 struct dinic {
44     struct flow_edge_list {
45         int size, begin[MAXN], dest[MAXM], next[MAXM], flow[
46             MAXM];
47         void clear (int n) { size = 0; std::fill (begin,
48             begin + n, -1); }
49         flow_edge_list (int n = MAXN) { clear (n); }
50         void add_edge (int u, int v, int f) {
51             dest[size] = v; next[size] = begin[u]; flow[size] =
52                 f; begin[u] = size++;
53             dest[size] = u; next[size] = begin[v]; flow[size] =
54                 0; begin[v] = size++; } };
55     int n, s, t, d[MAXN], w[MAXN], q[MAXN];
56     int bfs (flow_edge_list &e) {
57         std::fill (d, d + n, -1);
58         int l, r; q[l = r = 0] = s, d[s] = 0;
59         for (; l <= r; ++l)
60             for (int k = e.begin[q[l]]; ~k; k = e.next[k])
61                 if (!d[e.dest[k]] && e.flow[k] > 0) d[e.dest[k]]
62                     = d[q[l]] + 1, q[++r] = e.dest[k];
63         return ~d[t] ? 1 : 0; }
64     int dfs (flow_edge_list &e, int u, int ext) {
65         if (u == t) return ext; int k = w[u], ret = 0;
66         for (; ~k; k = e.next[k], w[u] = k) {
67             if (ext == 0) break;
68             if (d[e.dest[k]] == d[u] + 1 && e.flow[k] > 0) {
69                 int flow = dfs (e, e.dest[k], std::min (e.flow[k],
70                     ext));
71                 if (flow > 0) {
72                     e.flow[k] -= flow, e.flow[k ^ 1] += flow;
73                     ret += flow, ext -= flow; } } }
74         if (!k) d[u] = -1; return ret; }
75     int solve (flow_edge_list &e, int n_, int s_, int t_)
76     {
77         int ans = 0; n = n_; s = s_; t = t_;
78         while (bfs (e)) {
79             for (int i = 0; i < n; ++i) w[i] = e.begin[i];
80             ans += dfs (e, s, INF); }
81         return ans; } };

```

#### 4.4.2 Minimum cost flow

EK is better for sparse graphs, while ZKW is better for dense graphs.

```

1 template <int MAXN = 1000, int MAXM = 100000>
2 struct minimum_cost_flow {
3     struct cost_flow_edge_list {
4         int size, begin[MAXN], dest[MAXM], next[MAXM], cost[
5             MAXM], flow[MAXM];
6         void clear (int n) { size = 0; std::fill (begin,
7             begin + n, -1); }
8         cost_flow_edge_list (int n = MAXN) { clear (n); }
9         void add_edge (int u, int v, int c, int f) {
10             dest[size] = v; next[size] = begin[u]; cost[size] =
11                 c; flow[size] = f; begin[u] = size++;
12             dest[size] = u; next[size] = begin[v]; cost[size] =
13                 -c; flow[size] = 0; begin[v] = size++; } };
14     int n, s, t, prev[MAXN], dist[MAXN], occur[MAXN];
15     bool augment (cost_flow_edge_list &e) {
16         std::vector <int> queue;
17         std::fill (dist, dist + n, INF); std::fill (occur,
18             occur + n, 0);
19         dist[s] = 0; occur[s] = true; queue.push_back (s);
20         for (int head = 0; head < (int)queue.size(); ++head)
21             {
22                 int x = queue[head];
23                 for (int i = e.begin[x]; ~i; i = e.next[i]) {
24                     int y = e.dest[i];
25                     if (e.flow[i] && dist[y] > dist[x] + e.cost[i]) {
26                         dist[y] = dist[x] + e.cost[i]; prev[y] = i;
27                         if (!occur[y]) {
28                             occur[y] = true; queue.push_back (y); } } }
29                 occur[x] = false; }
30         return dist[t] < INF; }
31     std::pair <int, int> solve (cost_flow_edge_list &e,
32         int n_, int s_, int t_) {
33         n = n_; s = s_; t = t_; std::pair <int, int> ans =
34             std::make_pair (0, 0);
35         while (augment (e)) {
36             int num = INF;
37             for (int i = t; i != s; i = e.dest[prev[i] ^ 1])
38                 num = std::min (num, e.flow[prev[i]]);
39             ans.first += num;
40             for (int i = t; i != s; i = e.dest[prev[i] ^ 1]) {
41                 e.flow[prev[i]] -= num; e.flow[prev[i] ^ 1] += num;
42                 ans.second += num * e.cost[prev[i]]; } }
43         return ans; } };
44 template <int MAXN = 1000, int MAXM = 100000>
45 struct zkw_flow {
46     struct cost_flow_edge_list {
47         int size, begin[MAXN], dest[MAXM], next[MAXM], cost[
48             MAXM], flow[MAXM];
49         void clear (int n) { size = 0; std::fill (begin,
50             begin + n, -1); }
51         cost_flow_edge_list (int n = MAXN) { clear (n); }
52         void add_edge (int u, int v, int c, int f) {
53             dest[size] = v; next[size] = begin[u]; cost[size] =
54                 c; flow[size] = f; begin[u] = size++;
55             dest[size] = u; next[size] = begin[v]; cost[size] =
56                 -c; flow[size] = 0; begin[v] = size++; } };
57     int n, s, t, tf, tc, dis[MAXN], slack[MAXN], visit[
58         MAXN];
59     int modlable() {

```

```

47     int delta = INF;
48     for (int i = 0; i < n; i++) {
49         if (!visit[i] && slack[i] < delta) delta = slack[i];
50         slack[i] = INF; }
51     if (delta == INF) return 1;
52     for (int i = 0; i < n; i++) if (visit[i]) dis[i] +=
53         delta;
54     return 0; }
55     int dfs (cost_flow_edge_list &e, int x, int flow) {
56         if (x == t) { tf += flow; tc += flow * (dis[s] - dis
57             [t]); return flow; }
58         visit[x] = 1; int left = flow;
59         for (int i = e.begin[x]; ~i; i = e.next[i])
60             if (e.flow[i] > 0 && !visit[e.dest[i]]) {
61                 int y = e.dest[i];
62                 if (dis[y] + e.cost[i] == dis[x]) {
63                     int delta = dfs (e, y, std::min (left, e.flow[i])
64                         );
65                     e.flow[i] -= delta; e.flow[i ^ 1] += delta; left
66                         -= delta;
67                     if (!left) { visit[x] = false; return flow; } }
68             } else
69                 slack[y] = std::min (slack[y], dis[x] + e.cost[i]
70                     - dis[y]);
71         return flow - left; }
72     std::pair <int, int> solve (cost_flow_edge_list &e,
73         int n_, int s_, int t_) {
74         n = n_; s = s_; t = t_; tf = tc = 0;
75         std::fill (dis + 1, dis + n + 1, 0);
76         do { do {
77             std::fill (visit + 1, visit + n + 1, 0);
78             while (dfs (e, s, INF)); } while (!modlable ());
79         } while (true);
80         return std::make_pair (tf, tc);
81     } };

```

## 4.5 Matching

**Tutte-Berge formula** The theorem states that the size of a maximum matching of a graph  $G = (V, E)$  equals

$$\frac{1}{2} \min_{U \subseteq V} (|U| - \text{odd}(G - U) + |V|),$$

where  $\text{odd}(H)$  counts how many of the connected components of the graph  $H$  have an odd number of vertices.

**Tutte theorem** A graph,  $G = (V, E)$ , has a perfect matching if and only if for every subset  $U$  of  $V$ , the subgraph induced by  $V - U$  has at most  $|U|$  connected components with an odd number of vertices.

**Hall's marriage theorem** A family  $S$  of finite sets has a transversal if and only if  $S$  satisfies the marriage condition.

### 4.5.1 Blossom algorithm

Maximum matching for general graphs.

```

1 template <int MAXN = 500, int MAXM = 250000>
2 struct blossom {
3     int match[MAXN], d[MAXN], fa[MAXN], c1[MAXN], c2[MAXN]
4         , v[MAXN], q[MAXN];
5     int *qhead, *qtail;
6     struct {
7         int fa[MAXN];
8         void init (int n) { for (int i = 1; i <= n; i++) fa[i]
9             = i; }
10        int find (int x) { if (fa[x] != x) fa[x] = find (fa[x]);
11            return fa[x]; }
12        void merge (int x, int y) { x = find (x); y = find (y);
13            fa[x] = y; } } ufs;
14        void solve (int x, int y) {
15            if (x == y) return;
16            if (d[y] == 0) {
17                solve (x, fa[fa[y]]); match[fa[y]] = fa[fa[y]];
18                match[fa[fa[y]]] = fa[y];
19            } else if (d[y] == 1) {
20                solve (match[y], c1[y]); solve (x, c2[y]);
21                match[c1[y]] = c2[y]; match[c2[y]] = c1[y]; } }
22        int lca (int x, int y, int root) {
23            x = ufs.find (x); y = ufs.find (y);
24            while (x != y && v[x] != 1 && v[y] != 0) {
25                v[x] = 0; v[y] = 1;
26                if (x != root) x = ufs.find (fa[x]);
27                if (y != root) y = ufs.find (fa[y]); }
28            if (v[y] == 0) std::swap (x, y);
29            for (int i = x; i != y; i = ufs.find (fa[i])) v[i] =
30                -1;
31            v[y] = -1; return x; }
32        void contract (int x, int y, int b) {
33            for (int i = ufs.find (x); i != b; i = ufs.find (fa[i])) {
34                ufs.merge (i, b);
35                if (d[i] == 1) { c1[i] = x; c2[i] = y; *qtail++ = i; } } }
36        bool bfs (int root, int n, const edge_list <MAXN,
37            MAXM> &e) {
38            ufs.init (n); std::fill (d, d + MAXN, -1); std::fill
39                (v, v + MAXN, -1);
40            qhead = qtail = q; d[root] = 0; *qtail++ = root;
41            while (qhead < qtail) {
42                for (int loc = *qhead++; i = e.begin[loc]; ~i; i =
43                    e.next[i]) {
44                    int dest = e.dest[i];
45                    if (match[dest] == -2 || ufs.find (loc) == ufs.
46                        find (dest)) continue;
47                    if (d[dest] == -1)

```

```

39 if (match[dest] == -1) {
40     solve (root, loc); match[loc] = dest;
41     match[dest] = loc; return 1;
42 } else {
43     fa[dest] = loc; fa[match[dest]] = dest;
44     d[dest] = 1; d[match[dest]] = 0;
45     *qtail++ = match[dest];
46 } else if (d[ufs.find (dest)] == 0) {
47     int b = lca (loc, dest, root);
48     contract (loc, dest, b); contract (dest, loc, b);
49     return 0; }
50 int solve (int n, const edge_list <MAXN, MAXM> &e) {
51     std::fill (fa, fa + n, 0); std::fill (c1, c1 + n, 0);
52     std::fill (c2, c2 + n, 0); std::fill (match, match +
53         n, -1);
54     int re = 0; for (int i = 0; i < n; i++)
55         if (match[i] == -1) if (bfs (i, n, e)) ++re; else
56             match[i] = -2;
57     return re; }

```

#### 4.5.2 Blossom algorithm (weighted)

Maximum matching for general weighted graphs in  $O(n^3)$  (1-based).

Usage:

1. Set  $n$  to the size of the vertices.
2. Execute init.
3. Set  $g[] [] .w$  to the weight of the edges.
4. Execute solve.
5. The first result is the answer, the second one is the number of matching pairs. Obtain the exact matching with `match[]`.

```

1 struct weighted_blossom {
2     static const int INF = INT_MAX, MAXN = 400;
3     struct edge { int u, v, w; edge (int u = 0, int v = 0,
4         int w = 0): u(u), v(v), w(w) {} };
5     int n, n_x;
6     edge g[MAXN * 2 + 1][MAXN * 2 + 1];
7     int lab[MAXN * 2 + 1], match[MAXN * 2 + 1], slack[
8         MAXN * 2 + 1], st[MAXN * 2 + 1], pa[MAXN * 2 +
9         1];
10    int flower_from[MAXN * 2 + 1][MAXN + 1], S[MAXN * 2 +
11        1], vis[MAXN * 2 + 1];
12    std::vector <int> flower[MAXN * 2 + 1]; std::queue <
13        int> q;
14    int e_delta (const edge &e) { return lab[e.u] + lab[e
15        .v] - g[e.u][e.v].w * 2; }
16    void update_slack (int u, int x) { if (!slack[x] ||
17        e_delta (g[u][x]) < e_delta (g[slack[x]][x]))
18        slack[x] = u; }
19    void set_slack (int x) { slack[x] = 0; for (int u =
20        1; u <= n; ++u) if (g[u][x].w > 0 && st[u] != x &&
21        S[st[u]] == 0)
22        update_slack(u, x); }
23    void q_push (int x) {
24        if (x <= n) q.push (x);
25        else for (size_t i = 0; i < flower[x].size (); i++)
26            q.push (flower[x][i]); }
27    void set_st (int x, int b) {
28        st[x] = b; if (x > n) for (size_t i = 0; i < flower[
29            x].size (); ++i) set_st (flower[x][i], b); }
30    int get_pr (int b, int xr) {
31        int pr = std::find (flower[b].begin (), flower[b].
32            end (), xr) - flower[b].begin ();
33        if (pr % 2 == 1) { std::reverse (flower[b].begin ()
34            + 1, flower[b].end ()); return (int) flower[b].
35            size () - pr; }
36        else return pr; }
37    void set_match (int u, int v) {
38        match[u] = g[u][v].v; if (u > n) {
39            edge e = g[u][v]; int xr = flower_from[u][e.u], pr
40            = get_pr (u, xr);
41            for (int i = 0; i < n; ++i) set_match (flower[u][i]
42                , flower[u][i] ^ 1);
43            set_match (xr, v); std::rotate (flower[u].begin (),
44                flower[u].begin () + pr, flower[u].end ()); }
45        }
46    void augment (int u, int v) {
47        for (; ) {
48            int xnv = st[match[u]]; set_match (u, v);
49            if (!xnv) return; set_match (xnv, st[pa[xnv]]);
50            u = st[pa[xnv]], v = xnv; } }
51    int get_lca (int u, int v) {
52        static int t = 0;
53        for (++t; u || v; std::swap (u, v)) {
54            if (u == 0) continue; if (vis[u] == t) return u;
55            vis[u] = t; u = st[match[u]]; if (u) u = st[pa[u]];
56        }
57        return 0; }
58    void add_blossom (int u, int lca, int v) {
59        int b = n + 1; while (b <= n_x && st[b]) ++b;
60        if (b > n_x) ++n_x;
61        lab[b] = 0, S[b] = 0;
62        match[b] = match[lca]; flower[b].clear ();
63        flower[b].push_back (lca);
64        for (int x = u, y; x != lca; x = st[pa[y]]) {
65            flower[b].push_back (x), flower[b].push_back (y =
66                st[match[x]]), q.push (y); }
67        std::reverse (flower[b].begin () + 1, flower[b].end
68            ());
69        for (int x = v, y; x != lca; x = st[pa[y]]) {
70            flower[b].push_back (x), flower[b].push_back (y =
71                st[match[x]]), q.push (y); }

```

```

72    set_st (b, b);
73    for (int x = 1; x <= n_x; ++x) g[b][x].w = g[x][b].w
74        = 0;
75    for (int x = 1; x <= n; ++x) flower_from[b][x] = 0;
76    for (size_t i = 0; i < flower[b].size (); ++i) {
77        int xs = flower[b][i];
78        for (int x = 1; x <= n_x; ++x) if (g[b][x].w == 0
79            || e_delta (g[xs][x]) < e_delta (g[b][x]))
80            g[b][x] = g[xs][x], g[x][b] = g[x][xs];
81        for (int x = 1; x <= n; ++x) if (flower_from[xs][x])
82            flower_from[b][x] = xs; }
83    set_slack (b); }
84    void expand_blossom (int b) {
85        for (size_t i = 0; i < flower[b].size (); ++i)
86            set_st (flower[b][i], flower[b][i]);
87        int xr = flower_from[b][g[b][pa[b]].u], pr = get_pr (
88            b, xr);
89        for (int i = 0; i < pr; i += 2) {
90            int xs = flower[b][i], xns = flower[b][i + 1];
91            pa[xs] = g[xns][xs].u; S[xs] = 1, S[xns] = 0;
92            slack[xs] = 0, set_slack (xns); q_push (xns); }
93        S[xr] = 1, pa[xr] = pa[b];
94        for (size_t i = pr + 1; i < flower[b].size (); ++i)
95            {
96                int xs = flower[b][i]; S[xs] = -1, set_slack (xs); }
97        st[b] = 0; }
98    bool on_found_edge (const edge &e) {
99        int u = st[e.u], v = st[e.v];
100        if (S[v] == -1) {
101            pa[v] = e.u, S[v] = 1; int nu = st[match[v]];
102            slack[v] = slack[nu] = 0; S[nu] = 0, q_push (nu);
103        } else if (S[v] == 0) {
104            int lca = get_lca (u, v);
105            if (!lca) return augment (u, v), augment (v, u), true
106                ;
107            else add_blossom (u, lca, v); }
108    return false; }
109    bool matching () {
110        memset (S + 1, -1, sizeof (int) * n_x);
111        memset (slack + 1, 0, sizeof (int) * n_x);
112        q = std::queue <int> ();
113        for (int x = 1; x <= n_x; ++x) if (st[x] == x && !
114            match[x]) pa[x] = 0, S[x] = 0, q_push (x);
115        if (q.empty ()) return false;
116        for (; ) {
117            while (q.size ()) {
118                int u = q.front (); q.pop ();
119                if (S[st[u]] == 1) continue;
120                for (int v = 1; v <= n; ++v) if (g[u][v].w > 0 &&
121                    st[u] != st[v]) {
122                    if (e_delta (g[u][v]) == 0) {
123                        if (on_found_edge (g[u][v])) return true;
124                        else update_slack (u, st[v]); } }
125                int d = INF;
126                for (int b = n + 1; b <= n_x; ++b) if (st[b] == b &&
127                    S[b] == 1) d = std::min (d, lab[b] / 2);
128                for (int x = 1; x <= n_x; ++x) if (st[x] == x &&
129                    slack[x]) {
130                    if (S[x] == -1) d = std::min (d, e_delta (g[slack[
131                        x]][x]));
132                    else if (S[x] == 0) d = std::min (d, e_delta (g[
133                        slack[x]][x]) / 2); }
134                for (int u = 1; u <= n; ++u) {
135                    if (S[st[u]] == 0) {
136                        if (lab[u] <= d) return 0;
137                        lab[u] -= d;
138                    } else if (S[st[u]] == 1) lab[u] += d; }
139                for (int b = n + 1; b <= n_x; ++b)
140                    if (st[b] == b) {
141                        if (S[st[b]] == 0) lab[b] += d * 2;
142                        else if (S[st[b]] == 1) lab[b] -= d * 2; }
143                q = std::queue <int> ();
144                for (int x = 1; x <= n_x; ++x)
145                    if (st[x] == x && slack[x] && st[slack[x]] != x &&
146                        e_delta (g[slack[x]][x]) == 0)
147                        if (on_found_edge (g[slack[x]][x])) return true;
148                for (int b = n + 1; b <= n_x; ++b) if (st[b] == b
149                    && S[b] == 1 && lab[b] == 0) expand_blossom (b);
150            }
151            return false; }
152    std::pair <long long, int> solve () {
153        memset (match + 1, 0, sizeof (int) * n); n_x = n;
154        int n_matches = 0; long long tot_weight = 0;
155        for (int u = 0; u <= n; ++u) st[u] = u, flower[u].
156            clear ();
157        int w_max = 0;
158        for (int u = 1; u <= n; ++u) for (int v = 1; v <= n;
159            ++v) {
160            flower_from[u][v] = (u == v ? u : 0); w_max = std::
161                max (w_max, g[u][v].w); }
162        for (int u = 1; u <= n; ++u) lab[u] = w_max;
163        while (matching ()) ++n_matches;
164        for (int u = 1; u <= n; ++u) if (match[u] && match[u]
165            < u) tot_weight += g[u][match[u]].w;
166        return std::make_pair (tot_weight, n_matches); }
167    void init () { for (int u = 1; u <= n; ++u) for (int
168        v = 1; v <= n; ++v) g[u][v] = edge (u, v, 0); }

```

#### 4.5.3 Hopcroft-Karp algorithm

Unweighted maximum matching for bipartite graphs in  $O(m\sqrt{n})$ .

