Luna's Magic Reference

Suzune Nisiyama

October 7, 2018

Contents

1 Environment

1.1 Vimrc

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

2 Data Structure

2.1 Balanced tree

2.1.1 Link-cut tree

```cpp
void access (int x) {
 int u = x, v = -1;
 while (u != -1) {
  splay (u); push_down (u);
  if (~n[u].c[1]) n[n[u].c[1]].f = -1, n[n[u].c[1]].p = u;
  n[u].c[1] = v;
  if (~v) n[v].f = u, n[v].p = -1;
  update (u); u = n[v = u].p; }
 splay (x); }
```

2.1.2 Splay operation

```cpp
void rotate (int x, int k) {
 int y = a[x].fa;
 if (~a[y].fa) a[a[y].fa].son[a[a[y].fa].son[1] == y] = x;
 a[x].fa = a[y].fa; a[y].son[k] = a[x].son[k ^ 1];
 if (~a[x].son[k ^ 1]) a[a[x].son[k ^ 1]].fa = y;
 a[y].fa = x; a[x].son[k ^ 1] = y;
 update (y); update (x); }
void splay (int x, int s = -1) {
 push_down (x);
 while (a[x].fa != s) {
  int y = a[x].fa, z = a[y].fa;
  if (z == s) {
   push_down (z = y); push_down (x);
   rotate (x, a[y].son[1] == x);
  } else {
   push_down (z); push_down (y); push_down (x);
   int t1 = a[y].son[1] == x, t2 = a[z].son[1] == y;
   if (t1 == t2) rotate (y, t2), rotate (x, t1);
   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.

```cpp
template <int MAXN = 200000, int MAXK = 2>
struct kd_tree {
 int k, size;
 struct point { int data[MAXK], id; } p[MAXN];
 struct kd_node {
  int l, r; point p, dmin, dmax;
  kd_node() {}
  kd_node (const point &rhs) : l (-1), r (-1), p (rhs), dmin (rhs), dmax (rhs) {}
  void merge (const kd_node &rhs, int k) {
   for (register int i = 0; i < k; ++i) {
    dmin.data[i] = std::min (dmin.data[i], rhs.dmin.data[i]);
    dmax.data[i] = std::max (dmax.data[i], rhs.dmax.data[i]); } }
  long long min_dist (const point &rhs, int k) const {
   register long long ret = 0;
   for (register int i = 0; i < k; ++i) {
    if (dmin.data[i] <= rhs.data[i] && rhs.data[i] <= dmax.data[i]) continue;
    ret += std::min (1ll * (dmin.data[i] - rhs.data[i]) * (dmin.data[i] - rhs.data[i]),
     1ll * (dmax.data[i] - rhs.data[i]) * (dmax.data[i] - rhs.data[i]));
//    ret += std::max (0, rhs.data[i] - dmax.data[i]) + std::max (0, dmin.data[i] - rhs.data[i]);
   } return ret; }
  long long max_dist (const point &rhs, int k) {
   long long ret = 0;
   for (int i = 0; i < k; ++i) {
    int tmp = std::max (std::abs (dmin.data[i] - rhs.data[i]), std::abs (dmax.data[i] - rhs.data[i]));
    ret += 1ll * tmp * tmp; }
//    ret += std::max (std::abs (rhs.data[i] - dmax.data[i]) + std::abs (rhs.data[i] - dmin.data[i]));
   }
  return ret; } } tree[MAXN * 4];
 struct result {
  long long dist; point d; result() {}
  result (const long long &dist, const point &d) : dist (dist), d (d) {}
  bool operator > (const result &rhs) const { return dist > rhs.dist || (dist == rhs.dist && d.id > rhs.d.id); }
  bool operator < (const result &rhs) const { return dist < rhs.dist || (dist == rhs.dist && d.id < rhs.d.id); } };
 long long sqrdist (const point &a, const point &b) {
  long long ret = 0;
  for (int i = 0; i < k; ++i) ret += 1ll * (a.data[i] - b.data[i]) * (a.data[i] - b.data[i]);
//  for (int i = 0; i < k; ++i) ret += std::abs (a.data[i] - b.data[i]);
  return ret; }
 int alloc() { tree[size].l = tree[size].r = -1; return size++; }
 void build (const int &depth, int &rt, const int &l, const int &r) {
  if (l > r) return;
  register int middle = (l + r) >> 1;
  std::nth_element (p + l, p + middle, p + r + 1, [=] (const point & a, const point & b) { return a.data[depth] < b.data[depth]; });
  tree[rt = alloc()] = kd_node (p[middle]);
  if (l == r) return;
  build ((depth + 1) % k, tree[rt].l, l, middle - 1);
  build ((depth + 1) % k, tree[rt].r, middle + 1, r);
  if (~tree[rt].l) tree[rt].merge (tree[tree[rt].l], k);
  if (~tree[rt].r) tree[rt].merge (tree[tree[rt].r], k); }
 std::priority_queue<result, std::vector<result>, std::less <result>> heap_l;
 std::priority_queue<result, std::vector<result>, std::greater <result>> heap_r;
 void _min_kth (const int &depth, const int &rt, const int &m, const point &d) {
  result tmp = result (sqrdist (tree[rt].p, d), tree[rt].p);
  if ((int)heap_l.size() < m) heap_l.push (tmp);
  else if (tmp < heap_l.top()) {
   heap_l.pop();
   heap_l.push (tmp); }
  int x = tree[rt].l, y = tree[rt].r;
  if (~x && ~y && sqrdist (d, tree[x].p) > sqrdist (d, tree[y].p)) std::swap (x, y);
  if (~x && ((int)heap_l.size() < m || tree[x].min_dist (d, k) < heap_l.top().dist))
   _min_kth ((depth + 1) % k, x, m, d);
  if (~y && ((int)heap_l.size() < m || tree[y].min_dist (d, k) < heap_l.top().dist))
   _min_kth ((depth + 1) % k, y, m, d); }
 void _max_kth (const int &depth, const int &rt, const int &m, const point &d) {
  result tmp = result (sqrdist (tree[rt].p, d), tree[rt].p);
   if ((int)heap_r.size() < m) heap_r.push (tmp);
   else if (tmp > heap_r.top()) {
    heap_r.pop();
    heap_r.push (tmp); }
   int x = tree[rt].l, y = tree[rt].r;
   if (~x && ~y && sqrdist (d, tree[x].p) < sqrdist (d, tree[y].p)) std::swap (x, y);
   if (~x && ((int)heap_r.size() < m || tree[x].max_dist (d, k) >= heap_r.top().dist))
    _max_kth ((depth + 1) % k, x, m, d);
   if (~y && ((int)heap_r.size() < m || tree[y].max_dist (d, k) >= heap_r.top().dist))
    _max_kth ((depth + 1) % k, y, m, d); }
 void init (int n, int k) { this -> k = k; size = 0; int rt = 0; build (0, rt, 0, n - 1); }
 result min_kth (const point &d, const int &m) { heap_l = decltype (heap_l) (); _min_kth (0, 0, m, d); return heap_l.top (); }
 result max_kth (const point &d, const int &m) { heap_r = decltype (heap_r) (); _max_kth (0, 0, m, d); return heap_r.top (); } };
```

3 Geometry

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

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

3.1 3D geometry

3.1.1 3D point

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