```

1 template <int MAXN = 100000, int MAXM = 100000>

```



```

2 struct hopcroft_karp {
3   int mx[MAXN], my[MAXN], lv[MAXN];
4   bool dfs (edge_list <MAXN, MAXN> &e, int x) {
5     for (int i = e.begin[x]; ~i; i = e.next[i]) {
6       int y = e.dest[i], w = my[y];
7       if (!w || (lv[x] + 1 == lv[w] && dfs (e, w))) {
8         mx[x] = y; my[y] = x; return true; } }
9     lv[x] = -1; return false; }
10  int solve (edge_list <MAXN, MAXN> &e, int n, int m) {
11    std::fill (mx, mx + n, -1); std::fill (my, my + m,
12      -1);
13    for (int ans = 0; ; ) {
14      std::vector <int> q;
15      for (int i = 0; i < n; ++i)
16        if (mx[i] == -1) {
17          lv[i] = 0; q.push_back (i);
18        } else lv[i] = -1;
19      for (int head = 0; head < (int) q.size(); ++head) {
20        int x = q[head];
21        for (int i = e.begin[x]; ~i; i = e.next[i]) {
22          int y = e.dest[i], w = my[y];
23          if (~w && lv[w] < 0) { lv[w] = lv[x] + 1; q.
24            push_back (w); } } }
25      int d = 0; for (int i = 0; i < n; ++i) if (!~mx[i]
26        && dfs (e, i)) ++d;
27      if (d == 0) return ans; else ans += d; } } };

```

#### 4.5.4 Kuhn-Munkres algorithm

Weighted maximum matching on bipartition graphs. Input  $n$  and  $w$ . Collect the matching in  $m[]$ . The graph is 1-based.

```

1 template <int MAXN = 500>
2 struct kuhn_munkres {
3   int n, w[MAXN][MAXN], lx[MAXN], ly[MAXN], m[MAXN],
4     way[MAXN], sl[MAXN];
5   bool u[MAXN];
6   void hungary(int x) {
7     m[0] = x; int j0 = 0;
8     std::fill (sl, sl + n + 1, INF); std::fill (u, u + n
9       + 1, false);
10    do {
11      u[j0] = true; int i0 = m[j0], d = INF, j1 = 0;
12      for (int j = 1; j <= n; ++j)
13        if (u[j] == false) {
14          int cur = -w[i0][j] - lx[i0] - ly[j];
15          if (cur < sl[j]) { sl[j] = cur; way[j] = j0; }
16          if (sl[j] < d) { d = sl[j]; j1 = j; } }
17      for (int j = 0; j <= n; ++j) {
18        if (u[j]) { lx[m[j]] += d; ly[j] -= d; }
19        else sl[j] -= d; }
20      j0 = j1; } while (m[j0] != 0);
21    do {
22      int j1 = way[j0]; m[j0] = m[j1]; j0 = j1;
23    } while (j0); }
24  int solve() {
25    for (int i = 1; i <= n; ++i) m[i] = lx[i] = ly[i] =
26      way[i] = 0;
27    for (int i = 1; i <= n; ++i) hungary (i);
28    int sum = 0; for (int i = 1; i <= n; ++i) sum += w[m
29      [i]][i];
30    return sum; } };

```

## 4.6 Path

### 4.6.1 K-shortest path

```

1 const int maxn = 1005, maxe = 10005, maxm = maxe * 30;
2 struct A {
3   int x, d;
4   A (int x, int d) : x (x), d (d) {}
5   bool operator < (const A &a) const { return d > a.d; }
6 };
7 struct node {
8   int w, i, d;
9   node *lc, *rc;
10  node () {}
11  node (int w, int i) : w (w), i (i), d (0) {}
12  void refresh () { d = rc -> d + 1; }
13 } null[maxn], *ptr = null, *root[maxn];
14 struct B {
15   int x, w;
16   node *rt;
17   B (int x, node *rt, int w) : x (x), w (w), rt (rt) {}
18   bool operator < (const B &a) const { return w + rt ->
19     w > a.w + a.rt -> w; } };
20 std::vector <int> G[maxn], W[maxn], id[maxn]; // Store
21   reversed graph & clear G at the beginning.
22 bool vis[maxn], used[maxn];
23 int u[maxe], v[maxe], w[maxe]; // Store every edge (
24   uni-directional).
25 int d[maxn], p[maxn];
26 int n, m, k, s, t; // s, t - beginning and end.
27 // main
28 for (int i = 0; i <= n; ++i) root[i] = null;
29 //Read & build the reversed graph.
30 Dijkstra ();
31 // Clear G, W, id.
32 for (int i = 1; i <= n; ++i)
33   if (p[i]) { used[p[i]] = true; G[v[p[i]]].push_back (
34     i); }
35 for (int i = 1; i <= m; ++i) {
36   w[i] -= d[u[i]] - d[v[i]];
37   if (!used[i]) root[u[i]] = merge (root[u[i]], newnode
38     (w[i], i)); }

```

```

33 dfs (t);
34 std::priority_queue <B> heap;
35 heap.push (B (s, root[s], 0));
36 printf ("%d\n", d[s]); // The shortest path.
37 while (--k) {
38   if (heap.empty ()) printf ("-1\n");
39   else {
40     int x = heap.top ().x, w = heap.top ().w;
41     node *rt = heap.top ().rt; heap.pop ();
42     printf ("%d\n", d[s] + w + rt -> w);
43     if (rt -> lc != null || rt -> rc != null)
44       heap.push (B (x, merge (rt -> lc, rt -> rc), w));
45     if (root[v[rt -> i]] != null)
46       heap.push (B (v[rt -> i], root[v[rt -> i]], w + rt
47         -> w)); } }
48 void Dijkstra () {
49   memset (d, 63, sizeof (d)); d[t] = 0;
50   std::priority_queue <A> heap;
51   heap.push (A (t, 0));
52   while (!heap.empty ()) {
53     int x = heap.top ().x; heap.pop ();
54     if (vis[x]) continue; vis[x] = true;
55     for (int i = 0; i < (int) G[x].size (); ++i)
56       if (!vis[G[x][i]] && d[G[x][i]] > d[x] + W[x][i]) {
57         d[G[x][i]] = d[x] + W[x][i];
58         p[G[x][i]] = id[x][i];
59         heap.push (A (G[x][i], d[G[x][i]])); } } }
60 void dfs (int x) {
61   root[x] = merge (root[x], root[v[p[x]]]);
62   for (int i = 0; i < (int) G[x].size (); ++i) dfs (G[x
63     ][i]); }
64 node *newnode (int w, int i) {
65   **ptr = node (w, i);
66   ptr -> lc = ptr -> rc = null;
67   return ptr; }
68 node *merge (node *x, node *y) {
69   if (x == null) return y;
70   if (y == null) return x;
71   if (x -> w > y -> w) swap (x, y);
72   node *z = newnode (x -> w, x -> i);
73   z -> lc = x -> lc; z -> rc = merge (x -> rc, y);
74   if (z -> lc -> d > z -> rc -> d) swap (z -> lc, z ->
75     rc);
76   z -> refresh (); return z; }

```

### 4.6.2 Lindström-Gessel-Viennot lemma

Let  $G$  be a locally finite directed acyclic graph. This means that each vertex has finite degree, and that  $G$  contains no directed cycles. Consider base vertices  $A = \{a_1, \dots, a_n\}$  and destination vertices  $B = \{b_1, \dots, b_n\}$ , and also assign a weight  $w_e$  to each directed edge  $e$ . These edge weights are assumed to belong to some commutative ring. For each directed path  $P$  between two vertices, let  $\omega(P)$  be the product of the weights of the edges of the path. For any two vertices  $a$  and  $b$ , write  $e(a, b)$  for the sum  $e(a, b) = \sum_{P: a \rightarrow b} \omega(P)$  over all paths from  $a$  to  $b$ .

With this setup, write:

$$M = \begin{pmatrix} e(a_1, b_1) & e(a_1, b_2) & \cdots & e(a_1, b_n) \\ e(a_2, b_1) & e(a_2, b_2) & \cdots & e(a_2, b_n) \\ \vdots & \vdots & \ddots & \vdots \\ e(a_n, b_1) & e(a_n, b_2) & \cdots & e(a_n, b_n) \end{pmatrix}.$$

An  $n$ -tuple of non-intersecting paths from  $A$  to  $B$  means an  $n$ -tuple  $(P_1, \dots, P_n)$  of paths in  $G$  with the following properties:

1. There exists a permutation  $\sigma$  of  $\{1, 2, \dots, n\}$  such that, for every  $i$ , the path  $P_i$  is a path from  $a_i$  to  $b_{\sigma(i)}$ .
2. Whenever  $i \neq j$ , the paths  $P_i$  and  $P_j$  have no two vertices in common (not even endpoints).

Given such an  $n$ -tuple  $(P_1, \dots, P_n)$ , we denote by  $\sigma(P)$  the permutation of  $\sigma$  from the first condition.

The Lindström-Gessel-Viennot lemma then states that the determinant of  $M$  is the signed sum over all  $n$ -tuples  $P = (P_1, \dots, P_n)$  of non-intersecting paths from  $A$  to  $B$ :

$$\det(M) = \sum_{(P_1, \dots, P_n): A \rightarrow B} \text{sign}(\sigma(P)) \prod_{i=1}^n \omega(P_i).$$

That is, the determinant of  $M$  counts the weights of all  $n$ -tuples of non-intersecting paths starting at  $A$  and ending at  $B$ , each affected with the sign of the corresponding permutation of  $\{1, 2, \dots, n\}$ , given by  $P_i$  taking  $a_i$  to  $b_{\sigma(i)}$ .

In particular, if the only permutation possible is the identity (i.e., every  $n$ -tuple of non-intersecting paths from  $A$  to  $B$  takes  $a_i$  to  $b_i$  for each  $i$ ) and we take the weights to be 1, then  $\det(M)$  is exactly the number of non-intersecting  $n$ -tuples of paths starting at  $A$  and ending at  $B$ .

## 4.7 Tree

### 4.7.1 Optimum branching

The index of the root is 1. Check ( $\text{sel}[i]$ ,  $i$ ) for  $i$  in  $[2..n]$  for the result.

```

1 template <int MAXN = 1000>
2 struct optimum_branching {
3   int from[MAXN][MAXN * 2], n, m, edge[MAXN][MAXN * 2];
4   int sel[MAXN * 2], fa[MAXN * 2], vis[MAXN * 2];
5   int getfa (int x) { if (x == fa[x]) return x; return
6     fa[x] = getfa(fa[x]); }
7   void liuzhu () {
8     fa[1] = 1; for (int i = 2; i <= n; ++i) {

```



```

21 std::pair<int, long long> u[MAXN]; node tree[MAXN];
22 long long A[MAXN], B[MAXN];
23 void dfs1 (const edge_list <MAXN, MAXN> &e, int x,
24           int p = -1) {
25     tree[x] = node ();
26     for (int i = e.begin[x]; ~i; i = e.next[i]) {
27       int c = e.dest[i]; if (c != p) {
28         dfs1 (e, c, x); tree[x].add (tree[c].dl + 1, tree[
29           c].hl); } }
30     A[x] = tree[x].hash (); }
31 void dfs2 (const edge_list <MAXN, MAXN> &e, int x,
32           int p = -1) {
33     if (~p) tree[x].add (u[x].first, u[x].second);
34     B[x] = tree[x].hash ();
35     for (int i = e.begin[x]; ~i; i = e.next[i]) {
36       int c = e.dest[i]; if (c != p) {
37         u[c] = tree[x].del (tree[c].dl + 1, tree[c].hl);
38         dfs2 (e, c, x); } } }
39 void solve (const edge_list <MAXN, MAXN> &e, int root
40            ) {
41     dfs1 (e, root); dfs2 (e, root); } }
42 template<int MAXN, int MAXM, long long MOD>
43 long long tree_hash <MAXN, MAXM, MOD>::ra[MAXN];

```

```

1 template <int MAXN = 100000, int MAXM = 200000, long
   long MOD = 1000000000000000000000000>
2 struct tree_hash {
3     static long long ra[MAXN];
4     tree_hash () {
5         std::mt19937_64 mt (time (0));
6         std::uniform_int_distribution <long long> uid (0,
           MOD - 1);
7         for (int i = 0; i < MAXN; ++i) ra[i] = uid (mt); }
8     struct node {
9         std::vector <long long> s; int d1, d2; long long h1,
           h2;
10        node () { d1 = d2 = 0; }
11        void add (int d, long long v) {
12            s.push_back (v);
13            if (d > d1) d2 = d1, d1 = d; else if (d > d2) d2 =
              d; }
14        long long hash () {
15            h1 = h2 = 1; for (long long i : s) {
16                h1 = mul_mod (h1, ra[d1] + i, MOD);
17                h2 = mul_mod (h2, ra[d2] + i, MOD); } return h1; }
18        std::pair <int, long long> del (int d, long long v) {
19            if (d == d1) return { d2 + 1, mul_mod (h2, inverse (
              ra[d2] + v, MOD), MOD) };
20            return { d1 + 1, mul_mod (h1, inverse (ra[d1] + v,
              MOD), MOD) }; };

```

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

Usage:

1. CUBEN should be  $N^{\frac{1}{3}}$ .
2. Pass p\_f that returns the prefix sum of  $f(x)(1 \leq x < th)$ .
3. Pass p\_g that returns the prefix sum of  $g(x)(0 \leq x \leq N)$ .
4. Pass p\_c that returns the prefix sum of  $f * g(x)(0 \leq x \leq N)$ .
5. Pass th as the threshold, which generally should be  $N^{\frac{2}{3}}$ .
6. Pass mod as the module number, inv as the inverse of  $g(1)$  regarding mod.
7. Remember that  $x$  in p\_g(x) and p\_c(x) may be larger than mod!
8. Run init(n) first.
9. Use ans(x) to fetch answer for  $\frac{n}{x}$ .

```

1 template <int CUBEN = 3000>
2 struct prefix_mul {
3     typedef long long (*func) (long long);
4     func p_f, p_g, p_c;
5     long long mod, th, inv, n, mem[CUBEN];
6     prefix_mul (func p_f, func p_g, func p_c, long long
7         th, long long mod, long long inv) :
8         p_f (p_f), p_g (p_g), p_c (p_c), th (th), mod (mod),
9         inv (inv) {}
10    void init (long long n) {
11        prefix_mul::n = n;
12        for (long long i = 1, la; i <= n; i = la + 1) {
13            if ((la = n / (n / i)) < th) continue;
14            long long &ans = mem[n / la] = p_c (la);
15            for (long long j = 2, ne; j <= la; j = ne + 1) {
16                ne = la / (la / j);
17                ans = (ans + mod - (p_g (ne) - p_g (j - 1) + mod)
18                    *
19                    (la / j < th ? p_f (la / j) : mem[n / (la / j)])
20                    % mod);
21                if (ans >= mod) ans -= mod; }
22            if (inv != 1) ans = ans * inv % mod; } }
23    long long ans (long long x) {
24        if (n / x < th) return p_f (n / x);
25        return mem[n / (n / x)]; } };

```

#### 5.1.4 Euclidean-like algorithm

Compute  $\sum_{i=0}^{n-1} \lfloor \frac{a+bi}{m} \rfloor$ .

```

1 long long solve (long long n, long long a, long long b,
2     long long m) {
3     if (b == 0) return n * (a / m);
4     if (a >= m) return n * (a / m) + solve (n, a % m, b,
5         m);
6     if (b >= m) return (n - 1) * n / 2 * (b / m) + solve
7         (n, a, b % m, m);
8     return solve ((a + b * n) / m, (a + b * n) % m, m, b)
9         ; }

```

#### 5.1.5 Extended Eratosthenes sieve

Compute the prefix sum of multiplicative functions.

Usage:

1. Modify pre\_pow to compute the sum of powers.
2. Modify pfunc to compute  $f(p)$  with a prime  $p$ .
3. Modify cfunc to compute  $f(px)$  with  $f(x) = k$  and  $p|x$ .
4. Modify assemble to store  $f(x_i)$  in funca[i] with  $x_i^k$  equal to powa[k][i] and funcb[i] with  $x_i^k$  equal to powb[k][i].
5. Execute solve and profit.

```

1 template <int SN = 110000, int D = 2>
2 struct ees {
3     int co[SN], prime[SN], psize, sn;
4     long long powa[D + 1][SN], powb[D + 1][SN];
5     long long funca[SN], funcb[SN];
6     long long pow (long long x, int n) {
7         long long res = 1;
8         for (int i = 0; i < n; ++i) res *= x;
9         return res; }
10    long long pre_pow (long long x, int n) {
11        if (n == 0) return x;
12        if (n == 1) return (1 + x) * x / 2;
13        if (n == 2) return (1 + 2 * x) * (1 + x) * x / 6;
14        return 0; }
15    long long pfunc (long long p) { return -1; }
16    long long cfunc (long long k, long long p) { return
17        0; }
18    void assemble () {
19        for (int i = 1; i <= sn; ++i) {
20            funca[i] = -powa[0][i];
21            funcb[i] = -powb[0][i]; } }
22    void init (long long n) {
23        sn = std::max ((int) (ceil (sqrt (n)) + 1), 2);
24        psize = 0; for (int i = 2; i <= sn; ++i) {
25            if (!co[i]) prime[psize++] = i;
26            for (int j = 0; 1LL * i * prime[j] <= sn; ++j) {
27                co[i * prime[j]] = 1;
28                if (i % prime[j] == 0) break; } }
29        for (int d = 0; d <= D; ++d) {
30            long long *pa = powa[d], *pb = powb[d];
31            for (int i = 1; i <= sn; ++i) pa[i] = pre_pow (i, d)
32                - 1;
33            for (int i = 1; i <= sn; ++i) pb[i] = pre_pow (n /
34                i, d) - 1;
35            for (int i = 0; i < psize; ++i) { int &pi = prime[i]

```

```

36        for (int j = 1; j <= sn; ++j) if (n / j >= 1LL *
37            pi * pi) {
38                long long ch = n / j / pi;
39                pb[j] -= ((ch <= sn ? pa[ch] : pb[j * pi]) - pa[
40                    pi - 1]) * pow (pi, d);
41            } else break;
42        for (int j = sn; j >= 1; --j) if (j >= 1LL * pi *
43            pi)
44            pa[j] -= (pa[j / pi] - pa[pi - 1]) * pow (pi, d);
45        else break; } }
46    void dfs (int x, int f, long long mul, long long val,
47        long long n, long long &res) {
48        for (; x < psize && mul * prime[x] * prime[x] <= n;
49            ++x) {
50            long long nmul = mul * prime[x], nval = val * pfunc
51                (prime[x]);
52            for (; nmul <= n; nmul *= prime[x], nval = cfunc (
53                nval, prime[x]))
54                dfs (x + 1, prime[x], nmul, nval, n, res); }
55        if (n / mul > f) res += val * ((n / mul <= sn ?
56            funca[n / mul] : funcb[mul]) - funca[f]);
57        if (f > 1 && mul % (f * f) == 0) res += val; }
58    long long solve (long long n) {
59        if (n == 0) return 0;
60        long long res = 1;
61        init (n); dfs (0, 1, 1, 1, n, res);
62        return res; } };

```

#### 5.1.6 Fast power module

Compute  $x^n \bmod mod$ .