```cpp
#define cp3 const point3 &
struct point3 {
 double x, y, z;
```

```
 explicit point3 (cd x = 0, cd y = 0, cd z = 0) : x (x), y (y), z (z) {} };
point3 operator + (cp3 a, cp3 b) { return point3 (a.x + b.x, a.y + b.y, a.z + b.z); }
point3 operator - (cp3 a, cp3 b) { return point3 (a.x - b.x, a.y - b.y, a.z - b.z); }
point3 operator * (cp3 a, cd b) { return point3 (a.x * b, a.y * b, a.z * b); }
point3 operator / (cp3 a, cd b) { return point3 (a.x / b, a.y / b, a.z / b); }
double dot (cp3 a, cp3 b) { return a.x * b.x + a.y * b.y + a.z * b.z; }
point3 det (cp3 a, cp3 b) { return point3 (a.y * b.z - a.z * b.y, -a.x * b.z + a.z * b.x, a.x * b.y - a.y * b.x); }
double dis2 (cp3 a, cp3 b = point3 ()) { return sqr (a.x - b.x) + sqr (a.y - b.y) + sqr (a.z - b.z); }
double dis (cp3 a, cp3 b = point3 ()) { return msqrt (dis2 (a, b)); }
point3 rotate(cp3 p, cp3 axis, double w) {
 double x = axis.x, y = axis.y, z = axis.z;
 double s = x * x + y * y + z * z, ss = msqrt(s), cosw = cos(w), sinw = sin(w);
 double a[4][4]; memset(a, 0, sizeof (a));
 a[3][3] = 1;
 a[0][0] = ((y * y + z * z) * cosw + x * x) / s;
 a[0][1] = x * y * (1 - cosw) / s + z * sinw / ss;
 a[0][2] = x * z * (1 - cosw) / s - y * sinw / ss;
 a[1][0] = x * y * (1 - cosw) / s - z * sinw / ss;
 a[1][1] = ((x * x + z * z) * cosw + y * y) / s;
 a[1][2] = y * z * (1 - cosw) / s + x * sinw / ss;
 a[2][0] = x * z * (1 - cosw) / s + y * sinw / ss;
 a[2][1] = y * z * (1 - cosw) / s - x * sinw / ss;
 a[2][2] = ((x * x + y * y) * cos(w) + z * z) / s;
 double ans[4] = {0, 0, 0, 0}, c[4] = {p.x, p.y, p.z, 1};
 for (int i = 0; i < 4; ++i) for (int j = 0; j < 4; ++j)
  ans[i] += a[j][i] * c[j];
 return point3 (ans[0], ans[1], ans[2]);
}
```

3.1.2 3D line

```
#define cl3 const line3 &
struct line3 {
 point3 s, t;
 explicit line3 (cp3 s = point3 (), cp3 t = point3 ()) : s (s), t (t) {} };
point3 line_plane_intersection (cl3 a, cl3 b) { return a.s + (a.t - a.s) * dot (b.s - a.s, b.t - b.s) / dot (a.t - a.s, b.t - b.s); }
line3 plane_intersection (cl3 a, cl3 b) {
 point3 p = det (a.t - a.s, b.t - b.s), q = det (a.t - a.s, p), s = line_plane_intersection (line3 (a.s, a.s + q), b);
 return line3 (s, s + p); }
point3 project_to_plane (cp3 a, cl3 b) { return a + (b.t - b.s) * dot (b.t - b.s, b.s - a) / dis2 (b.t - b.s); }
```

3.1.3 3D convex hull

Input n and p. Return *face*.
Note: Remove coincide points first.

```
template <int MAXN = 500>
struct convex_hull3 {
 double mix (cp3 a, cp3 b, cp3 c) { return dot (det (a, b), c); }
 double volume (cp3 a, cp3 b, cp3 c, cp3 d) { return mix (b - a, c - a, d - a); }
 struct tri {
  int a, b, c; tri() {}
  tri(int _a, int _b, int _c): a(_a), b(_b), c(_c) {}
  double area() const { return dis (det (p[b] - p[a], p[c] - p[a])) / 2; }
  point3 normal() const { return det (p[b] - p[a], p[c] - p[a]).unit (); }
  double dis (cp3 p0) const { return dot (normal (), p0 - p[a]); } };
 int n; std::vector <point3> p;
 std::vector <tri> face, tmp;
 int mark[MAXN][MAXN], time;
 void add (int v) {
  ++time; tmp.clear ();
  for (int i = 0; i < (int) face.size (); ++i) {
   int a = face[i].a, b = face[i].b, c = face[i].c;
   if (sgn (volume (p[v], p[a], p[b], p[c])) > 0)
    mark[a][b] = mark[b][a] = mark[a][c] = mark[c][a] = mark[b][c] = mark[c][b] = time;
   else tmp.push_back (face[i]); }
  face.clear (); face = tmp;
  for (int i = 0; i < (int) tmp.size (); ++i) {
   int a = face[i].a, b = face[i].b, c = face[i].c;
   if (mark[a][b] == time) face.emplace_back (v, b, a);
   if (mark[b][c] == time) face.emplace_back (v, c, b);
   if (mark[c][a] == time) face.emplace_back (v, a, c); } }
 void reorder () {
  for (int i = 2; i < n; ++i) {
   point3 tmp = det (p[i] - p[0], p[i] - p[1]);
   if (sgn (dis (tmp))) {
    std::swap (p[i], p[2]);
    for (int j = 3; j < n; ++ j)
     if (sgn (volume (p[0], p[1], p[2], p[j]))) {
      std::swap (p[j], p[3]); return; } } } }
 void build_convex () {
  reorder (); face.clear ();
  face.emplace_back (0, 1, 2);
  face.emplace_back (0, 2, 1);
  for (int i = 3; i < n; ++i) add(i); } };
```

3.2 Circle

1. `line_circle_intersect`: In order of the direction of a.
2. `circle_intersect`: Counter-clockwise with respect of O_a.
3. `tangent`: Counter-clockwise with respect of a.
4. `extangent`: Counter-clockwise with respect of O_a.
5. `intangent`: Counter-clockwise with respect of O_a.

```
#define cc const circle &
struct circle {
 point c; double r;
 explicit circle (point c = point (), double r = 0) : c (c), r (r) {} };
bool operator == (cc a, cc b) { return a.c == b.c && cmp (a.r, b.r) == 0; }
bool operator != (cc a, cc b) { return !(a == b); }
bool in_circle (cp a, cc b) { return cmp (dis (a, b.c), b.r) <= 0; }
circle make_circle (cp a, cp b) { return circle ((a + b) / 2, dis (a, b) / 2); }
circle make_circle (cp a, cp b, cp c) { point p = circumcenter (a, b, c); return circle (p, dis (p, a)); }
std::vector <point> line_circle_intersect (cl a, cc b) {
 if (cmp (point_to_line (b.c, a), b.r) > 0) return std::vector <point> ();
 double x = msqrt (sqr (b.r) - sqr (point_to_line (b.c, a)));
 point s = project_to_line (b.c, a), u = (a.t - a.s).unit ();
 if (sgn (x) == 0) return std::vector <point> ({s});
 return std::vector <point> ({s - u * x, s + u * x}); }
double circle_intersect_area (cc a, cc b) {
 double d = dis (a.c, b.c);
 if (sgn (d - (a.r + b.r)) >= 0) return 0;
 if (sgn (d - abs(a.r - b.r)) <= 0) {
  double r = std::min (a.r, b.r); return r * r * PI; }
 double x = (d * d + a.r * a.r - b.r * b.r) / (2 * d), t1 = acos (min (1., max (-1., x / a.r))), t2 = acos (min (1., max (-1., (d - x) / b.r)));
 return a.r * a.r * t1 + b.r * b.r * t2 - d * a.r * sin (t1); }
std::vector <point> circle_intersect (cc a, cc b) {
 if (a.c == b.c || cmp (dis (a.c, b.c), a.r + b.r) > 0 || cmp (dis (a.c, b.c), std::abs (a.r - b.r)) < 0) return std::vector <point> ();
 point r = (b.c - a.c).unit (); double d = dis (a.c, b.c);
 double x = ((sqr (a.r) - sqr (b.r)) / d + d) / 2, h = msqrt (sqr (a.r) - sqr (x));
 if (sgn (h) == 0) return std::vector <point> ({a.c + r * x});
 return std::vector <point> ({a.c + r * x - r.rot90 () * h, a.c + r * x + r.rot90 () * h}); }
std::vector <point> tangent (cp a, cc b) { circle p = make_circle (a, b.c); return circle_intersect (p, b); }
std::vector <line> extangent (cc a, cc b) {
 std::vector <line> ret;
 if (cmp (dis (a.c, b.c), std::abs (a.r - b.r)) <= 0) return ret;
 if (sgn (a.r - b.r) == 0) {
  point dir = b.c - a.c; dir = (dir * a.r / dis (dir)).rot90 ();
  ret.push_back (line (a.c - dir, b.c - dir));
  ret.push_back (line (a.c + dir, b.c + dir));
 } else {
  point p = (b.c * a.r - a.c * b.r) / (a.r - b.r);
  std::vector <point> pp = tangent (p, a), qq = tangent (p, b);
  if (pp.size () == 2 && qq.size () == 2) {
   if (cmp (a.r, b.r) < 0) std::swap (pp[0], pp[1]), std::swap (qq[0], qq[1]);
   ret.push_back (line (pp[0], qq[0]));
   ret.push_back (line (pp[1], qq[1])); } }
 return ret; }
std::vector <line> intangent (cc a, cc b) {
 std::vector <line> ret;
 point p = (b.c * a.r + a.c * b.r) / (a.r + b.r);
 std::vector <point> pp = tangent (p, a), qq = tangent (p, b);
 if (pp.size () == 2 && qq.size () == 2) {
  ret.push_back (line (pp[0], qq[0]));
  ret.push_back (line (pp[1], qq[1])); }
 return ret; }
```

3.2.1 Intersection of a polygon and a circle

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