```

1 int fpm (int x, int n, int mod) {
2     int ans = 1, mul = x; while (n) {
3         if (n & 1) ans = int (1ll * ans * mul % mod);
4         mul = int (1ll * mul * mul % mod); n >>= 1; }
5     return ans; }
6 long long mul_mod (long long x, long long y, long long
7     mod) {
8     long long t = (x * y - (long long) ((long double) x /
9         mod * y + 1E-3) * mod) % mod;
10    return t < 0 ? t + mod : t; }
11 long long llfpm (long long x, long long n, long long
12     mod) {
13     long long ans = 1, mul = x; while (n) {
14         if (n & 1) ans = mul_mod (ans, mul, mod);
15         mul = mul_mod (mul, mul, mod); n >>= 1; }
16     return ans; }

```

#### 5.1.7 Lucas's theorem

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

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

where

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

and

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

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

#### 5.1.8 Mobius inversion

Mobius inversion formula

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

#### Gcd inversion

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

The formula can be computed in  $O(n \log n)$  complexity. Moreover, let  $l = dt$ , then

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

Let  $f(l) = \sum_{d|l} d^2 \mu\left(\frac{l}{d}\right)$ . 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 computed in  $O(n)$  complexity.

### 5.1.9 Zeller's congruence

Convert between a calendar date and its Gregorian calendar day ( $y \geq 1$ ) (0 = Monday, 1 = Tuesday, ..., 6 = Sunday).

```
1 int get_id (int y, int m, int d) {
2   if (m < 3) { --y; m += 12; }
3   return 365 * y + y / 4 - y / 100 + y / 400 + (153 * (
4     m - 3) + 2) / 5 + d - 307; }
5 std::tuple<int, int, int> date (int id) {
6   int x = id + 1789995, n, i, j, y, m, d;
7   n = 4 * x / 146097; x -= (146097 * n + 3) / 4;
8   i = (4000 * (x + 1)) / 1461001; x -= 1461 * i / 4 -
9     31;
10  j = 80 * x / 2447; d = x - 2447 * j / 80;
11  x = j / 11;
12  m = j + 2 - 12 * x; y = 100 * (n - 49) + i + x;
13  return std::make_tuple (y, m, d); }
```

## 5.2 Dynamic programming

Divide & conquer optimization For recurrence

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

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

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

## 5.3 Equality and inequality

### 5.3.1 Baby step giant step algorithm

Solve  $a^x = b \pmod c$  in  $O(\sqrt{c})$ .

```
1 struct bsqs {
2   int solve (int a, int b, int c) {
3     std::unordered_map<int, int> bs;
4     int m = (int) sqrt ((double) c) + 1, res = 1;
5     for (int i = 0; i < m; ++i) {
6       if (bs.find (res) == bs.end ()) bs[res] = i;
7       res = int (1LL * res * a % c); }
8     int mul = 1, inv = (int) inverse (a, c);
9     for (int i = 0; i < m; ++i) mul = int (1LL * mul *
10       inv % c);
11     res = b % c;
12     for (int i = 0; i < m; ++i) {
13       if (bs.find (res) != bs.end ()) return i * m + bs[
14         res];
15       res = int (1LL * res * mul % c); }
16     return -1; }
```

### 5.3.2 Chinese remainder theorem

Find positive integers  $x = out_{first} + k \cdot out_{second}$  that satisfies  $x \equiv in_{i,first} \pmod{in_{i,second}}$ .

```
1 struct crt {
2   long long fix (const long long &a, const long long &b)
3     { return (a % b + b) % b; }
4   bool solve (const std::vector<std::pair<long long,
5     long long>> &in, std::pair<long long, long long>
6     &out) {
7     out = std::make_pair (1LL, 1LL);
8     for (int i = 0; i < (int) in.size (); ++i) {
9       long long n, u;
10      euclid (out.second, in[i].second, n, u);
11      long long divisor = std::__gcd (out.second, in[i].
12        second);
13      if ((in[i].first - out.first) % divisor) return
14        false;
15      n *= (in[i].first - out.first) / divisor;
16      n = fix (n, in[i].second);
17      out.first += out.second * n;
18      out.second *= in[i].second / divisor;
19      out.first = fix (out.first, out.second); }
20   return true; }
```

### 5.3.3 Extended Euclidean algorithm

Solve  $ax + by = \gcd(a, b)$ .

```
1 void euclid (const long long &a, const long long &b,
2   long long &x, long long &y) {
3   if (b == 0) x = 1, y = 0;
4   else euclid (b, a % b, y, x), y -= a / b * x; }
5 long long inverse (long long x, long long m) {
6   long long a, b; euclid (x, m, a, b); return (a % m +
7   m) % m; }
```

### 5.3.4 Pell equation

Find the smallest integer root of  $x^2 - ny^2 = 1$  when  $n$  is not a square number, with the solution set  $x_{k+1} = x_0 x_k + n y_0 y_k, y_{k+1} = x_0 y_k + y_0 x_k$ .

```
1 template<int MAXN = 100000>
2 struct pell {
3   std::pair<long long, long long> solve (long long n)
4   { }
```

```
5   static long long p[MAXN], q[MAXN], g[MAXN], h[MAXN],
6     a[MAXN];
7   p[1] = q[0] = h[1] = 1; p[0] = q[1] = g[1] = 0;
8   a[2] = (long long) (floor (sqrtl (n) + 1e-7L));
9   for (int i = 2; ; ++i) {
10    g[i] = -g[i-1] + a[i] * h[i-1];
11    h[i] = (n - g[i] * g[i]) / h[i-1];
12    a[i+1] = (g[i] + a[2]) / h[i];
13    p[i] = a[i] * p[i-1] + p[i-2];
14    q[i] = a[i] * q[i-1] + q[i-2];
15    if (p[i] * p[i] - n * q[i] * q[i] == 1)
16      return { p[i], q[i] }; }
```

### 5.3.5 Quadric residue

Solve  $x^2 \equiv n \pmod p$  ( $0 \leq a < p$ ) where  $p$  is prime in  $O(\log p)$ .

```
1 struct quadric {
2   void multiply (long long &c, long long &d, long long a
3     , long long b, long long w, long long p) {
4     int cc = (a * c + b * d % p * w) % p;
5     int dd = (a * d + b * c) % p; c = cc, d = dd; }
6   bool solve (int n, int p, int &x) {
7     if (n == 0) return x = 0, true; if (p == 2) return x
8       = 1, true;
9     if (power (n, p / 2, p) == p - 1) return false;
10    long long c = 1, d = 0, b = 1, a, w;
11    do { a = rand () % p; w = (a * a - n + p) % p;
12      if (w == 0) return x = a, true; }
13    while (power (w, p / 2, p) != p - 1);
14    for (int times = (p + 1) / 2; times >= 1; {
15      if (times & 1) multiply (c, d, a, b, w, p);
16      multiply (a, b, a, b, w, p); }
17    return x = c, true; }
```

### 5.3.6 Simplex

Maximize  $\sum c_j x_j$  ( $0 \leq j < n$ ) with constraints  $\sum a_{ij} x_j \leq b_i$  ( $0 \leq i < m, 0 \leq j < n$ ). Collect the solution in `an[]`.

Note: maximizing  $\mathbf{c}^T \mathbf{x}$  subject to  $\mathbf{A} \mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq \mathbf{0}$  is equivalent to minimizing  $\mathbf{b}^T \mathbf{y}$  subject to  $\mathbf{A}^T \mathbf{x} \geq \mathbf{c}, \mathbf{y} \geq \mathbf{0}$ .

```
1 template<int MAXN = 100, int MAXM = 100>
2 struct simplex {
3   int n, m; double a[MAXN][MAXN], b[MAXN], c[MAXN];
4   bool infeasible, unbounded;
5   double v, an[MAXN + MAXM]; int q[MAXN + MAXM];
6   void pivot (int l, int e) {
7     std::swap (q[e], q[l + n]);
8     double t = a[l][e]; a[l][e] = 1; b[l] /= t;
9     for (int i = 0; i < n; ++i) a[l][i] /= t;
10    for (int i = 0; i < m; ++i) if (i != l && std::abs (
11      a[i][e]) > EPS) {
12      t = a[i][e]; a[i][e] = 0; b[i] -= t * b[l];
13      for (int j = 0; j < n; ++j) a[i][j] -= t * a[l][j]; }
14    if (std::abs (c[e]) > EPS) {
15      t = c[e]; c[e] = 0; v += t * b[l];
16      for (int j = 0; j < n; ++j) c[j] -= t * a[l][j]; }
17    bool pre () {
18      for (int l, e; ; ) {
19        l = e = -1;
20        for (int i = 0; i < m; ++i) if (b[i] < -EPS && (!~l
21          || rand () < 1)) l = i;
22        if (!~l) return false;
23        for (int i = 0; i < n; ++i) if (a[l][i] < -EPS &&
24          (!~e || rand () < 1)) e = i;
25        if (!~e) return infeasible = true;
26        pivot (l, e); } }
27    double solve () {
28      double p; std::fill (q, q + n + m, -1);
29      for (int i = 0; i < n; ++i) q[i] = i;
30      v = 0; infeasible = unbounded = false;
31      if (pre ()) return 0;
32      for (int l, e; ; pivot (l, e)) {
33        l = e = -1; for (int i = 0; i < n; ++i) if (c[i] >
34          EPS) { e = i; break; }
35        if (!~e) break; p = INF;
36        for (int i = 0; i < m; ++i) if (a[i][e] > EPS && p
37          > b[i] / a[i][e]) {
38          p = b[i] / a[i][e], l = i;
39          if (!~l) return unbounded = true, 0; }
40        for (int i = n; i < n + m; ++i) if (~q[i]) an[q[i]]
41          = b[i - n];
42        return v; }
```

## 5.4 Game theory

**Nim** For simplicity, we denote  $a_i$  as the number of stones in the  $i$ -th pile,  $M_i(S)$  as removing stones with the amount chosen in the set  $S$  from the  $i$ -th pile, and  $M_i = M_i[1, a_i]$ . Without further explanation, it is assumed that the SG function of a game  $SG = \bigoplus_{i=1}^n SG(a_i)$ .

**Nim**  $M = \bigcup_{i=1}^n M_i$ .

Normal:  $SG(n) = n$ .

Misere: The same, opposite if all piles are 1's.

**Nim (powers)** Given  $k, M = \bigcup_{i=1}^n M_i[k^m | m \geq 0]$ .

Normal: If  $k$  is odd,  $SG(n) = n \% 2$ . Otherwise,

$$SG(n) = \begin{cases} 2 & n \% (k+1) \% 2 \\ n \% (k+1) = k & \text{otherwise} \end{cases}$$



**Nim (no greater than half)**  $M = \bigcup_{i=1}^n M_i[1, \frac{a_i}{2}]$ .

Normal:  $SG(2n) = n, SG(2n+1) = SG(n)$ .

**Nim (always greater than half)**  $M = \bigcup_{i=1}^n M_i[\lceil \frac{a_i}{2} \rceil, a_i]$ .

Normal:  $SG(0) = 0, SG(n) = \lfloor \log_2 n \rfloor + 1$ .

**Nim (proper divisors)**  $M = \bigcup_{i=1}^n M_i\{x | x > 1 \wedge a_i \% x = 0\}$ .

Normal:  $SG(1) = 0, SG(n) = \max_x (n \% 2^x = 0)$ .

**Nim (divisors)**  $M = \bigcup_{i=1}^n M_i\{x | a_i \% x = 0\}$ .

Normal:  $SG(0) = 0, SG(n) = 1 + \max_x (n \% 2^x = 0)$ .

**Nim (fixed)** Given a finite set  $S$ ,  $M = \bigcup_{i=1}^n M_i(S)$ .

Normal:  $SG_1(n)$  is eventually periodic.

Given a finite set  $S$ ,  $M = \bigcup_{i=1}^n M_i(S \cup a_i)$ .

Normal:  $SG_2(n) = SG_1(n) + 1$ .

**Moore's Nim** Given  $k$ ,  $M = \bigcup \{M_{x_1} \times M_{x_2} \cdots \times M_{x_l} | l \leq k \wedge \forall i (x_i < x_{i+1})\}$ .

Normal: Sum all  $(a_i)_2$  in base  $k+1$  without carry. Lose if the result is 0.

Misere: The same, except if all piles are 1's.

**Staircase Nim** One can take any number of objects from  $a_{i+1}$  to  $a_i (i \geq 0)$ .

Normal: Lose if  $\bigoplus_{i=0}^{(n-1)/2} a_{2i+1} = 0$ .

**Lasker's Nim**  $M = \bigcup_{i=1}^n M_i \cup S_i$ . ( $S_i$ : Split a pile into two non-empty piles.)

Normal:  $SG(n) = \begin{cases} n & n \% 4 = 1, 2 \\ n+1 & n \% 4 = 3 \\ n-1 & n \% 4 = 0 \end{cases}$ .

**Kayles**  $M = \bigcup_{i=1}^n M_i[1, 2] \cup MS_i[1, 2]$ . ( $MS_i$ : Split a pile into two non-empty piles after removing stones.)

Normal: Periodic from the 72-th item with period length 12.

**Dawson's chess**  $n$  stones in a line. One can take a stone if its neighbours are not taken.

Normal: Periodic from the 52-th item with period length 34.

**Ferguson game** Two boxes with  $m$  stones and  $n$  stones. One can empty any one box and move any positive number of stones from another box to this box each step.

Normal: Lose if both  $m$  and  $n$  are odd.

**Fibonacci game**  $n$  stones. The first player may take any positive number of stones during the first move, but not all of them. After that, each player may take any positive number of stones, but less than twice the number of stones taken during the last turn.

Normal: Win if  $n$  is not a fibonacci number.

**Wythoff's game** Two piles of stones. Players take turns removing stones from one or both piles; when removing stones from both piles, the numbers of stones removed from each pile must be equal.

Normal: Lose if  $\lfloor \frac{\sqrt{5}+1}{2} |A-B| \rfloor = \min(A, B)$

**Mock turtles**  $n$  coins in a line. One can turn over any 1, 2, or 3 coins, but the rightmost coin turned must be from head to tail.

Normal:  $SG(n) = 2n + [\text{popcount}(n) \text{ is even}]$ .

**Ruler**  $n$  coins in a line. One can turn over any consecutive coins, but the rightmost coin turned must be from head to tail.

Normal:  $SG(n) = \text{lowbit}(n)$ .

**Hackenbush** The game starts with the players drawing a ground line (conventionally, but not necessarily, a horizontal line at the bottom of the paper or other playing area) and several line segments such that each line segment is connected to the ground, either directly at an endpoint, or indirectly, via a chain of other segments connected by endpoints. Any number of segments may meet at a point and thus there may be multiple paths to ground.

On his turn, a player cuts (erases) any line segment of his choice. Every line segment no longer connected to the ground by any path falls (i.e., gets erased). According to the normal play convention of combinatorial game theory, the first player who is unable to move loses.

Played exclusively with vertical stacks of line segments, also referred to as bamboo stalks, the game directly becomes Nim and can be directly analyzed as such. Divergent segments, or trees, add an additional wrinkle to the game and require use of the colon principle stating that when branches come together at a vertex, one may replace the branches by a non-branching stalk of length equal to their nim sum. This principle changes the representation of the game to the more basic version of the bamboo stalks. The last possible set of graphs that can be made are convergent ones, also known as arbitrarily rooted graphs. By using the fusion principle, we can state that all vertices on any cycle may be fused together without changing the value of the graph. Therefore, any convergent graph can also be interpreted as a simple bamboo stalk graph. By combining all three types of graphs we can add complexity to the game, without ever changing the Nim sum of the game, thereby allowing the game to take the strategies of Nim.

**Joseph cycle**  $n$  players are numbered with  $0, 1, 2, \dots, n-1$ .  $f_{1,m} = 0, f_{n,m} = (f_{n-1,m} + m) \bmod n$ .

## 5.5 Group theory

### 5.5.1 Pólya enumeration theorem

The enumeration theorem employs a multivariate generating function called the cycle index:

$$Z_G(t_1, t_2, \dots, t_n) = \frac{1}{|G|} \sum_{g \in G} t_1^{j_1(g)} t_2^{j_2(g)} \dots t_n^{j_n(g)},$$

where  $n$  is the number of elements of  $X$  and  $j_k(g)$  is the number of  $k$ -cycles of the group element  $g$  as a permutation of  $X$ .

The theorem states that the generating function  $F$  of the number of colored arrangements by weight is given by:

$$F(t) = Z_G(f(t), f(t^2), f(t^3), \dots, f(t^n)),$$

or in the multivariate case:

$$F(t_1, \dots) = Z_G(f(t_1, \dots), f(t_1^2, \dots), f(t_1^3, \dots), \dots, f(t_1^n, \dots)).$$

For instance, when separating the graphs with the number of edges, we let  $f(t) = 1 + t$ , and examine the coefficient of  $t^k$  for a graph with  $k$  edges, and when separating the necklaces with the number of beads with three different colors, we let  $f(x, y, z) = x + y + z$ , and examine the coefficient of  $x^i y^j z^k$ .

### 5.5.2 Schreier Sims

Check elements in the minimal group.