```
  double c = (2.0 * r * r - dis2 (a, b)) / (2.0 * r * r);
  return r * r * acos (c) / 2.0; }
 double area (cp a, cp b, const double &r) {
  double dA = dot (a, a), dB = dot (b, b), dC = point_to_segment (point (), line (a, b));
  if (sgn (dA - r * r) <= 0 && sgn (dB - r * r) <= 0) return det (a, b) / 2.0;
  point tA = a.unit () * r, tB = b.unit () * r;
  if (sgn (dC - r) > 0) return sector_area (tA, tB, r);
  std::vector <point> ret = line_circle_intersect (line (a, b), circle (point (), r));
  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);
  if (sgn (dA - r * r) > 0) return det (ret[0], b) / 2.0 + sector_area (tA, ret[0], r);
  else return det (a, ret[1]) / 2.0 + sector_area (ret[1], tB, r); }
 double solve (const std::vector <point> &p, cc c) {
  double ret = 0.0;
  for (int i = 0; i < (int) p.size (); ++i) {
   int s = sgn (det (p[i] - c.c, p[(i + 1) % p.size ()] - c.c));
   if (s > 0) ret += area (p[i] - c.c, p[(i + 1) % p.size ()] - c.c, c.r);
   else ret -= area (p[(i + 1) % p.size ()] - c.c, p[i] - c.c, c.r); }
  return std::abs (ret); } };
```

3.2.2 Minimum circle

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

3.2.3 Union of circles