```

1 struct perm {
2     std::vector<int> P; perm() {} perm(int n) { P.
        resize(n); }
3     perm inv() const {
4         perm ret(P.size()); for(int i = 0; i < int(P.
        size()); ++i) ret.P[P[i]] = i;
5         return ret; }
6     int &operator[] (const int &dn) { return P[dn]; }
7     void resize(const size_t &sz) { P.resize(sz); }
8     size_t size() const { return P.size(); }
9     const int &operator[] (const int &dn) const { return
        P[dn]; } };
10 perm operator*(const perm &a, const perm &b) {
11     perm ret(a.size()); for(int i = 0; i < (int) a.
        size(); ++i) ret[i] = b[a[i]];
12     return ret; }
13 typedef std::vector<perm> bucket;
14 typedef std::vector<int> table;
15 typedef std::pair<int, int> pii;
16 int n, m;
17 std::vector<bucket> buckets, buckets_i; std::vector<
        table> lookup_table;
18 int fast_filter(const perm &g, bool add = true) {
19     int n = buckets.size(); perm p(g);
20     for(int i = 0; i < n; ++i) {
21         int res = lookup_table[i][p[i]];
22         if(res == -1) {
23             if(add) {
24                 buckets[i].push_back(p);
25                 buckets_i[i].push_back(p.inv());
26                 lookup_table[i][p[i]] = (int) buckets[i].size() -
                    1; }
27             return i; }
28         p = p * buckets_i[i][res]; }
29     return -1; }
30 long long calc_total_size() { long long res = 1; for
        (int i = 0; i < n; ++i) res *= buckets[i].size();
        return res; }
31 bool in_group(const perm &g) { return fast_filter(g,
        false) == -1; }
32 void solve(const bucket &gen, int _n) {
33     n = _n, m = gen.size();
34     std::vector<bucket> _buckets(n); std::swap(
        buckets, _buckets);
35     std::vector<bucket> _buckets_i(n); std::swap(
        buckets_i, _buckets_i);
36     std::vector<table> _lookup_table(n); std::swap(
        lookup_table, _lookup_table);
37     for(int i = 0; i < n; ++i) {
38         lookup_table[i].resize(n);
39         std::fill(lookup_table[i].begin(), lookup_table[i]
            .end(), -1); }
40     perm id(n);
41     for(int i = 0; i < n; ++i) id[i] = i;
42     for(int i = 0; i < n; ++i) {
43         buckets[i].push_back(id); buckets_i[i].push_back(
            id);
44         lookup_table[i][i] = 0; }
45     for(int i = 0; i < m; ++i) fast_filter(gen[i]);
46     std::queue<std::pair<pii, pii>> toUpdate;
47     for(int i = 0; i < n; ++i) for(int j = i; j < n; ++
        j)
48         for(int k = 0; k < (int) buckets[i].size(); ++k)
            for(int l = 0; l < (int) buckets[j].size(); ++
                l)
49             toUpdate.push(std::make_pair(pii(i, k), pii(j,
                l)));
50     while(!toUpdate.empty()) {
51         pii a = toUpdate.front().first, b = toUpdate.front()
            .second; toUpdate.pop();
52         int res = fast_filter(buckets[a.first][a.second] *
            buckets[b.first][b.second]);
53         if(res == -1) continue;
54         pii newPair(res, (int) buckets[res].size() - 1);
55         for(int i = 0; i < n; ++i)
56             for(int j = 0; j < (int) buckets[i].size(); ++j)
57                 if(i <= res) toUpdate.push(std::make_pair(pii(
                    i, j), newPair));
58                 if(res <= i) toUpdate.push(std::make_pair(
                    newPair, pii(i, j))); } }
```

## 5.6 Machine learning

### 5.6.1 Neural network

Train with  $ft$  features,  $n$  layers and  $m$  neurons per layer.

```

1 template<int ft = 3, int n = 2, int m = 3, int
        MAXDATA = 100000>
2 struct network {
```



```

3 double wp[n][m][ft/* or m, if larger */, bp[n][m], w
  [m], b, val[n][m], del[n][m], avg[ft + 1], sig[ft
  + 1];
4 network () {
5   std::mt19937_64 mt (time (0));
6   std::uniform_real_distribution <double> urdn (0, 2 *
  sqrt (m));
7   for (int i = 0; i < n; ++i) for (int j = 0; j < m;
  ++j) for (int k = 0; k < (i ? m : ft); ++k)
8     wp[i][j][k] = urdn (mt);
9   for (int i = 0; i < n; ++i) for (int j = 0; j < m;
  ++j) bp[i][j] = urdn (mt);
10  for (int i = 0; i < m; ++i) w[i] = urdn (mt); b =
  urdn (mt);
11  for (int i = 0; i < ft + 1; ++i) avg[i] = sig[i] =
  0; }
12 double compute (double *x) {
13   for (int j = 0; j < m; ++j) {
14     val[0][j] = bp[0][j]; for (int k = 0; k < ft; ++k)
15       val[0][j] += wp[0][j][k] * x[k];
16     val[0][j] = 1 / (1 + exp (-val[0][j]));
17   }
18   for (int i = 1; i < n; ++i) for (int j = 0; j < m;
  ++j) {
19     val[i][j] = bp[i][j]; for (int k = 0; k < m; ++k)
20       val[i][j] += wp[i][j][k] * val[i - 1][k];
21     val[i][j] = 1 / (1 + exp (-val[i][j]));
22   }
23   double res = b; for (int i = 0; i < m; ++i) res +=
  val[n - 1][i] * w[i];
24   // return 1 / (1 + exp (-res));
25   return res; }
26 void desc (double *x, double t, double eta) {
27   double o = compute (x), delo = (o - t); // * o * (1
  - o)
28   for (int j = 0; j < m; ++j) del[n - 1][j] = w[j] *
  delo * val[n - 1][j] * (1 - val[n - 1][j]);
29   for (int i = n - 2; i >= 0; --i) for (int j = 0; j <
  m; ++j) {
30     del[i][j] = 0; for (int k = 0; k < m; ++k)
31       del[i][j] += wp[i + 1][k][j] * del[i + 1][k] * val
  [i][j] * (1 - val[i][j]);
32   }
33   for (int j = 0; j < m; ++j) bp[0][j] -= eta * del
  [0][j];
34   for (int j = 0; j < m; ++j) for (int k = 0; k < ft;
  ++k) wp[0][j][k] -= eta * del[0][j] * x[k];
35   for (int i = 1; i < n; ++i) for (int j = 0; j < m;
  ++j) bp[i][j] -= eta * del[i][j];
36   for (int i = 1; i < n; ++i) for (int j = 0; j < m;
  ++j) for (int k = 0; k < m; ++k)
37     wp[i][j][k] -= eta * del[i][j] * val[i - 1][k];
38   b -= eta * delo;
39   // for (int i = 0; i < m; ++i) w[i] -= eta * delo * o
  * (1 - o) * val[i];
40   for (int i = 0; i < m; ++i) w[i] -= eta * delo * val
  [n - 1][i]; }
41 void train (double data[MAXDATA][ft + 1], int dn, int
  epoch, double eta) {
42   for (int i = 0; i < ft + 1; ++i) for (int j = 0; j <
  dn; ++j) avg[i] += data[j][i];
43   for (int i = 0; i < ft + 1; ++i) avg[i] /= dn;
44   for (int i = 0; i < ft + 1; ++i) for (int j = 0; j <
  dn; ++j) sig[i] += (data[j][i] - avg[i]) * (data[j][i] - avg
  [i]);
45   for (int i = 0; i < ft + 1; ++i) sig[i] = sqrt (sig[
  i] / dn);
46   for (int i = 0; i < ft + 1; ++i) for (int j = 0; j <
  dn; ++j) data[j][i] = (data[j][i] - avg[i]) / sig[i];
47   for (int cnt = 0; cnt < epoch; ++cnt) for (int test
  = 0; test < dn; ++test)
48     desc (data[test], data[test][ft], eta); }
49 double predict (double *x) {
50   for (int i = 0; i < ft; ++i) x[i] = (x[i] - avg[i])
  / sig[i];
51   return compute (x) * sig[ft] + avg[ft]; }
52 std::string to_string () {
53   std::ostringstream os; os << std::fixed << std:::
  setprecision (16);
54   for (int i = 0; i < n; ++i) for (int j = 0; j < m;
  ++j) for (int k = 0; k < (i ? m : ft); ++k)
55     os << wp[i][j][k] << " ";
56   for (int i = 0; i < n; ++i) for (int j = 0; j < m;
  ++j) os << bp[i][j] << " ";
57   for (int i = 0; i < m; ++i) os << w[i] << " "; os <<
  b << " ";
58   for (int i = 0; i < ft + 1; ++i) os << avg[i] << " ";
59   for (int i = 0; i < ft + 1; ++i) os << sig[i] << " ";
60   return os.str (); }
61 void read (const std::string &str) {
62   std::istringstream is (str);
63   for (int i = 0; i < n; ++i) for (int j = 0; j < m;
  ++j) for (int k = 0; k < (i ? m : ft); ++k)
64     is >> wp[i][j][k];
65   for (int i = 0; i < n; ++i) for (int j = 0; j < m;
  ++j) is >> bp[i][j];
66   for (int i = 0; i < m; ++i) is >> w[i]; is >> b;
67   for (int i = 0; i < ft + 1; ++i) is >> avg[i];
68   for (int i = 0; i < ft + 1; ++i) is >> sig[i]; } }

```

## 5.7 Primality

### 5.7.1 Miller Rabin primality test

Test whether a certain integer is prime.

```

1 struct miller_rabin {
2   int BASE[12] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
  31, 37};
3   bool check (const long long &p, const long long &b) {
4     long long n = p - 1;
5     for (; ~n & 1; n >= 1);
6     long long res = llfpm (b, n, p);
7     for (; n != p - 1 && res != 1 && res != p - 1; n <=
  1)
8       res = mul_mod (res, res, p);
9     return res == p - 1 || (n & 1) == 1; }
10  bool solve (const long long &n) {
11    if (n < 2) return false;
12    if (n < 4) return true;
13    if (~n & 4) return false;
14    for (int i = 0; i < 12 && BASE[i] < n; ++i) if (!
  check (n, BASE[i])) return false;
15    return true; } };

```

### 5.7.2 Pollard's Rho algorithm

Factorize an integer.

```

1 struct pollard_rho {
2   miller_rabin is_prime; const long long thr = 13E9;
3   long long factor (const long long &n, const long long
  &seed) {
4     long long x = rand () % (n - 1) + 1, y = x;
5     for (int head = 1, tail = 2; ; ) {
6       x = mul_mod (x, x, n);
7       x = (x + seed) % n;
8       if (x == y) return n;
9       long long ans = std::gcd (std::abs (x - y), n);
10      if (ans > 1 && ans < n) return ans;
11      if (++head == tail) { y = x; tail <= 1; } } }
12  void search (const long long &n, std::vector <long
  long> &div) {
13    if (n > 1) {
14      if (is_prime.solve (n)) div.push_back (n);
15      else {
16        long long fac = n;
17        for (; fac >= n; fac = factor (n, rand () % (n -
  1) + 1));
18        search (n / fac, div); search (fac, div); } } }
19  std::vector <long long> solve (const long long &n) {
20    std::vector <long long> ans;
21    if (n > thr) search (n, ans);
22    else {
23      long long rem = n;
24      for (long long i = 2; i * i <= rem; ++i)
25        while (!rem % i) { ans.push_back (i); rem /= i;
  }
26      if (rem > 1) ans.push_back (rem); }
27    return ans; } };

```

### 5.7.3 SQUFOF

Factorize an integer in  $O(N^{\frac{1}{4}})$ .

```

1 namespace NT {
2   typedef unsigned int l;
3   typedef unsigned long long ll;
4   inline ll mul (ll const &a, ll const &b, ll const &
  mod) {
5     ll ret = a * (ll) b - (ll) ((long double) a * b /
  mod - 1.1) * mod;
6     if (-ret < ret) ret = mod - 1 - (-ret - 1) % mod;
7     else ret %= mod;
8     return ret; }
9   inline ll pow (ll const &a, ll const &b, ll const &
  mod) {
10    ll ret = 1, base = a;
11    for (l i = 0; b >> i; ++i) {
12      if ((b >> i) & 1) ret = mul (ret, base, mod);
13      base = mul (base, base, mod); }
14    return ret; }
15  bool miller_rabin_single (ll const &x, ll base) {
16    if (x < 4) return x > 1;
17    if (x % 2 == 0) return false;
18    if ((base % x) == 0) return true;
19    ll xml = x - 1; l j = (1) __builtin_ctzl (xml);
20    ll t = pow (base, xml >> j, x);
21    if (t == 1 || t == xml) return true;
22    for (l k = 1; k < j; ++k) {
23      t = mul (t, t, x);
24      if (t == xml) return true;
25      if (t <= 1) break; }
26    return false; }
27  bool miller_rabin_multi (ll const &x, ...) {
28    va_list args; va_start (args, x); ll base; bool ret
  = true;
29    while (ret && (base = va_arg (args, ll))) ret &=
  miller_rabin_single (x, base);
30    va_end (args); return ret; }
31  bool miller_rabin (ll const &x) {
32    if (x < 316349281) return miller_rabin_multi (x,
  11000544, 31481107, 0);
33    if (x < 4759123141ull) return miller_rabin_multi (x,
  2, 7, 61, 0);

```

```

34 return miller_rabin_multi(x, 2, 325, 9375, 28178,
    450775, 9780504, 1795265022, 0); }
35 ll isqrt(ll const &x) {
36     ll ret = (ll) (sqrtl(x));
37     while (ret > 0 && ret * ret > x) --ret;
38     while (x - ret * ret > 2 * ret) ++ret;
39     return ret; }
40 ll icbrt(ll const &x) {
41     ll ret = (ll) (cbrt(x));
42     while (ret > 0 && ret * ret * ret > x) --ret;
43     while (x - ret * ret * ret > 3 * ret * (ret + 1)) ++
        ret;
44     return ret; }
45 std::vector<ll> saved;
46 ll squfof_iter_better(ll const &x, ll const &k, ll
    const &it_max, ll cutoff_div) {
47     if(std::gcd((ll) k, x) != 1) return std::gcd((
        ll) k, x);
48     saved.clear(); ll scaledn = k * x;
49     if (scaledn >> 62) return 1;
50     ll sqrt_n = isqrt(scaledn), cutoff = isqrt(2 * sqrt_n
        ) / cutoff_div;
51     ll q0 = 1, p1 = sqrt_n, q1 = scaledn - p1 * p1;
52     if (q1 == 0) { ll factor = std::gcd(x, (ll) p1);
        return factor == x ? 1 : factor; }
53     ll multiplier = 2 * k, coarse_cutoff = cutoff *
        multiplier;
54     ll i, j, p0 = 0, sqrtq = 0;
55     for (i = 0; i < it_max; ++i) {
56         ll q = 1, bits, tmp = sqrt_n + p1 - q1;
57         if (tmp >= q1) q += tmp / q1;
58         p0 = q * q1 - p1;
59         q0 = q0 + (p1 - p0) * q;
60         if (q1 < coarse_cutoff) {
61             tmp = q1 / std::gcd(q1, multiplier);
62             if (tmp < cutoff) saved.push_back((ll) tmp); }
63         bits = 0; tmp = q0;
64         while (!(tmp & 1)) { bits++; tmp >>= 1; }
65         if (!(bits & 1) && ((tmp & 7) == 1)) {
66             sqrtq = (ll) isqrt(q0);
67             if (sqrtq * sqrtq == q0) {
68                 for (j = 0; j < saved.size(); ++j)
69                     if (saved[j] == sqrtq) break;
70                 if (j == saved.size()) break; } }
71         tmp = sqrt_n + p0 - q0; q = 1;
72         if (tmp >= q0) q += tmp / q0;
73         p1 = q * q0 - p0; q1 = q1 + (p0 - p1) * q;
74         if (q0 < coarse_cutoff) {
75             tmp = q0 / std::gcd(q0, multiplier);
76             if (tmp < cutoff) saved.push_back((ll) tmp); } }
77     if (sqrtq == 1 || i == it_max) return 1;
78     q0 = sqrtq; p1 = p0 + sqrtq * ((sqrt_n - p0) / sqrtq)
        ;
79     q1 = (scaledn - (ll) p1 * (ll) p1) / (ll) q0;
80     for (j = 0; j < it_max; ++j) {
81         ll q = 1, tmp = sqrt_n + p1 - q1;
82         if (tmp >= q1) q += tmp / q1;
83         p0 = q * q1 - p1;
84         q0 = q0 + (p1 - p0) * q;
85         if (p0 == p1) { q0 = q1; break; }
86         q = 1; tmp = sqrt_n + p0 - q0;
87         if (tmp >= q0) q += tmp / q0;
88         p1 = q * q0 - p0;
89         q1 = q1 + (p0 - p1) * q;
90         if (p0 == p1) break; }
91     ll factor = std::gcd((ll) q0, x);
92     if (factor == x) factor = 1;
93     return factor; }
94 ll squfof(ll const &x) {
95     static ll multipliers[16] = {1, 3, 5, 7, 11, 15, 21,
        33, 35, 55, 77, 105, 165, 231, 385, 1155};
96     ll cbrt_x = icbrt(x);
97     if (cbrt_x * cbrt_x * cbrt_x == x) return cbrt_x;
98     ll iter_lim = 300;
99     for (ll iter_fact = 1; iter_fact < 20000; iter_fact
        *= 4) {
100         for (ll i = 0; i < 16; ++i) {
101             ll k = multipliers[i];
102             if ((~(ll) 0) / k <= x) continue;
103             ll const it_max = iter_fact * iter_lim;
104             ll factor = squfof_iter_better(x, k, it_max, 1);
105             if (factor == 1 || factor == x) continue;
106             return factor; } } return 1; }
107 #define trial(i) while (x % i == 0) { x /= i; ret.
        push_back(i); }
108 std::vector<ll> factorize(ll x) {
109     std::vector<ll> ret;
110     const ll trial_limit = 5000;
111     trial(2); trial(3);
112     for (ll i = 5, j = 2; i < trial_limit && i * i <= x;
        i += j, j = 6 - j) trial(i);
113     if (x > 1) {
114         static std::stack<ll> s; s.push(x);
115         while (!s.empty()) {
116             x = s.top(); s.pop();
117             if (!miller_rabin(x)) {
118                 ll factor = squfof(x);
119                 s.push(factor); s.push(x / factor);
120             } else ret.push_back(x); } }
121     std::sort(ret.begin(), ret.end()); return ret; }

```

## 5.8 Recurrence relation

### 5.8.1 Berlekamp Massey algorithm

Find the recursive equation with the first elements of the sequence in  $O(n^2)$ .

Sample input: {1, 1, 2, 3}.

Sample output: {1, 1000000006, 1000000006} mod  $10^9 + 7$ , i.e.  $a_i - a_{i-1} - a_{i-2} = 0$ .

```

1 struct berlekamp-massey {
2     struct poly { std::vector<int> a; poly() { a.clear()
        ; }
3         poly(std::vector<int> &a) : a(a) {}
4         int length() const { return a.size(); }
5         poly move(int d) { std::vector<int> na(d, 0);
6             na.insert(na.end(), a.begin(), a.end());
7             return poly(na); }
8         int calc(std::vector<int> &d, int pos) { int ret =
9             0;
10            for (int i = 0; i < (int) a.size(); ++i) {
11                if ((ret += 1LL * d[pos - i] * a[i] % MOD) >= MOD)
12                    ret -= MOD; } }
13            return ret; }
14        poly operator - (const poly &b) {
15            std::vector<int> na(std::max(this->length(),
16                b.length()));
17            for (int i = 0; i < (int) na.size(); ++i) {
18                int aa = i < this->length() ? this->a[i] : 0;
19                bb = i < b.length() ? b.a[i] : 0;
20                na[i] = (aa + MOD - bb) % MOD; }
21            return poly(na); }
22        poly operator * (const int &c, const poly &p) {
23            std::vector<int> na(p.length());
24            for (int i = 0; i < (int) na.size(); ++i) {
25                na[i] = 1LL * c * p.a[i] % MOD; }
26            return na; }
27        std::vector<int> solve(vector<int> a) {
28            int n = a.size(); poly s, b;
29            s.a.push_back(1), b.a.push_back(1);
30            for (int i = 0, j = -1, ld = 1; i < n; ++i) {
31                int d = s.calc(a, i); if (d) {
32                    if ((s.length() - 1) * 2 <= i) {
33                        poly ob = b; b = s;
34                        s = s - 1LL * d * inverse(ld) % MOD * ob.move(i
35                            - j);
36                        j = i; ld = d;
37                    } else {
38                        s = s - 1LL * d * inverse(ld) % MOD * b.move(i
39                            - j); } } }
40            return s.a; } };

```

### 5.8.2 Linear Recurrence

Find the  $n$ -th element of a linear recurrence.

Sample input: {2, 1}, {2, 1} ( $a_1 = 2, a_2 = 1, a_n = 2a_{n-1} + a_{n-2}$ ).

Sample output:  $calc(3) = 5, calc(10007) = 959155122 \bmod 10^9 + 7$ .