```
template <int MAXN = 500> struct union_circle {
 int C; circle c[MAXN]; double area[MAXN];
 struct event {
  point p; double ang; int delta;
  event (cp p = point (), double ang = 0, int delta = 0) : p(p), ang(ang), delta(delta) {}
  bool operator < (const event &a) { return ang < a.ang; }
 };
 void addevent(cc a, cc b, std::vector <event> &evt, int &cnt) {
  double d2 = dis2 (a.c, b.c), d_ratio = ((a.r - b.r) * (a.r + b.r) / d2 + 1) / 2,
   p_ratio = msqrt (std::max (0., -(d2 - sqr(a.r - b.r)) * (d2 - sqr(a.r + b.r)) / (d2 * d2 * 4)));
  point d = b.c - a.c, p = d.rot(PI / 2), q0 = a.c + d * d_ratio + p * p_ratio, q1 = a.c + d * d_ratio - p * p_ratio;
  double ang0 = atan2 ((q0 - a.c).y, (q0 - a.c).x), ang1 = atan2 ((q1 - a.c).x, (q1 - a.c).y);
  evt.emplace_back(q1, ang1, 1); evt.emplace_back(q0, ang0, -1); cnt += ang1 > ang0; }
 bool same(cc a, cc b) { return sgn (dis (a.c, b.c)) == 0 && sgn (a.r - b.r) == 0; }
 bool overlap(cc a, cc b) {  return sgn (a.r - b.r - dis (a.c, b.c)) >= 0;  }
 bool intersect(cc a, cc b) { return sgn (dis (a.c, b.c) - a.r - b.r) < 0; }
 void solve() {
  std::fill (area, area + C + 2, 0);
  for (int i = 0; i < C; ++i) {
   int cnt = 1; std::vector <event> evt;
   for (int j = 0; j < i; ++j) if (same (c[i], c[j])) ++cnt;
   for (int j = 0; j < C; ++j) if (j != i && !same (c[i], c[j]) && overlap (c[j], c[i])) ++cnt;
   for (int j = 0; j < C; ++j) if (j != i && !overlap (c[j], c[i]) && !overlap (c[i], c[j]) && intersect (c[i], c[j]))
    addevent (c[i], c[j], evt, cnt);
   if (evt.empty ()) area[cnt] += PI * c[i].r * c[i].r;
   else {
    std::sort (evt.begin (), evt.end ());
    evt.push_back (evt.front ());
    for (int j = 0; j + 1 < (int) evt.size (); ++j) {
     cnt += evt[j].delta; area[cnt] += det (evt[j].p, evt[j + 1].p) / 2;
     double ang = evt[j + 1].ang - evt[j].ang; if (ang < 0) ang += PI * 2;
     area[cnt] += ang * c[i].r * c[i].r / 2 - sin(ang) * c[i].r * c[i].r / 2; } } } } };
```

3.3 Line

```
#define cl const line &
struct line {
 point s, t;
 explicit line (cp s = point (), cp t = point ()) : s (s), t (t) {} };
bool point_on_segment (cp a, cl b) { return sgn (det (a - b.s, b.t - b.s)) == 0 && sgn (dot (b.s - a, b.t - a)) <= 0; }
bool two_side (cp a, cp b, cl c) { return sgn (det (a - c.s, c.t - c.s)) * sgn (det (b - c.s, c.t - c.s)) < 0; }
bool intersect_judgment (cl a, cl b) {
 if (point_on_segment (b.s, a) || point_on_segment (b.t, a)) return true;
 if (point_on_segment (a.s, b) || point_on_segment (a.t, b)) return true;
 return two_side (a.s, a.t, b) && two_side (b.s, b.t, a); }
point line_intersect (cl a, cl b) {
 double s1 = det (a.t - a.s, b.s - a.s), s2 = det (a.t - a.s, b.t - a.s);
 return (b.s * s2 - b.t * s1) / (s2 - s1); }
double point_to_line (cp a, cl b) { return std::abs (det (b.t - b.s, a - b.s)) / dis (b.s, b.t); }
point project_to_line (cp a, cl b) { return b.s + (b.t - b.s) * (dot (a - b.s, b.t - b.s) / dis2 (b.t, b.s)); }
double point_to_segment (cp a, cl b) {
 if (sgn (dot (b.s - a, b.t - b.s) * dot (b.t - a, b.t - b.s)) <= 0) return std::abs (det (b.t - b.s, a - b.s)) / dis (b.s, b.t);
 return std::min (dis (a, b.s), dis (a, b.t)); }
bool in_polygon (cp p, const std::vector <point> & po) {
 int n = (int) po.size (), counter = 0;
 for (int i = 0; i < n; ++i) {
  point a = po[i], b = po[(i + 1) % n];
  //Modify the next line if necessary.
  if (point_on_segment (p, line (a, b))) return true;
  int x = sgn (det (p - a, b - a)), y = sgn (a.y - p.y), z = sgn (b.y - p.y);
  if (x > 0 && y <= 0 && z > 0) counter++;
  if (x < 0 && z <= 0 && y > 0) counter--; }
 return counter != 0; }
double polygon_area (const std::vector <point> &a) {
 double ans = 0.0;
 for (int i = 0; i < (int) a.size (); ++i) ans += det (a[i], a[ (i + 1) % a.size ()]) / 2.0;
 return ans; }
```

3.3.1 Half plane intersection

1. `cut`: Online in $O(n^2)$.
2. `half_plane_intersect`: Offline in $O(mlogm)$.

```
std::vector <point> cut (const std::vector<point> &c, line p) {
 std::vector <point> ret;
 if (c.empty ()) return ret;
 for (int i = 0; i < (int) c.size (); ++i) {
  int j = (i + 1) % (int) c.size ();
  if (turn_left (p.s, p.t, c[i])) ret.push_back (c[i]);
  if (two_side (c[i], c[j], p)) ret.push_back (line_intersect (p, line (c[i], c[j]))); }
 return ret; }
bool turn_left (cl l, cp p) { return sgn (det (l.t - l.s, p - l.s)) >= 0; }
int cmp (cp a, cp b) { return a.dim () != b.dim () ? (a.dim () < b.dim () ? -1 : 1) : -sgn (det (a, b)); }
std::vector <point> half_plane_intersect (std::vector <line> h) {
 typedef std::pair <point, line> polar;
 std::vector <polar> g; g.resize (h.size ());
 for (int i = 0; i < (int) h.size (); ++i) g[i] = std::make_pair (h[i].t - h[i].s, h[i]);
 sort (g.begin (), g.end (), [&] (const polar &a, const polar &b) {
  if (cmp (a.first, b.first) == 0) return sgn (det (a.second.t - a.second.s, b.second.t - a.second.s)) < 0;
  else return cmp (a.first, b.first) < 0; });
 h.resize (std::unique (g.begin (), g.end (), [] (const polar &a, const polar &b) { return cmp (a.first, b.first) == 0; }) - g.begin ());
 for (int i = 0; i < (int) h.size (); ++i) h[i] = g[i].second;
 int fore = 0, rear = -1; std::vector <line> ret (h.size (), line ());
 for (int i = 0; i < (int) h.size (); ++i) {
  while (fore < rear && !turn_left (h[i], line_intersect (ret[rear - 1], ret[rear]))) --rear;
  while (fore < rear && !turn_left (h[i], line_intersect (ret[fore], ret[fore + 1]))) ++fore;
  ret[++rear] = h[i]; }
 while (rear - fore > 1 && !turn_left (ret[fore], line_intersect (ret[rear - 1], ret[rear]))) --rear;
```

```
 while (rear - fore > 1 && !turn_left (ret[rear], line_intersect (ret[fore], ret[fore + 1]))) ++fore;
 if (rear - fore < 2) return std::vector <point> ();
 std::vector <point> ans; ans.resize (rear + 1);
 for (int i = 0; i < rear + 1; ++i) ans[i] = line_intersect (ret[i], ret[(i + 1) % (rear + 1)]);
 return ans; }
```

3.4 Point

`rot90`: Counter-clockwise rotation.