```

1 struct linear_rec {
2     const int LOG = 30, MOD = 1E9 + 7; int n;
3     std::vector<int> first, trans;
4     std::vector<std::vector<int>> bin;
5     std::vector<int> add(std::vector<int> &a, std:::
        vector<int> &b) {
6         std::vector<int> result(n * 2 + 1, 0);
7         for (int i = 0; i <= n; ++i) for (int j = 0; j <= n;
8             ++j)
9             if ((result[i + j] += 1LL * a[i] * b[j] % MOD) >=
10                 MOD) result[i + j] -= MOD;
11         for (int i = 2 * n; i > n; --i) {
12             for (int j = 0; j < n; ++j)
13                 if ((result[i - 1 - j] += 1LL * result[i] * trans[
14                     j] % MOD) >= MOD) result[i - 1 - j] -= MOD;
15             result[i] = 0; }
16         result.erase(result.begin() + n + 1, result.end());
17         return result; }
18     linear_rec(const std::vector<int> &first, const std
19         :vector<int> &trans) : first(first), trans(
20             trans) {
21         n = first.size(); std::vector<int> a(n + 1, 0); a
22             [1] = 1; bin.push_back(a);
23         for (int i = 1; i < LOG; ++i) bin.push_back(add(bin
24             [i - 1], bin[i - 1])); }
25     int solve(int k) {
26         std::vector<int> a(n + 1, 0); a[0] = 1;
27         for (int i = 0; i < LOG; ++i) if (k >> i & 1) a =
28             add(a, bin[i]);
29         int ret = 0;
30         for (int i = 0; i < n; ++i) if ((ret += (long long)
31             a[i + 1] * first[i] % MOD) >= MOD) ret -= MOD;
32         return ret; } };

```

## 5.9 Sequence manipulation

### 5.9.1 Discrete Fourier transform

Complexity  $O(n \log n)$ .

```

1 template<int MAXN = 1000000>
2 struct dft {
3     typedef std::complex<double> complex;
4     complex e[2][MAXN];
5     int init(int n) {
6         int len = 1; for (; len <= 2 * n; len <<= 1);

```

```

7  for (int i = 0; i < len; ++i) {
8      e[0][i] = complex (cos (2 * PI * i / len), sin (2 *
9          PI * i / len));
10     e[1][i] = complex (cos (2 * PI * i / len), -sin (2
11         * PI * i / len)); }
12     return len; }
13 void solve (complex *a, int n, int f) {
14     for (int i = 0, j = 0; i < n; ++i) {
15         if (i > j) std::swap (a[i], a[j]);
16         for (int t = n >> 1; (j ^= t) < t; t >>= 1); }
17     for (int i = 2; i <= n; i <= 1)
18         for (int j = 0; j < n; j += i)
19             for (int k = 0; k < (i >> 1); ++k) {
20                 complex A = a[j + k];
21                 complex B = e[f][n / i * k] * a[j + k + (i >> 1)
22                     ];
23                 a[j + k] = A + B;
24                 a[j + k + (i >> 1)] = A - B; }
25     if (f == 1) {
26         for (int i = 0; i < n; ++i) a[i] = complex (a[i].
27             real () / n, a[i].imag ()); } } }

```

### 5.9.2 Fast Walsh-Hadamard transform

Compute  $C_k = \sum_{i \oplus j = k} A_i B_j$ .

```

1  void fwt (int *a, int n, int w) {
2      for (int i = 1; i < n; i <= 1)
3          for (int j = 0; j < n; j += i < 1) {
4              for (int k = 0; k < i; ++k) {
5                  int x = a[j + k], y = a[i + j + k];
6                  if (w) {
7                      /* xor : a[j + k] = (x + y) / 2, a[i + j + k] = (x
8                          - y) / 2; and : a[j + k] = x - y; or : a[i +
9                          j + k] = y - x; */
10                     } else {
11                         /* xor : a[j + k] = x + y, a[i + j + k] = x - y;
12                             and : a[j + k] = x + y; or : a[i + j + k] = x
13                             + y; */
14                     } } } }

```

### 5.9.3 Number theoretic transform

Complexity  $O(n \log n)$ . In case of a non-NTT prime module, perform the multiplication on 3 different NTT prime modules and use crt to merge the result.

```

1  template <int MAXN = 1000000, int LOGN = 20>
2  struct ntt {
3      int MOD[3] = {1045430273, 1051721729, 1053818881},
4          PRT[3] = {3, 6, 7};
5      int exp[LOGN][MAXN][2];
6      void init (int n = MAXN, int mod = 998244353, int prt
7          = 3) {
8          for (int i = 2, cnt = 0; i <= n; i <= 1, ++cnt) {
9              exp[cnt][0][0] = exp[cnt][0][1] = 1; exp[cnt][1][0]
10                 = fpm (prt, (mod - 1) / i, mod);
11                 exp[cnt][1][1] = fpm (exp[cnt][1][0], mod - 2, mod)
12                 ;
13             for (int k = 2; k < (i >> 1); ++k) for (int t = 0;
14                 t < 2; ++t)
15                 exp[cnt][k][t] = int (1ll * exp[cnt][k - 1][t] *
16                     exp[cnt][1][t] % mod); } }
17     void solve (int *a, int n = MAXN, int f = 0, int mod
18         = 998244353) {
19         for (int i = 0, j = 0; i < n; ++i) {
20             if (i > j) std::swap (a[i], a[j]);
21             for (int t = n >> 1; (j ^= t) < t; t >>= 1); }
22         for (int i = 2, cnt = 0; i <= n; i <= 1, ++cnt)
23             for (int j = 0; j < n; j += i) for (int k = 0; k <
24                 (i >> 1); ++k) {
25                 int &a = a[j + k], &b = a[j + k + (i >> 1)];
26                 int A = a, B = int (1ll * b * exp[cnt][k][f] %
27                     mod);
28                 a = A + B < mod ? A + B : A + B - mod;
29                 b = A - B >= 0 ? A - B : A - B + mod; }
30         if (f == 1) {
31             int rev = fpm (n, mod - 2, mod);
32             for (int i = 0; i < n; ++i) a[i] = int (1ll * a[i]
33                 * rev % mod); } }
34     int crt (int *a, int mod) {
35         static int inv[3][3];
36         for (int i = 0; i < 3; ++i) for (int j = 0; j < 3;
37             ++j)
38             inv[i][j] = (int) inverse (MOD[i], MOD[j]);
39         static int x[3];
40         for (int i = 0; i < 3; ++i) { x[i] = a[i];
41             for (int j = 0; j < i; ++j) {
42                 int t = (x[i] - x[j] + MOD[i]) % MOD[i];
43                 if (t < 0) t += MOD[i];
44                 x[i] = int (1ll * t * inv[j][i] % MOD[i]); } }
45         int sum = 1, ret = x[0] % mod;
46         for (int i = 1; i < 3; ++i) {
47             sum = int (1ll * sum * MOD[i - 1] % mod);
48             ret += int (1ll * x[i] * sum % mod);
49             if (ret >= mod) ret -= mod; }
50         return ret; } }

```

### 5.9.4 Polynomial operation

1. inverse: Find a polynomial  $b$  so that  $a(x)b(x) \equiv 1 \pmod{x^n}$  mod  $mod$ . Note:  $n$  must be a power of 2. The max length of the array should be at least twice the actual length.

2. sqrt: Find a polynomial  $b$  so that  $b^2(x) \equiv a(x) \pmod{x^n}$  mod  $mod$ . Note:  $n \geq 2$  must be a power of 2. The max length of the array should be at least twice the actual length.
3. divide: Given polynomial  $a$  and  $b$  with degree  $n$  and  $m$  respectively, find  $a(x) = d(x)b(x) + r(x)$  with  $\deg(d) \leq n - m$  and  $\deg(r) < m$ . The max length of the array should be at least four times the actual length.

```

1  template <int MAXN = 1000000>
2  struct polynomial {
3      ntt <MAXN> tr;
4      void inverse (int *a, int *b, int n, int mod, int prt
5          ) {
6          static int c[MAXN]; b[0] = ::inverse (a[0], mod); b
7          [1] = 0;
8          for (int m = 2, i; m <= n; m <= 1) {
9              std::copy (a, a + m, c);
10             std::fill (b + m, b + m + m, 0); std::fill (c + m,
11                 c + m + m, 0);
12             tr.solve (c, m + m, 0, mod, prt); tr.solve (b, m +
13                 m, 0, mod, prt);
14             for (int i = 0; i < m + m; ++i) b[i] = 1LL * b[i] *
15                 (2 - 1LL * b[i] * c[i] % mod + mod) % mod;
16             tr.solve (b, m + m, 1, mod, prt); std::fill (b + m,
17                 b + m + m, 0); } }
18     void sqrt (int *a, int *b, int n, int mod, int prt) {
19         static int d[MAXN], ib[MAXN]; b[0] = 1; b[1] = 0;
20         int i2 = ::inverse (2, mod), m, i;
21         for (int m = 2; m <= n; m <= 1) {
22             std::copy (a, a + m, d);
23             std::fill (d + m, d + m + m, 0); std::fill (b + m,
24                 b + m + m, 0);
25             tr.solve (d, m + m, 0, mod, prt); inverse (b, ib, m
26                 , mod, prt);
27             tr.solve (ib, m + m, 0, mod, prt); tr.solve (b, m +
28                 m, 0, mod, prt);
29             for (int i = 0; i < m + m; ++i) b[i] = (1LL * b[i]
30                 * i2 + 1LL * i2 * d[i] % mod * ib[i]) % mod;
31             tr.solve (b, m + m, 1, mod, prt); std::fill (b + m,
32                 b + m + m, 0); } }
33     void divide (int *a, int n, int *b, int m, int *d,
34         int *r, int mod, int prt) {
35         static int u[MAXN], v[MAXN]; while (!b[m - 1]) --m;
36         int p = 1, t = n - m + 1; while (p < t < 1) p <=
37             1;
38         std::fill (u, u + p, 0); std::reverse_copy (b, b + m
39             , u); inverse (u, v, p, mod, prt);
40         std::fill (v + t, v + p, 0); tr.solve (v, p, 0, mod,
41             prt); std::reverse_copy (a, a + n, u);
42         std::fill (u + t, u + p, 0); tr.solve (u, p, 0, mod,
43             prt);
44         for (int i = 0; i < p; ++i) u[i] = 1LL * u[i] * v[i]
45             % mod;
46         tr.solve (u, p, 1, mod, prt); std::reverse (u, u + t
47             );
48         for (p = 1; p < n; p <= 1); std::fill (u + t, u + p
49             , 0);
50         tr.solve (u, p, 0, mod, prt); std::copy (b, b + m, v
51             );
52         std::fill (v + m, v + p, 0); tr.solve (v, p, 0, mod,
53             prt);
54         for (int i = 0; i < p; ++i) u[i] = 1LL * u[i] * v[i]
55             % mod;
56         tr.solve (u, p, 1, mod, prt);
57         for (int i = 0; i < m; ++i) r[i] = (a[i] - u[i] +
58             mod) % mod;
59         std::fill (r + m, r + p, 0); } }

```

## 6 String

### 6.1 Decomposition

#### 6.1.1 Lyndon word

A  $k$ -ary Lyndon word of length  $n > 0$  is an  $n$ -character string over an alphabet of size  $k$ , and which is the unique minimum element in the lexicographical ordering of all its rotations. Being the singularly smallest rotation implies that a Lyndon word differs from any of its non-trivial rotations, and is therefore aperiodic.

Alternately, a Lyndon word has the property that it is nonempty and, whenever it is split into two nonempty substrings, the left substring is always lexicographically less than the right substring. That is, if  $w$  is a Lyndon word, and  $w = uv$  is any factorization into two substrings, with  $u$  and  $v$  understood to be non-empty, then  $u < v$ . This definition implies that a string  $w$  of length  $\geq 2$  is a Lyndon word if and only if there exist Lyndon words  $u$  and  $v$  such that  $u < v$  and  $w = uv$ . Although there may be more than one choice of  $u$  and  $v$  with this property, there is a particular choice, called the standard factorization, in which  $v$  is as long as possible.

Lyndon words correspond to aperiodic necklace class representatives and can thus be counted with Moreau's necklace-counting function.

Duval provides an efficient algorithm for listing the Lyndon words of length at most  $n$  with a given alphabet size  $s$  in lexicographic order. If  $w$  is one of the words in the sequence, then the next word after  $w$  can be found by the following steps:

1. Repeat the symbols from  $w$  to form a new word  $x$  of length exactly  $n$ , where the  $i$ th symbol of  $x$  is the same as the symbol at position  $(i \bmod \text{length}(w))$  of  $w$ .
2. As long as the final symbol of  $x$  is the last symbol in the sorted ordering of the alphabet, remove it, producing a shorter word.
3. Replace the final remaining symbol of  $x$  by its successor in the sorted ordering of the alphabet.

The sequence of all Lyndon words of length at most  $n$  can be generated in time proportional to the length of the sequence.



According to the Chen-Fox-Lyndon theorem, every string may be formed in a unique way by concatenating a sequence of Lyndon words, in such a way that the words in the sequence are nonincreasing lexicographically. The final Lyndon word in this sequence is the lexicographically smallest suffix of the given string. A factorization into a nonincreasing sequence of Lyndon words (the so-called Lyndon factorization) can be constructed in linear time.

Given a string  $S$  of length  $N$ , one should proceed with the following steps:

1. Let  $m$  be the index of the symbol-candidate to be appended to the already collected symbols. Initially,  $m = 1$  (indices of symbols in a string start from zero).
2. Let  $k$  be the index of the symbol we would compare others to. Initially,  $k = 0$ .
3. While  $k$  and  $m$  are less than  $N$ , compare  $S[k]$  (the  $k$ -th symbol of the string  $S$ ) to  $S[m]$ . There are three possible outcomes:
  - (a)  $S[k]$  is equal to  $S[m]$ : append  $S[m]$  to the current collected symbols. Increment  $k$  and  $m$ .
  - (b)  $S[k]$  is less than  $S[m]$ : if we append  $S[m]$  to the current collected symbols, we'll get a Lyndon word. But we can't add it to the result list yet because it may be just a part of a larger Lyndon word. Thus, just increment  $m$  and set  $k$  to 0 so the next symbol would be compared to the first one in the string.
  - (c)  $S[k]$  is greater than  $S[m]$ : if we append  $S[m]$  to the current collected symbols, it will be neither a Lyndon word nor a possible beginning of one. Thus, add the first  $m - k$  collected symbols to the result list, remove them from the string, set  $m$  to 1 and  $k$  to 0 so that they point to the second and the first symbol of the string respectively.
4. When  $m > N$ , it is essentially the same as encountering minus infinity, thus, add the first  $m - k$  collected symbols to the result list after removing them from the string, set  $m$  to 1 and  $k$  to 0, and return to the previous step.
5. Add  $S$  to the result list.

If one concatenates together, in lexicographic order, all the Lyndon words that have length dividing a given number  $n$ , the result is a de Bruijn sequence, a circular sequence of symbols such that each possible length- $n$  sequence appears exactly once as one of its contiguous subsequences.

```
1 std::vector<int> mnsuf (char *s, int *mn, int n) {
2   std::vector<int> ret;
3   for (int i = 0; i < n; ) {
4     int j = i, k = i + 1; mn[i] = i;
5     for (; k < n && s[j] <= s[k]; ++k)
6       if (s[j] == s[k]) mn[k] = mn[j] + k - j, ++j;
7     else mn[k] = j = i;
8     for (; i <= j; i += k - j) ret.push_back (i);
9   } return ret; }

10 std::vector<int> mxsuf (char *s, int *mx, int n) {
11   std::vector<int> ret; std::fill (mx, mx + n, -1);
12   for (int i = 0; i < n; ) {
13     int j = i, k = i + 1; if (mx[i] == -1) mx[i] = i;
14     for (; k < n && s[j] >= s[k]; ++k) {
15       j = s[j] == s[k] ? j + 1 : i;
16       if (mx[k] == -1) mx[k] = i;
17     } for (; i <= j; i += k - j) ret.push_back (i);
18   } return ret; }
```

## 6.2 Matching

### 6.2.1 Exkmp

$$a[i] = lcp(s + i, t), b[i] = lcp(t + i, t).$$

```
1 void exkmp (char *s, int *a, int n) {
2   a[0] = n; int p = 0, r = 0;
3   for (int i = 1; i < n; ++i) {
4     a[i] = (r > i) ? std::min (r - i, a[i - p]) : 0;
5     while (i + a[i] < n && s[i + a[i]] == s[a[i]]) ++a[i];
6     if (r < i + a[i]) r = i + a[i], p = i; } }
7 void mat (char *s, char *t, int *a, int *b, int n, int m) {
8   exkmp (t, b, m); int p = 0, r = 0;
9   for (int i = 0; i < n; ++i) {
10    a[i] = (r > i) ? std::min (r - i, b[i - p]) : 0;
11    while (i + a[i] < n && a[i] < m && s[i + a[i]] == t[a[i]]) ++a[i];
12    if (r < i + a[i]) r = i + a[i], p = i; } }
```

### 6.2.2 Minimal string rotation

Return the start index.

```
1 int min_rep (char *s, int l) {
2   int i, j, k;
3   i = 0; j = 1; k = 0;
4   while (i < l && j < l) {
5     k = 0; while (s[i + k] == s[j + k] && k < l) ++k;
6     if (k == l) return i;
7     if (s[i + k] > s[j + k])
8       if (i + k + 1 > j) i = i + k + 1;
9     else i = j + 1;
10    else if (j + k + 1 > i) j = j + k + 1;
11    else j = i + 1;
12    if (i < l) return i; else return j; }
```

## 6.3 Palindrome

### 6.3.1 Manacher

Odd palindromes only.

```
1 char s[0..n] = '#1#2#3#';
2 int p[n], id, mx;
3 for (int i = 1; i <= n; ++i) {
4   if (mx > i) p[i] = std::min (p[2 * id - 1], mx - i);
5   else p[i] = 1;
6   while (s[i - p[i]] == s[i + p[i]]) ++p[i];
7   if (i + p[i] > mx) {
8     mx = i + p[i]; id = i; } }
```

### 6.3.2 Palindromic tree

Usage:

1. extend: Return whether the tree has generated a new node.
2. odd, even: Root of two trees.
3. last: The node representing the last char.
4. node::len: The length of the palindromic string of the node.

```
1 template<int MAXN = 1000000, int MAXC = 26>
2 struct palindromic_tree {
3   struct node {
4     node *child[MAXC], *fail; int len;
5     node (int len) : fail (NULL), len (len) {
6       memset (child, NULL, sizeof (child)); }
7   } node_pool[MAXN * 2]; int tot_node;
8   int size, text[MAXN];
9   node *odd, *even, *last;
10  node *match (node *now) {
11    for (; text[size - now -> len - 1] != text[size];
12      now = now -> fail);
13    return now; }
14  bool extend (int token) {
15    text[++size] = token; node *now = match (last);
16    if (now -> child[token])
17      return last = now -> child[token], false;
18    last = now -> child[token] = new (tot_node++) node (
19      now -> len + 2);
20    if (now == odd) last -> fail = even;
21    else {
22      now = match (now -> fail);
23      last -> fail = now -> child[token]; }
24    return true; }
25  void init () {
26    text[size = 0] = -1; tot_node = node_pool;
27    last = even = new (tot_node++) node (0); odd = new (
28      tot_node++) node (-1);
29    even -> fail = odd; }
30  palindromic_tree () { init (); } }
```

## 6.4 Suffix

### 6.4.1 Suffix array (SAIS)

Ensure that  $\text{str}[n] \geq 0$  is the unique lexicographically smallest character in  $\text{str}$ .

1.  $\text{sa}[i]$ : The beginning position of the  $i$ -th smallest suffix. Note that  $\text{sa}[0] = n$ .
2.  $\text{rk}[i]$ : The rank of the suffix beginning at position  $i$ .
3.  $\text{ht}[i]$ : The longest common prefix of  $\text{sa}[i]$  and  $\text{sa}[i - 1]$ .

```
1 template<int MAXN = 1200000>
2 struct sa {
3   int sa[MAXN], rk[MAXN], ht[MAXN], s[MAXN << 1], t[
4     MAXN << 1], p[MAXN], cnt[MAXN], cur[MAXN];
5   #define push_s(x) sa[cur[s[x]]++] = x
6   #define push_l(x) sa[cur[s[x]]++] = x
7   #define induced_sort(v) std::fill_n (sa, n, -1); std::fill_n (cnt, m, 0); \
8     for (int i = 0; i < n; ++i) cnt[s[i]]++; \
9     for (int i = 1; i < m; ++i) cnt[i] += cnt[i - 1]; \
10    for (int i = 0; i < m; ++i) cur[i] = cnt[i - 1]; \
11    for (int i = n - 1; i >= 0; --i) push_s (v[i]); \
12    for (int i = 1; i < m; ++i) cur[i] = cnt[i - 1]; \
13    for (int i = 0; i < n; ++i) if (sa[i] > 0 && t[sa[i] - 1]) push_l (sa[i] - 1); \
14    for (int i = 0; i < m; ++i) cur[i] = cnt[i - 1]; \
15    for (int i = n - 1; i >= 0; --i) if (sa[i] > 0 && !t[sa[i] - 1]) push_s (sa[i] - 1);
16  void sais (int n, int m, int *s, int *t, int *p) {
17    int nl = t[n - 1] = 0, ch = rk[0] = -1, *sl = s + n;
18    for (int i = n - 2; i >= 0; --i) t[i] = s[i] == s[i + 1] ? t[i + 1] : s[i] > s[i + 1];
19    for (int i = 1; i < n; ++i) rk[i] = t[i - 1] && !t[i] ? (p[nl] = i, nl++) : -1;
20    induced_sort (p);
21    for (int i = 0, x, y; i < n; ++i) if (~x = rk[sa[i]]) {
22      if (ch < 1 || p[x + 1] - p[x] != p[y + 1] - p[y]) ++ch;
23      else for (int j = p[x], k = p[y]; j <= p[x + 1]; ++j, ++k)
24        if ((s[j] << 1 | t[j]) != (s[k] << 1 | t[k])) { ++ch; break; }
25      sl[y = x] = ch;
26      if (ch + 1 < nl) sais (nl, ch + 1, sl, t + n, p + nl);
27    } else for (int i = 0; i < nl; ++i) sa[sl[i]] = i;
28    for (int i = 0; i < nl; ++i) sl[i] = p[sa[i]];
29    induced_sort (sl); }
30 template<typename T> int map_char_to_int (int n, const T *str) {
```



```

30 int m = *std::max_element (str, str + n);
31 std::fill_n (rk, m + 1, 0);
32 for (int i = 0; i < n; ++i) rk[str[i]] = 1;
33 for (int i = 0; i < m; ++i) rk[i + 1] += rk[i];
34 for (int i = 0; i < n; ++i) s[i] = rk[str[i]] - 1;
35 return rk[m]; }
36 template <typename T> void suffix_array (int n, const
    T *str) {
37     int m = map_char_to_int (++n, str);
38     sais (n, m, s, t, p);
39     for (int i = 0; i < n; ++i) rk[sa[i]] = i;
40     for (int i = 0, h = ht[0] = 0; i < n - 1; ++i) {
41         int j = sa[rk[i] - 1];
42         while (i + h < n && j + h < n && s[i + h] == s[j +
            h]) ++h;
43         if (ht[rk[i]] = h) --h; } } };

```

## 6.4.2 Suffix automaton

Usage:

1. head: The first state.
2. tail: The last state. Terminating states can be reached via visiting the ancestors of tail.
3. len: The longest length of the string in the state. The shortest one is parent  $\rightarrow$  len + 1.
4. first: The first location in the string where the state can be reached.
5. cnt: The number of times a particular substring  $S$  of substrings corresponding to the state appears.
6. parent: the parent link. A larger set of all matching possibilities.
7. go: the automaton link.

```

1 template <int MAXN = 1000000, int MAXC = 26>
2 struct suffix_automaton {
3     struct state {
4         int len, first, cnt; state *parent, *go[MAXN];
5         state (int len = 0, int first = 0, int cnt = 0) :
            len (len), first (first), cnt (cnt), parent (0)
6         {
7             std::fill (go, go + MAXC, 0); } } node_pool[MAXN *
            2], *tot_node, *start, *null;
8     state *extend (state *tail, int token) {
9         state *p = tail; state *np = tail -> go[token] ?
            null : new (*tot_node++) state (tail -> len + 1,
10            tail -> len, 1);
11         while (p && !p -> go[token]) p -> go[token] = np, p
            = p -> parent;
12         if (!p) np -> parent = start;
13         else {
14             state *q = p -> go[token];
15             if (p -> len + 1 == q -> len) {
16                 np -> parent = q; } else {
17                 state *nq = new (*tot_node++) state (*q); nq -> cnt
                    = 0;
18                 nq -> len = p -> len + 1; np -> parent = q ->
                    parent = nq;
19                 while (p && p -> go[token] == q) {
20                     p -> go[token] = nq, p = p -> parent; } } }
21         return np == null ? np -> parent : np; }
22     void calc_cnt () {
23         static state *list[MAXN * 2]; state **end = list;
24         for (state *it = node_pool; it != tot_node; *(end++)
            = it++);
25         std::sort (list, end, [&] (state *a, state *b) {
26             return a -> len > b -> len; });
27         for (state **it = list, it != end; ++it) (*it) ->
            parent -> cnt += (*it) -> cnt; }
28     void init () {
29         tot_node = node_pool; start = new (*tot_node++) state
            (); null = new state (); }
30     suffix_automaton () { init (); } };

```

## 7 System

### 7.1 Builtin functions

1. `_builtin_clz`: Returns the number of leading 0-bits in  $x$ , starting at the most significant bit position. If  $x$  is 0, the result is undefined.
2. `_builtin_ctz`: Returns the number of trailing 0-bits in  $x$ , starting at the least significant bit position. If  $x$  is 0, the result is undefined.
3. `_builtin_clzsb`: Returns the number of leading redundant sign bits in  $x$ , i.e. the number of bits following the most significant bit that are identical to it. There are no special cases for 0 or other values.
4. `_builtin_popcount`: Returns the number of 1-bits in  $x$ .
5. `_builtin_parity`: Returns the parity of  $x$ , i.e. the number of 1-bits in  $x$  modulo 2.
6. `_builtin_bswap16`, `_builtin_bswap32`, `_builtin_bswap64`: Returns  $x$  with the order of the bytes (8 bits as a group) reversed.
7. `bitset::Find_first()`, `bitset::Find_next(idx)`: Finds 1 in a bitset.
8. `roundq`: Rounds `_float128`.

### 7.2 Container memory release

```

1 template <typename T> void clear (T &a) {
2     a.clear (); T (a).swap (a); }

```

### 7.3 Fast IO

```

1 #define __attribute__ ((optimize ("-O3")))
2 #define __inline__ __attribute__ ((__gnu_inline__,
    __always_inline__, __artificial__))
3 namespace io {
4     const int SIZE = 1000000; static char buf[SIZE + 1],
        *p = buf + SIZE;
5     template <class t> _bool read_int (t &x) {
6         register int f = 0, sgn = 0; x = 0;
7         while ((*p || (p = buf, buf[fread (buf, 1, SIZE,
            stdin)] = 0, buf[0])) &&
8             (isdigit (*p) && (x = x * 10 + (*p - '0'), f = 1)
                || !f && (*p != '-' || (sgn = 1)))) ++p;
9         if (sgn) x = -x;
10        return f; }
11    _int read_str (char *x, int len, char d = '\n') {
12        register int cnt = 0;
13        while ((*p || (p = buf, buf[fread (buf, 1, SIZE,
            stdin)] = 0, buf[0])) &&
14            cnt < len && *p != d) *(x++) = *(p++), ++cnt;
15        if (*p == d) ++p;
16        return cnt; }
17    //Set f to true to force an output (typically at the
        last write command).
18    const int WSIZE = 1000000; static char wbuf[2 * WSIZE
        ], *q = wbuf;
19    _void write (bool f, const char *str, ...) {
20        va_list args; va_start (args, str);
21        if ((q += vsprintf (q, str, args)) - wbuf >= WSIZE
            || f) fwrite (wbuf, 1, q - wbuf, stdout), q =
            wbuf;
22        va_end (args); } }

```

## 7.4 Formatting

Faster cin and cout.

```

1 std::ios::sync_with_stdio (0);
2 std::cin.tie (0); std::cout.tie (0);

```

Examples on IO functions.

```

1 std::string str;
2 std::getline (std::cin, str, '#');
3 char ch[100];
4 std::cin.getline (ch, 100, '#');
5 fgets (ch, 100, stdin);
6 int c = std::cin.peek ();
7 std::cin.ignore (100, '#');
8 std::cin.ignore (100, EOF);
9 std::cin.seekg (0, std::cin.end);
10 int length = std::cin.tellg ();
11 std::cin.seekg (0, std::cin.beg);
12 char *buf = new char[length];
13 std::cin.read (buf, length);
14 std::cout << std::setw (10);
15 std::cout << std::setfill ('#');
16 std::cout << std::left << x << "\n";
17 std::cout << std::internal << x << "\n";
18 std::cout << std::right << x << "\n";
19 std::cout << std::setprecision (10);
20 std::cout << std::fixed; // std::cout << std:::
    scientific;

```

## 7.5 Java

**Import** Libraries that are commonly used.

```

1 import java.io.*;
2 import java.lang.*;
3 import java.math.*;
4 import java.util.*;

```

**Input** Scanner is generally used to handle input.

```

1 Scanner in = new Scanner (System.in);

```

Or:

```

1 Scanner in = new Scanner (new BufferedInputStream (
    System.in));

```

Usage: `next + <typename> ()`, `hasNext + <typename> ()`.  
e.g. `in.nextInt ()`, `in.nextBigInteger ()`, `in.nextLine ()`, `in.hasNextInt ()`, etc.

**Output** Use `System.out` for output.

```

1 System.out.print (/*...*/);
2 System.out.println (/*...*/);
3 System.out.printf (/*...*/);

```

**BigInteger** To convert to a `BigInteger`, use `BigInteger.valueOf (int)` or `BigInteger (String, radix)`.

To convert from a `BigInteger`, use `.intValue ()`, `.longValue ()`, `.toString (radix)`.

Common unary operations include `.abs ()`, `.negate ()`, `.not ()`.

Common binary operations include `.max`, `.min`, `.add`, `.subtract`, `.multiply`, `.divide`, `.remainder`, `.gcd`, `.modInverse`, `.and`, `.or`, `.xor`, `.shiftLeft (int)`, `.shiftRight (int)`, `.pow (int)`, `.compareTo`.

Divide and remainder: `BigInteger[] .divideAndRemainder (BigInteger val)`.  
 Power module: `.modPow (BigInteger exponent, module)`.  
 Primality check: `.isProbablePrime (int certainty)`.  
 Square root:

```
1 public static BigInteger sqrt (BigInteger x) {
2   if (x.equals (BigInteger.ZERO) || x.equals (
3     BigInteger.ONE)) return x;
4   BigInteger d = BigInteger.ZERO.setBit (x.bitLength ()
5     / 2);
6   BigInteger d2 = d;
7   for (; ; ) {
8     BigInteger y = d.add (x.divide (d)).shiftRight (1);
9     if (y.equals (d) || y.equals (d2)) return d.min (d2);
10    d2 = d; d = y; } }
```

**BigDecimal** Literally a `BigInteger` and a scale.

When rounding, it is necessary to specify a `RoundingMode`, namely `RoundingMode.<mode>`, which includes:  
`CEILING, DOWN, FLOOR, HALF_DOWN, HALF_EVEN, HALF_UP, UNNECESSARY, UP`.

To convert to a `BigDecimal`, use `BigDecimal.valueOf (...)`, `BigDecimal (BigInteger, scale)` or `BigDecimal (String)`.

To divide: `.divide (BigDecimal, scale, roundingmode)`.

To set the scale: `.setScale (scale, roundingmode)`.

To remove trailing zeroes: `.stripTrailingZeros ()`.

**Array** Sort: `Arrays.sort (T[] a)`;

`Arrays.sort (T[] a, int fromIndex, int toIndex)`;

`Arrays.sort (T[] a, int fromIndex, int toIndex,`

`Comparator <? super T> comparator)`;

**PriorityQueue** An implementation of a min-heap.

Add element: `add (E)`.

Retrieve and pop element: `poll ()`.

Retrieve element: `peek ()`.

Size: `size ()`.

Clear: `clear ()`.

Comparator: `PriorityQueue <E> (int initcap, Comparator <? super E> comparator)`

**TreeMap** An implementation of a map. The entry is named `Map.Entry <K, V>`.

Retrieve key and value from an entry: `getKey, getValue ()`, `setValue (V)`.

Retrieve entry: `ceilingEntry, floorEntry, higherEntry, lowerEntry`.

Simplified operations: `clear (), put (K, V), get (K), remove (K), size ()`.

Comparator: `TreeMap <K, V> (Comparator <? super K> comparator)`.

**StringBuilder** Construction: `StringBuilder (String)`.

Insertion: `append (...)`, `insert (offset, ...)`. ... can be almost every type!

Fetch: `charAt (int)`.

Modification: `setCharAt (int, char)`, `delete (int, int)`, `reverse ()`.

Output: `length (), toString ()`.

**String** Formatting: `String.format (String, ...)`.

Case transform: `toLowerCase, toUpperCase`.

**Comparator** An example on a comparator.

```
1 public class Main {
2   public class Point {
3     public int x; public int y;
4     public Point () {
5       x = 0;
6       y = 0; }
7     public Point (int xx, int yy) {
8       x = xx;
9       y = yy; } }
10  public class Cmp implements Comparator <Point> {
11    public int compare (Point a, Point b) {
12      if (a.x < b.x) return -1;
13      if (a.x == b.x) {
14        if (a.y < b.y) return -1;
15        if (a.y == b.y) return 0; }
16      return 1; } }
17  public static void main (String [] args) {
18    Cmp c = new Cmp ();
19    TreeMap <Point, Point> t = new TreeMap <Point, Point>
20      > (c);
21    return; }
```

**Comparable** An example to implement `Comparable`.

```
1 public class Point implements Comparable <Point> {
2   public int x; public int y;
3   public Point () {
4     x = 0;
5     y = 0; }
6   public Point (int xx, int yy) {
7     x = xx;
8     y = yy; }
9   public int compareTo (Point p) {
10    if (x < p.x) return -1;
11    if (x == p.x) {
12      if (y < p.y) return -1;
13      if (y == p.y) return 0; }
14    return 1; }
15   public boolean equalTo (Point p) {
16     return (x == p.x && y == p.y); }
```

```
17 public int hashCode () {
18   return x + y; }
```

**Fast IO** A class for faster IO.

```
1 public class Main {
2   static class InputReader {
3     public BufferedReader reader;
4     public StringTokenizer tokenizer;
5     public InputReader (InputStream stream) {
6       reader = new BufferedReader (new InputStreamReader
7         (stream), 32768);
8       tokenizer = null; }
9     public String next() {
10      while (tokenizer == null || !tokenizer.
11        hasMoreTokens()) {
12        try {
13          String line = reader.readLine();
14          tokenizer = new StringTokenizer (line);
15        } catch (IOException e) {
16          throw new RuntimeException (e); } }
17      return tokenizer.nextToken(); }
18    public BigInteger nextBigInteger() {
19      return new BigInteger (next (), 10); /* radix */ }
20    public int nextInt() {
21      return Integer.parseInt (next()); }
22    public double nextDouble() {
23      return Double.parseDouble (next()); } }
24    public static void main (String[] args) {
25      InputReader in = new InputReader (System.in);
26    }
```

## 7.6 Random numbers

An example on the usage of generator and distribution.

```
1 std::mt19937_64 mt (time (0));
2 std::uniform_int_distribution <int> uid (1, 100);
3 std::uniform_real_distribution <double> urd (1, 100);
4 std::cout << uid (mt) << " " << urd (mt) << "\n";
```

## 7.7 Regular expression

This is an example to construct a pattern:

```
1 std::string str = ("The_the_there");
2 std::regex pattern ("(th|Th)[\\w]*", std:::
3   regex_constants::optimize | std:::regex_constants:::
4   ECMA5Script);
5 std::smatch match; //std::cmatch for char *
```

Use `std::regex_match` to find exact matches:

```
1 std::regex_match (str, match, pattern);
```

Use `std::sregex_iterator` to search for patterns:

```
1 auto mbegin = std::sregex_iterator (str.begin (), str.
2   end (), pattern);
3 auto mend = std::sregex_iterator ();
4 std::cout << "Found " << std::distance (mbegin, mend)
5   << " words:\n";
6 for (std::sregex_iterator i = mbegin; i != mend; ++i)
7 {
8   match = *i; /*...*/ }
```

The whole match is in `match[0]`, and backreferences are in `match[i]` up to `match.size ()`. `match.prefix ()` and `match.suffix ()` give the prefix and the suffix. `match.length ()` gives length and `match.position ()` gives the position of the match.

To replace a certain regular expression with another one, use `std::regex_replace`.

```
1 std::regex_replace (str, pattern, "sh");
```

where `$n` is the backreference, `$&` is the entire match, `$'` is the prefix, `$'` is the suffix, `$$` is the `$` sign.

## 7.8 Stack hack

The following lines allow the program to use larger stack memory.

```
1 //C++
2 #pragma comment (linker, "/STACK:36777216")
3 //G++
4 int __size__ = 256 << 20;
5 char * __p__ = (char*) malloc (__size__ + __size__);
6 __asm__ ("movl __0, %%esp\n" :: "r" (__p__));
```

## 7.9 Time hack

The following lines allow the program to check current time.

```
1 clock_t t = clock ();
2 std::cout << 1. * t / CLOCKS_PER_SEC << "\n";
```

## 8 Appendix

### 8.1 Table of formulae

#### Binomial coefficients

$$\binom{n}{k} = (-1)^k \binom{k-n-1}{k}, \quad \sum_{k \leq n} \binom{r+k}{k} = \binom{r+n+1}{n}$$

$$\sum_{k=0}^n \binom{k}{m} = \binom{n+1}{m+1}$$

$$\sqrt{1+z} = 1 + \sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{k \times 2^{2k-1}} \binom{2k-2}{k-1} z^k$$

$$\sum_{k=0}^r \binom{r-k}{m} \binom{s+k}{n} = \binom{r+s+1}{m+n+1}$$

$$C_{n,m} = \binom{n+m}{m} - \binom{n+m}{m-1}, n \geq m$$

$$\binom{n}{k} \equiv [n \& k = k] \pmod{2}$$

$$\binom{n_1 + \dots + n_p}{m} = \sum_{k_1 + \dots + k_p = m} \binom{n_1}{k_1} \dots \binom{n_p}{k_p}$$

#### Fibonacci numbers

$$F(z) = \frac{z}{1-z-z^2}$$

$$f_n = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}, \phi = \frac{1+\sqrt{5}}{2}, \hat{\phi} = \frac{1-\sqrt{5}}{2}$$

$$\sum_{k=1}^n f_k = f_{n+2} - 1, \quad \sum_{k=1}^n f_k^2 = f_n f_{n+1}$$

$$\sum_{k=0}^n f_k f_{n-k} = \frac{1}{5}(n-1)f_n + \frac{2}{5}n f_{n-1}$$

$$\frac{f_{2n}}{f_n} = f_{n-1} + f_{n+1}$$

$$f_1 + 2f_2 + 3f_3 + \dots + n f_n = n f_{n+2} - f_{n+3} + 2]$$

$$\gcd(f_m, f_n) = f_{\gcd(m,n)}$$

$$f_n^2 + (-1)^n = f_{n+1} f_{n-1}$$

$$f_{n+k} = f_n f_{k+1} + f_{n-1} f_k$$

$$f_{2n+1} = f_n^2 + f_{n+1}^2$$

$$(-1)^k f_{n-k} = f_n f_{k-1} - f_{n-1} f_k$$

$$\text{Modulo } f_n, f_{mn+r} \equiv \begin{cases} f_r, & m \bmod 4 = 0; \\ (-1)^{r+1} f_{n-r}, & m \bmod 4 = 1; \\ (-1)^n f_r, & m \bmod 4 = 2; \\ (-1)^{r+1+n} f_{n-r}, & m \bmod 4 = 3. \end{cases}$$

Period modulo a prime  $p$  is a factor of  $2p+2$  or  $p-1$ .

Only exception:  $G(5) = 20$ .

Period modulo the power of a prime  $p^k$ :  $G(p^k) = G(p)p^{k-1}$ .

Period modulo  $n = p_1^{k_1} \dots p_m^{k_m}$ :  $G(n) = \text{lcm}(G(p_1^{k_1}), \dots, G(p_m^{k_m}))$ .

#### Lucas numbers

$$L_0 = 2, L_1 = 1, L_n = L_{n-1} + L_{n-2} = \left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n$$

$$L(x) = \frac{2-x}{1-x-x^2}$$

#### Catlan numbers

$$c_1 = 1, c_n = \sum_{i=0}^{n-1} c_i c_{n-1-i} = c_{n-1} \frac{4n-2}{n+1} = \frac{\binom{2n}{n}}{n+1}$$

$$= \binom{2n}{n} - \binom{2n}{n-1}, c(x) = \frac{1-\sqrt{1-4x}}{2x}$$

#### Stirling cycle numbers

Divide  $n$  elements into  $k$  non-empty cycles.

$$s(n, 0) = 0, s(n, n) = 1, s(n+1, k) = s(n, k-1) - ns(n, k)$$

$$s(n, k) = (-1)^{n-k} \left[ \begin{matrix} n \\ k \end{matrix} \right]$$

$$\left[ \begin{matrix} n+1 \\ k \end{matrix} \right] = n \left[ \begin{matrix} n \\ k \end{matrix} \right] + \left[ \begin{matrix} n \\ k-1 \end{matrix} \right], \left[ \begin{matrix} n+1 \\ 2 \end{matrix} \right] = n! H_n$$

$$x^{\underline{n}} = x(x-1)\dots(x-n+1) = \sum_{k=0}^n \left[ \begin{matrix} n \\ k \end{matrix} \right] (-1)^{n-k} x^k$$

$$x^{\overline{n}} = x(x+1)\dots(x+n-1) = \sum_{k=0}^n \left[ \begin{matrix} n \\ k \end{matrix} \right] x^k$$

#### Stirling subset numbers

Divide  $n$  elements into  $k$  non-empty subsets.

$$\left\{ \begin{matrix} n+1 \\ k \end{matrix} \right\} = k \left\{ \begin{matrix} n \\ k \end{matrix} \right\} + \left\{ \begin{matrix} n \\ k-1 \end{matrix} \right\}$$

$$x^n = \sum_{k=0}^n \left\{ \begin{matrix} n \\ k \end{matrix} \right\} x^{\underline{k}} = \sum_{k=0}^n \left\{ \begin{matrix} n \\ k \end{matrix} \right\} (-1)^{n-k} x^{\overline{k}}$$

$$m! \left\{ \begin{matrix} n \\ m \end{matrix} \right\} = \sum_{k=0}^m \binom{m}{k} k^n (-1)^{m-k}$$

$$\sum_{k=1}^n k^p = \sum_{k=0}^p \left\{ \begin{matrix} p \\ k \end{matrix} \right\} (n+1)^{\underline{k}}$$

For a fixed  $k$ , generating functions :

$$\sum_{n=0}^{\infty} \left\{ \begin{matrix} n \\ k \end{matrix} \right\} x^{n-k} = \prod_{r=1}^k \frac{1}{1-rx}$$

#### Motzkin numbers

Draw non-intersecting chords between  $n$  points on a circle.

Pick  $n$  numbers  $k_1, k_2, \dots, k_n \in \{-1, 0, 1\}$  so that  $\sum_i^a k_i (1 \leq a \leq n)$  is non-negative and the sum of all numbers is 0.

$$M_{n+1} = M_n + \sum_i^{n-1} M_i M_{n-1-i} = \frac{(2n+3)M_n + 3nM_{n-1}}{n+3}$$

$$M_n = \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} \binom{n}{2k} \text{Catlan}(k)$$

$$M(X) = \frac{1-x-\sqrt{1-2x-3x^2}}{2x^2}$$

#### Eulerian numbers

Permutations of the numbers 1 to  $n$  in which exactly  $k$  elements are greater than the previous element.

$$\langle n \rangle_k = (k+1) \langle n-1 \rangle_k + (n-k) \langle n-1 \rangle_{k-1}$$

$$x^n = \sum_k \langle n \rangle_k \binom{x+k}{n}$$

$$\langle n \rangle_m = \sum_{k=0}^m \binom{n+1}{k} (m+1-k) n^{(k)} (-1)^k$$

#### Harmonic numbers

Sum of the reciprocals of the first  $n$  natural numbers.

$$\sum_{k=1}^n H_k = (n+1)H_n - n$$

$$\sum_{k=1}^n k H_k = \frac{n(n+1)}{2} H_n - \frac{n(n-1)}{4}$$

$$\sum_{k=1}^n \binom{k}{m} H_k = \binom{n+1}{m+1} (H_{n+1} - \frac{1}{m+1})$$

#### Pentagonal number theorem

$$\prod_{n=1}^{\infty} (1-x^n) = \sum_{n=-\infty}^{\infty} (-1)^k x^{k(3k-1)/2}$$

$$p(n) = p(n-1) + p(n-2) - p(n-5) - p(n-7) + \dots$$

$$f(n, k) = p(n) - p(n-k) - p(n-2k) + p(n-5k) + p(n-7k) - \dots$$

#### Bell numbers

Divide a set that has exactly  $n$  elements.

$$B_n = \sum_{k=1}^n \left\{ \begin{matrix} n \\ k \end{matrix} \right\}, \quad B_{n+1} = \sum_{k=0}^n \binom{n}{k} B_k$$

$$B_{p^m+n} \equiv m B_n + B_{n+1} \pmod{p}$$

$$B(x) = \sum_{n=0}^{\infty} \frac{B_n}{n!} x^n = e^{e^x - 1}$$

#### Bernoulli numbers

$$B_n = 1 - \sum_{k=0}^{n-1} \binom{n}{k} \frac{B_k}{n-k+1}$$

$$G(x) = \sum_{k=0}^{\infty} \frac{B_k}{k!} x^k = \frac{1}{\sum_{k=0}^{\infty} \frac{x^k}{(k+1)!}}$$

$$\sum_{k=1}^n k^m = \frac{1}{m+1} \sum_{k=0}^m \binom{m+1}{k} B_k n^{m-k+1}$$

#### Sum of powers

$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}, \quad \sum_{i=1}^n i^3 = \left(\frac{n(n+1)}{2}\right)^2$$

$$\sum_{i=1}^n i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

$$\sum_{i=1}^n i^5 = \frac{n^2(n+1)^2(2n^2+2n-1)}{12}$$

**Sum of squares** Denote  $r_k(n)$  the ways to form  $n$  with  $k$  squares.  
If :

$$n = 2^{a_0} p_1^{2a_1} \cdots p_r^{2a_r} q_1 b_1 \cdots q_s b_s$$

where  $p_i \equiv 3 \pmod{4}$ ,  $q_i \equiv 1 \pmod{4}$ , then

$$r_2(n) = \begin{cases} 0 & \text{if any } a_i \text{ is a half-integer} \\ 4 \prod_{i=1}^r (b_i + 1) & \text{if all } a_i \text{ are integers} \end{cases}$$

$r_3(n) > 0$  when and only when  $n$  is not  $4^a(8b+7)$ .

**Derangement**

$$D_1 = 0, D_2 = 1, D_n = n! \left( \frac{1}{0!} - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \cdots + \frac{(-1)^n}{n!} \right)$$

$$D_n = (n-1)(D_{n-1} + D_{n-2})$$

**Tetrahedron volume** If  $U, V, W, u, v, w$  are lengths of edges of the tetrahedron (first three form a triangle;  $u$  opposite to  $U$  and so on)

$$V = \frac{\sqrt{4u^2v^2w^2 - \sum_{cyc} u^2(v^2+w^2-U^2)^2 + \prod_{cyc} (v^2+w^2-U^2)}}{12}$$

## 8.2 Table of integrals

**Integral formulae**

$$\int_L f(x, y, z) ds = \int_{\alpha}^{\beta} f(x(t), y(t), z(t)) \sqrt{x'^2(t) + y'^2(t) + z'^2(t)} dt$$

$$\iint_{\Sigma} f(x, y, z) dS = \iint_D f(x(u, v), y(u, v), z(u, v)) \sqrt{EG - F^2} du dv,$$

where  $E = x_u^2 + y_u^2 + z_u^2, F = x_u x_v + y_u y_v + z_u z_v, G = x_v^2 + y_v^2 + z_v^2$ .

$$\int_L P(x, y, z) dx + Q(x, y, z) dy + R(x, y, z) dz$$

$$= \int_a^b [P(x(t), y(t), z(t))x'(t) + Q(x(t), y(t), z(t))y'(t) + R(x(t), y(t), z(t))z'(t)] dt$$

$$\iint_L P(x, y, z) dy dz + Q(x, y, z) dz dx + R(x, y, z) dx dy$$

$$= \pm \iint_D [P(x(u, v), y(u, v), z(u, v)) \frac{\partial(y, z)}{\partial(u, v)} +$$

$$Q(x(u, v), y(u, v), z(u, v)) \frac{\partial(z, x)}{\partial(u, v)} +$$

$$R(x(u, v), y(u, v), z(u, v)) \frac{\partial(x, y)}{\partial(u, v)}] du dv$$

**Variable substitution**

$$\iint_{T(D)} f(x, y) dx dy = \iint_D f(x(u, v), y(u, v)) \left| \frac{\partial(x, y)}{\partial(u, v)} \right| du dv$$

**Substitution with polar coordinates**

$$x = r \cos \theta, y = r \sin \theta$$

$$\left| \frac{\partial(x, y)}{\partial(r, \theta)} \right| = r$$

**Substitution with cylindrical coordinates**

$$x = r \cos \theta, y = r \sin \theta, z = z$$

$$\left| \frac{\partial(x, y, z)}{\partial(r, \theta, z)} \right| = r$$

**Substitution with spherical coordinates**

$$x = r \sin \varphi \cos \theta, y = r \sin \varphi \sin \theta, z = r \cos \varphi$$

$$\left| \frac{\partial(x, y, z)}{\partial(r, \varphi, \theta)} \right| = r^2 \sin \varphi$$

**Differentiation**

$$\begin{aligned} \left(\frac{u}{v}\right)' &= \frac{u'v - uv'}{v^2} & (\operatorname{arcsec} x)' &= \frac{1}{x\sqrt{1-x^2}} \\ (a^x)' &= (\ln a)a^x & (\tanh x)' &= \operatorname{sech}^2 x \\ (\tan x)' &= \sec^2 x & (\coth x)' &= -\operatorname{csch}^2 x \\ (\cot x)' &= \csc^2 x & (\operatorname{sech} x)' &= -\operatorname{sech} x \tanh x \\ (\sec x)' &= \tan x \sec x & (\operatorname{csch} x)' &= -\operatorname{csch} x \coth x \\ (\csc x)' &= -\cot x \csc x & (\operatorname{arcsinh} x)' &= \frac{1}{\sqrt{1+x^2}} \\ (\arcsin x)' &= \frac{1}{\sqrt{1-x^2}} & (\operatorname{arccosh} x)' &= \frac{1}{\sqrt{x^2-1}} \\ (\arccos x)' &= -\frac{1}{\sqrt{1-x^2}} & (\operatorname{arctanh} x)' &= \frac{1}{1-x^2} \\ (\arctan x)' &= \frac{1}{1+x^2} & (\operatorname{arccoth} x)' &= \frac{1}{x^2-1} \\ (\operatorname{arccot} x)' &= -\frac{1}{1+x^2} & (\operatorname{arccsch} x)' &= -\frac{1}{|x|\sqrt{1+x^2}} \\ (\operatorname{arccsc} x)' &= -\frac{1}{x\sqrt{1-x^2}} & (\operatorname{arcsech} x)' &= -\frac{1}{x\sqrt{1-x^2}} \end{aligned}$$

**Integration**

$ax + b$  ( $a \neq 0$ )

$$\begin{aligned} 1. \int \frac{x}{ax+b} dx &= \frac{1}{a^2} (ax+b - b \ln |ax+b|) + C \\ 2. \int \frac{x^2}{ax+b} dx &= \frac{1}{a^3} \left( \frac{1}{2} (ax+b)^2 - 2b(ax+b) + b^2 \ln |ax+b| \right) + C \\ 3. \int \frac{dx}{x(ax+b)} &= -\frac{1}{b} \ln \left| \frac{ax+b}{x} \right| + C \\ 4. \int \frac{dx}{x^2(ax+b)} &= -\frac{1}{bx} + \frac{a}{b^2} \ln \left| \frac{ax+b}{x} \right| + C \\ 5. \int \frac{x}{(ax+b)^2} dx &= \frac{1}{a^2} \left( \ln |ax+b| + \frac{b}{ax+b} \right) + C \\ 6. \int \frac{x^2}{(ax+b)^2} dx &= \frac{1}{a^3} \left( ax+b - 2b \ln |ax+b| - \frac{b^2}{ax+b} \right) + C \\ 7. \int \frac{dx}{x(ax+b)^2} &= \frac{1}{b(ax+b)} - \frac{1}{b^2} \ln \left| \frac{ax+b}{x} \right| + C \end{aligned}$$

$\sqrt{ax+b}$

$$\begin{aligned} 1. \int \sqrt{ax+b} dx &= \frac{2}{3a} \sqrt{(ax+b)^3} + C \\ 2. \int x \sqrt{ax+b} dx &= \frac{2}{15a^2} (3ax-2b) \sqrt{(ax+b)^3} + C \\ 3. \int x^2 \sqrt{ax+b} dx &= \frac{2}{105a^3} (15a^2x^2 - 12abx + 8b^2) \sqrt{(ax+b)^3} + C \\ 4. \int \frac{x}{\sqrt{ax+b}} dx &= \frac{2}{3a^2} (ax-2b) \sqrt{ax+b} + C \\ 5. \int \frac{x^2}{\sqrt{ax+b}} dx &= \frac{2}{15a^3} (3a^2x^2 - 4abx + 8b^2) \sqrt{ax+b} + C \\ 6. \int \frac{dx}{x \sqrt{ax+b}} &= \begin{cases} \frac{1}{\sqrt{b}} \ln \left| \frac{\sqrt{ax+b} - \sqrt{b}}{\sqrt{ax+b} + \sqrt{b}} \right| + C & (b > 0) \\ \frac{2}{\sqrt{-b}} \arctan \sqrt{\frac{ax+b}{-b}} + C & (b < 0) \end{cases} \\ 7. \int \frac{dx}{x^2 \sqrt{ax+b}} &= -\frac{\sqrt{ax+b}}{bx} - \frac{a}{2b} \int \frac{dx}{x \sqrt{ax+b}} \\ 8. \int \frac{\sqrt{ax+b}}{x} dx &= 2\sqrt{ax+b} + b \int \frac{dx}{x \sqrt{ax+b}} \\ 9. \int \frac{\sqrt{ax+b}}{x^2} dx &= -\frac{\sqrt{ax+b}}{x} + \frac{a}{2} \int \frac{dx}{x \sqrt{ax+b}} \end{aligned}$$

$x^2 \pm a^2$

$$\begin{aligned} 1. \int \frac{dx}{x^2+a^2} &= \frac{1}{a} \arctan \frac{x}{a} + C \\ 2. \int \frac{dx}{(x^2+a^2)^n} &= \frac{1}{2(n-1)a^2} \frac{x}{(x^2+a^2)^{n-1}} + \frac{2n-3}{2(n-1)a^2} \int \frac{dx}{(x^2+a^2)^{n-1}} \\ 3. \int \frac{dx}{x^2-a^2} &= \frac{1}{2a} \ln \left| \frac{x-a}{x+a} \right| + C \end{aligned}$$

$ax^2 + b$  ( $a > 0$ )

$$\begin{aligned} 1. \int \frac{dx}{ax^2+b} &= \begin{cases} \frac{1}{\sqrt{ab}} \arctan \sqrt{\frac{a}{b}} x + C & (b > 0) \\ \frac{1}{2\sqrt{-ab}} \ln \left| \frac{\sqrt{ax+b} - \sqrt{-b}}{\sqrt{ax+b} + \sqrt{-b}} \right| + C & (b < 0) \end{cases} \\ 2. \int \frac{x}{ax^2+b} dx &= \frac{1}{2a} \ln |ax^2+b| + C \\ 3. \int \frac{x^2}{ax^2+b} dx &= \frac{x}{a} - \frac{b}{a} \int \frac{dx}{ax^2+b} \\ 4. \int \frac{dx}{x(ax^2+b)} &= \frac{1}{2b} \ln \left| \frac{x^2}{ax^2+b} \right| + C \\ 5. \int \frac{dx}{x^2(ax^2+b)} &= -\frac{1}{bx} - \frac{a}{b} \int \frac{dx}{ax^2+b} \\ 6. \int \frac{dx}{x^3(ax^2+b)} &= \frac{a}{2b^2} \ln \left| \frac{ax^2+b}{x^2} \right| - \frac{1}{2bx^2} + C \\ 7. \int \frac{dx}{(ax^2+b)^2} &= \frac{x}{2b(ax^2+b)} + \frac{1}{2b} \int \frac{dx}{ax^2+b} \end{aligned}$$

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

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

$\sqrt{x^2+a^2}$  ( $a > 0$ )

$$\begin{aligned} 1. \int \frac{dx}{\sqrt{x^2+a^2}} &= \operatorname{arsh} \frac{x}{a} + C_1 = \ln(x + \sqrt{x^2+a^2}) + C \\ 2. \int \frac{dx}{\sqrt{(x^2+a^2)^3}} &= \frac{x}{a^2 \sqrt{x^2+a^2}} + C \\ 3. \int \frac{x}{\sqrt{x^2+a^2}} dx &= \sqrt{x^2+a^2} + C \\ 4. \int \frac{x}{\sqrt{(x^2+a^2)^3}} dx &= -\frac{1}{\sqrt{x^2+a^2}} + C \\ 5. \int \frac{x^2}{\sqrt{x^2+a^2}} dx &= \frac{x}{2} \sqrt{x^2+a^2} - \frac{a^2}{2} \ln(x + \sqrt{x^2+a^2}) + C \\ 6. \int \frac{x^2}{\sqrt{(x^2+a^2)^3}} dx &= -\frac{x}{\sqrt{x^2+a^2}} + \ln(x + \sqrt{x^2+a^2}) + C \\ 7. \int \frac{dx}{x \sqrt{x^2+a^2}} &= \frac{1}{a} \ln \left| \frac{\sqrt{x^2+a^2}-a}{|x|} \right| + C \\ 8. \int \frac{dx}{x^2 \sqrt{x^2+a^2}} &= -\frac{\sqrt{x^2+a^2}}{a^2 x} + C \\ 9. \int \sqrt{x^2+a^2} dx &= \frac{x}{2} \sqrt{x^2+a^2} + \frac{a^2}{2} \ln(x + \sqrt{x^2+a^2}) + C \\ 10. \int \sqrt{(x^2+a^2)^3} dx &= \frac{x}{8} (2x^2+5a^2) \sqrt{x^2+a^2} + \frac{3}{8} a^4 \ln(x + \sqrt{x^2+a^2}) + C \\ 11. \int x \sqrt{x^2+a^2} dx &= \frac{1}{3} \sqrt{(x^2+a^2)^3} + C \\ 12. \int x^2 \sqrt{x^2+a^2} dx &= \frac{x}{8} (2x^2+a^2) \sqrt{x^2+a^2} - \frac{a^4}{8} \ln(x + \sqrt{x^2+a^2}) + C \\ 13. \int \frac{\sqrt{x^2+a^2}}{x} dx &= \sqrt{x^2+a^2} + a \ln \left| \frac{\sqrt{x^2+a^2}-a}{|x|} \right| + C \end{aligned}$$



$$14. \int \frac{\sqrt{x^2+a^2}}{x^2} dx = -\frac{\sqrt{x^2+a^2}}{x} + \ln(x + \sqrt{x^2+a^2}) + C$$

$$\sqrt{x^2-a^2} \ (a > 0)$$

- $\int \frac{dx}{\sqrt{x^2-a^2}} = \frac{x}{|x|} \operatorname{arch} \frac{|x|}{a} + C_1 = \ln \left| x + \sqrt{x^2-a^2} \right| + C$
- $\int \frac{dx}{\sqrt{(x^2-a^2)^3}} = -\frac{x}{a^2 \sqrt{x^2-a^2}} + C$
- $\int \frac{x}{\sqrt{x^2-a^2}} dx = \sqrt{x^2-a^2} + C$
- $\int \frac{x}{\sqrt{(x^2-a^2)^3}} dx = -\frac{1}{\sqrt{x^2-a^2}} + C$
- $\int \frac{x^2}{\sqrt{x^2-a^2}} dx = \frac{x}{2} \sqrt{x^2-a^2} + \frac{a^2}{2} \ln |x + \sqrt{x^2-a^2}| + C$
- $\int \frac{x^2}{\sqrt{(x^2-a^2)^3}} dx = -\frac{x}{\sqrt{x^2-a^2}} + \ln |x + \sqrt{x^2-a^2}| + C$
- $\int \frac{dx}{x \sqrt{x^2-a^2}} = \frac{1}{a} \operatorname{arccos} \frac{a}{|x|} + C$
- $\int \frac{dx}{x^2 \sqrt{x^2-a^2}} = \frac{\sqrt{x^2-a^2}}{a^2 x} + C$
- $\int \sqrt{x^2-a^2} dx = \frac{x}{2} \sqrt{x^2-a^2} - \frac{a^2}{2} \ln |x + \sqrt{x^2-a^2}| + C$
- $\int \sqrt{(x^2-a^2)^3} dx = \frac{x}{8} (2x^2-5a^2) \sqrt{x^2-a^2} + \frac{3}{8} a^4 \ln |x + \sqrt{x^2-a^2}| + C$
- $\int x \sqrt{x^2-a^2} dx = \frac{1}{3} \sqrt{(x^2-a^2)^3} + C$
- $\int x^2 \sqrt{x^2-a^2} dx = \frac{x}{8} (2x^2-a^2) \sqrt{x^2-a^2} - \frac{a^4}{8} \ln |x + \sqrt{x^2-a^2}| + C$
- $\int \frac{\sqrt{x^2-a^2}}{x} dx = \sqrt{x^2-a^2} - a \operatorname{arccos} \frac{a}{|x|} + C$
- $\int \frac{\sqrt{x^2-a^2}}{x^2} dx = -\frac{\sqrt{x^2-a^2}}{x} + \ln |x + \sqrt{x^2-a^2}| + C$

$$\sqrt{a^2-x^2} \ (a > 0)$$

- $\int \frac{dx}{\sqrt{a^2-x^2}} = \arcsin \frac{x}{a} + C$
- $\frac{dx}{\sqrt{(a^2-x^2)^3}} = \frac{x}{a^2 \sqrt{a^2-x^2}} + C$
- $\int \frac{x}{\sqrt{a^2-x^2}} dx = -\sqrt{a^2-x^2} + C$
- $\int \frac{x}{\sqrt{(a^2-x^2)^3}} dx = \frac{1}{\sqrt{a^2-x^2}} + C$
- $\int \frac{x^2}{\sqrt{a^2-x^2}} dx = -\frac{x}{2} \sqrt{a^2-x^2} + \frac{a^2}{2} \arcsin \frac{x}{a} + C$
- $\int \frac{x^2}{\sqrt{(a^2-x^2)^3}} dx = \frac{x}{\sqrt{a^2-x^2}} - \arcsin \frac{x}{a} + C$
- $\int \frac{dx}{x \sqrt{a^2-x^2}} = \frac{1}{a} \ln \frac{a - \sqrt{a^2-x^2}}{|x|} + C$
- $\int \frac{dx}{x^2 \sqrt{a^2-x^2}} = -\frac{\sqrt{a^2-x^2}}{a^2 x} + C$
- $\int \sqrt{a^2-x^2} dx = \frac{x}{2} \sqrt{a^2-x^2} + \frac{a^2}{2} \arcsin \frac{x}{a} + C$
- $\int \sqrt{(a^2-x^2)^3} dx = \frac{x}{8} (5a^2-2x^2) \sqrt{a^2-x^2} + \frac{3}{8} a^4 \arcsin \frac{x}{a} + C$
- $\int x \sqrt{a^2-x^2} dx = -\frac{1}{3} \sqrt{(a^2-x^2)^3} + C$
- $\int x^2 \sqrt{a^2-x^2} dx = \frac{x}{8} (2x^2-a^2) \sqrt{a^2-x^2} + \frac{a^4}{8} \arcsin \frac{x}{a} + C$
- $\int \frac{\sqrt{a^2-x^2}}{x} dx = \sqrt{a^2-x^2} + a \ln \frac{a - \sqrt{a^2-x^2}}{|x|} + C$
- $\int \frac{\sqrt{a^2-x^2}}{x^2} dx = -\frac{\sqrt{a^2-x^2}}{x} - \arcsin \frac{x}{a} + C$

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

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

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

- $\int \sqrt{\frac{x-a}{x-b}} dx = (x-b) \sqrt{\frac{x-a}{x-b}} + (b-a) \ln(|\sqrt{x-a}| + |\sqrt{x-b}|) + C$
- $\int \sqrt{\frac{x-a}{b-x}} dx = (x-b) \sqrt{\frac{x-a}{b-x}} + (b-a) \arcsin \sqrt{\frac{x-a}{b-x}} + C$
- $\int \frac{dx}{\sqrt{(x-a)(b-x)}} = 2 \arcsin \sqrt{\frac{x-a}{b-x}} + C \ (a < b)$
- $\int \sqrt{(x-a)(b-x)} dx = \frac{2x-a-b}{4} \sqrt{(x-a)(b-x)} + \frac{(b-a)^2}{4} \arcsin \sqrt{\frac{x-a}{b-x}} + C \ (a < b)$

## Triangular function

- $\int \tan x dx = -\ln |\cos x| + C$
- $\int \cot x dx = \ln |\sin x| + C$
- $\int \sec x dx = \ln \left| \tan \left( \frac{x}{2} + \frac{\pi}{2} \right) \right| + C = \ln |\sec x + \tan x| + C$
- $\int \csc x dx = \ln \left| \tan \frac{x}{2} \right| + C = \ln |\csc x - \cot x| + C$
- $\int \sec^2 x dx = \tan x + C$
- $\int \csc^2 x dx = -\cot x + C$
- $\int \sec x \tan x dx = \sec x + C$
- $\int \csc x \cot x dx = -\csc x + C$
- $\int \sin^2 x dx = \frac{x}{2} - \frac{1}{4} \sin 2x + C$
- $\int \cos^2 x dx = \frac{x}{2} + \frac{1}{4} \sin 2x + C$
- $\int \sin^n x dx = -\frac{1}{n} \sin^{n-1} x \cos x + \frac{n-1}{n} \int \sin^{n-2} x dx$
- $\int \cos^n x dx = \frac{1}{n} \cos^{n-1} x \sin x + \frac{n-1}{n} \int \cos^{n-2} x dx$
- $\frac{dx}{\sin^n x} = -\frac{1}{n-1} \frac{\cos x}{\sin^{n-1} x} + \frac{n-2}{n-1} \int \frac{dx}{\sin^{n-2} x}$
- $\frac{dx}{\cos^n x} = \frac{1}{n-1} \frac{\sin x}{\cos^{n-1} x} + \frac{n-2}{n-1} \int \frac{dx}{\cos^{n-2} x}$

15.

$$\begin{aligned} & \int \cos^m x \sin^n x dx \\ &= \frac{1}{m+n} \cos^{m-1} x \sin^{n+1} x + \frac{m-1}{m+n} \int \cos^{m-2} x \sin^n x dx \\ &= -\frac{1}{m+n} \cos^{m+1} x \sin^{n-1} x + \frac{n-1}{m+1} \int \cos^m x \sin^{n-2} x dx \end{aligned}$$

- $\int \sin ax \cos bx dx = -\frac{1}{2(a+b)} \cos(a+b)x - \frac{1}{2(a-b)} \cos(a-b)x + C$
- $\int \sin ax \sin bx dx = -\frac{1}{2(a+b)} \sin(a+b)x + \frac{1}{2(a-b)} \sin(a-b)x + C$
- $\int \cos ax \cos bx dx = \frac{1}{2(a+b)} \sin(a+b)x + \frac{1}{2(a-b)} \sin(a-b)x + C$
- $\int \frac{dx}{a+b \sin x} = \begin{cases} \frac{2}{\sqrt{a^2-b^2}} \arctan \frac{a \tan \frac{x}{2} + b}{\sqrt{a^2-b^2}} + C & (a^2 > b^2) \\ \frac{1}{\sqrt{b^2-a^2}} \ln \left| \frac{a \tan \frac{x}{2} + b - \sqrt{b^2-a^2}}{a \tan \frac{x}{2} + b + \sqrt{b^2-a^2}} \right| + C & (a^2 < b^2) \end{cases}$
- $\int \frac{dx}{a+b \cos x} = \begin{cases} \frac{2}{a+b} \sqrt{\frac{a+b}{a-b}} \arctan \left( \sqrt{\frac{a-b}{a+b}} \tan \frac{x}{2} \right) + C & (a^2 > b^2) \\ \frac{1}{a+b} \sqrt{\frac{a+b}{a-b}} \ln \left| \frac{\tan \frac{x}{2} + \sqrt{\frac{a+b}{b-a}}}{\tan \frac{x}{2} - \sqrt{\frac{a+b}{b-a}}} \right| + C & (a^2 < b^2) \end{cases}$
- $\int \frac{dx}{a^2 \cos^2 x + b^2 \sin^2 x} = \frac{1}{ab} \arctan \left( \frac{b}{a} \tan x \right) + C$
- $\int \frac{dx}{a^2 \cos^2 x - b^2 \sin^2 x} = \frac{1}{2ab} \ln \left| \frac{b \tan x + a}{b \tan x - a} \right| + C$
- $\int x \sin ax dx = \frac{1}{a^2} \sin ax - \frac{1}{a} x \cos ax + C$
- $\int x^2 \sin ax dx = -\frac{1}{a} x^2 \cos ax + \frac{2}{a^2} x \sin ax + \frac{2}{a^3} \cos ax + C$
- $\int x \cos ax dx = \frac{1}{a^2} \cos ax + \frac{1}{a} x \sin ax + C$
- $\int x^2 \cos ax dx = \frac{1}{a} x^2 \sin ax + \frac{2}{a^2} x \cos ax - \frac{2}{a^3} \sin ax + C$

## Inverse triangular function ( $a > 0$ )

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

## Exponential function

- $\int a^x dx = \frac{1}{\ln a} a^x + C$
- $\int e^{ax} dx = \frac{1}{a} e^{ax} + C$
- $\int x e^{ax} dx = \frac{1}{a^2} (ax-1) e^{ax} + C$
- $\int x^n e^{ax} dx = \frac{1}{a} x^n e^{ax} - \frac{n}{a^2} \int x^{n-1} e^{ax} dx$
- $\int x a^x dx = \frac{x}{\ln a} a^x - \frac{1}{(\ln a)^2} a^x + C$
- $\int x^n a^x dx = \frac{1}{\ln a} x^n a^x - \frac{n}{\ln a} \int x^{n-1} a^x dx$
- $\int e^{ax} \sin bx dx = \frac{1}{a^2+b^2} e^{ax} (a \sin bx - b \cos bx) + C$
- $\int e^{ax} \cos bx dx = \frac{1}{a^2+b^2} e^{ax} (b \sin bx + a \cos bx) + C$
- $\int e^{ax} \sin^n bx dx = \frac{1}{a^2+b^2 n^2} e^{ax} \sin^{n-1} bx (a \sin bx - nb \cos bx) + \frac{n(n-1)b^2}{a^2+b^2 n^2} \int e^{ax} \sin^{n-2} bx dx$
- $\int e^{ax} \cos^n bx dx = \frac{1}{a^2+b^2 n^2} e^{ax} \cos^{n-1} bx (a \cos bx + nb \sin bx) + \frac{n(n-1)b^2}{a^2+b^2 n^2} \int e^{ax} \cos^{n-2} bx dx$

## Logarithmic function

- $\int \ln x dx = x \ln x - x + C$
- $\int \frac{dx}{x \ln x} = \ln |\ln x| + C$
- $\int x^n \ln x dx = \frac{1}{n+1} x^{n+1} (\ln x - \frac{1}{n+1}) + C$
- $\int (\ln x)^n dx = x (\ln x)^n - n \int (\ln x)^{n-1} dx$
- $\int x^m (\ln x)^n dx = \frac{1}{m+1} x^{m+1} (\ln x)^n - \frac{n}{m+1} \int x^m (\ln x)^{n-1} dx$

## 8.3 Table of range

Type	Width	Range
signed char	1	127
unsigned char	1	255
short	2	32 767
unsigned short	2	65 535
int	4	2 147 483 647
unsigned int	4	4 294 967 295
long long	8	9 223 372 036 854 775 807
unsigned long long	8	18 446 744 073 709 551 615
float	4	+/- 3.4e +/- 38 (7 digits)
double	8	+/- 1.7e +/- 308 (15 digits)
_float128	16	+/- 1.1e +/- 4932 (31 digits)

## 8.4 Table of regular expression

### Special pattern characters

Characters	Description
.	Not newline
\t	Tab (HT)
\n	Newline (LF)
\v	Vertical tab (VT)
\f	Form feed (FF)
\r	Carriage return (CR)
\cletter	Control code
\xhh	ASCII character
\uhhhh	Unicode character
\0	Null
\int	Backreference
\d	Digit
\D	Not digit
\s	Whitespace
\S	Not whitespace
\w	Word (letters, numbers and the underscore)
\W	Not word
\character	Character
[class]	Character class
[^class]	Negated character class

## Quantifiers

Characters	Times
*	0 or more
+	1 or more
?	0 or 1
{int}	int
{int,}	int or more
{min,max}	Between min and 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.

## Groups

Characters	Description
(subpattern)	Group with backreference
(?:subpattern)	Group without backreference

## Assertions

Characters	Description
^	Beginning of line
\$	End of line
\b	Word boundary
\B	Not a word boundary
(?=subpattern)	Positive lookahead
(?!subpattern)	Negative lookahead

**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.

## Character classes

Class	Description
[ :alnum:]	Alpha-numerical character
[ :alpha:]	Alphabetic character
[ :blank:]	Blank character
[ :cntrl:]	Control character
[ :digit:]	Decimal digit character
[ :graph:]	Character with graphical representation
[ :lower:]	Lowercase letter
[ :print:]	Printable character
[ :punct:]	Punctuation mark character
[ :space:]	Whitespace character
[ :upper:]	Uppercase letter
[ :xdigit:]	Hexadecimal digit character
[ :d:]	Decimal digit character
[ :w:]	Word character
[ :s:]	Whitespace character

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.

## 8.5 Table of operator precedence

Precedence	Operator	Associativity
1	::	Left-to-right
2	a++ a-- type() type{ } a() a[] . ->	
3	++a --a +a -a ! (type) *a &a sizeof new new[] delete delete[]	
4	.* ->*	
5	a*b a/b a%b	Left-to-right
6	a+b a-b	
7	<< >>	
8	< <= > >=	
9	== !=	
10	a&b	
11	a^b	
12	a b	
13	&&	
14		
15	a?b:c throw = += -= *= /= %= <<= >>= &= ^=  =	Right-to-left
16	,	Left-to-right