```
#define cp const point &
struct point {
 double x, y;
 explicit point (cd x = 0, cd y = 0) : x (x), y (y) {}
 int dim () const { return sgn (y) == 0 ? sgn (x) > 0 : sgn (y) > 0; }
 point unit () const { double l = msqrt (x * x + y * y); return point (x / l, y / l); }
 point rot90 () const { return point (-y, x); }
 point _rot90 () const { return point (y, -x); }
 point rot (cd t) const {
  double c = cos (t), s = sin (t);
  return point (x * c - y * s, x * s + y * c); } };
bool operator == (cp a, cp b) { return cmp (a.x, b.x) == 0 && cmp (a.y, b.y) == 0; }
bool operator != (cp a, cp b) { return cmp (a.x, b.x) != 0 || cmp (a.y, b.y) != 0; }
bool operator < (cp a, cp b) { return cmp (a.x, b.x) == 0 ? cmp (a.y, b.y) < 0 : cmp (a.x, b.x) < 0; }
point operator - (cp a) { return point (-a.x, -a.y); }
point operator + (cp a, cp b) { return point (a.x + b.x, a.y + b.y); }
point operator - (cp a, cp b) { return point (a.x - b.x, a.y - b.y); }
point operator * (cp a, cd b) { return point (a.x * b, a.y * b); }
point operator / (cp a, cd b) { return point (a.x / b, a.y / b); }
double dot (cp a, cp b) { return a.x * b.x + a.y * b.y; }
double det (cp a, cp b) { return a.x * b.y - a.y * b.x; }
double dis2 (cp a, cp b = point ()) { return sqr (a.x - b.x) + sqr (a.y - b.y); }
double dis (cp a, cp b = point ()) { return msqrt (dis2 (a, b)); }
```

3.4.1 Convex hull

Counter-clockwise, starting with the smallest point, and with the minimum number of points. Modify >= to > in `turn_left` to conserve all points on the hull.

```
bool turn_left (cp a, cp b, cp c) { return sgn (det (b - a, c - a)) >= 0; }
std::vector <point> convex_hull (std::vector <point> a) {
 int cnt = 0; std::sort (a.begin (), a.end ());
 static std::vector <point> ret; ret.resize (a.size () << 1);
 for (int i = 0; i < (int) a.size (); ++i) {
  while (cnt > 1 && turn_left (ret[cnt - 2], a[i], ret[cnt - 1])) --cnt;
  ret[cnt++] = a[i]; }
 int fixed = cnt;
 for (int i = (int) a.size () - 1; i >= 0; --i) {
  while (cnt > fixed && turn_left (ret[cnt - 2], a[i], ret[cnt - 1])) --cnt;
  ret[cnt++] = a[i]; }
 return std::vector <point> (ret.begin (), ret.begin () + cnt - 1); }
```

3.4.2 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 subgraph 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 vertice of the corresponding edge `u.p[(i + 1) % 3]` and `u.p[(i + 2) % 3]`.

```
const int N = 100000 + 5, MAX_TRIS = N * 6;
bool in_circumcircle (cp p1, cp p2, cp p3, cp p4) {
 double u11 = p1.x - p4.x, u21 = p2.x - p4.x, u31 = p3.x - p4.x;
 double u12 = p1.y - p4.y, u22 = p2.y - p4.y, u32 = p3.y - p4.y;
 double u13 = sqr (p1.x) - sqr (p4.x) + sqr (p1.y) - sqr (p4.y);
 double u23 = sqr (p2.x) - sqr (p4.x) + sqr (p2.y) - sqr (p4.y);
 double u33 = sqr (p3.x) - sqr (p4.x) + sqr (p3.y) - sqr (p4.y);
 double det = -u13 * u22 * u31 + u12 * u23 * u31 + u13 * u21 * u32 - u11 * u23 * u32 - u12 * u21 * u33 + u11 * u22 * u33;
 return sgn (det) > 0; }
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); }
typedef int side_t; struct tri; typedef tri* tri_r;
struct edge {
 tri_r t; side_t side;
 edge (tri_r t = 0, side_t side = 0) : t(t), side(side) {} };
struct tri {
 point p[3]; edge e[3]; tri_r child[3]; tri () {}
 tri (cp p0, cp p1, cp p2) { p[0] = p0; p[1] = p1; p[2] = p2;
  child[0] = child[1] = child[2] = 0; }
 bool has_child() const { return child[0] != 0; }
 int num_child() const { return child[0] == 0 ? 0 : child[1] == 0 ? 1 : child[2] == 0 ? 2 : 3; }
 bool contains (cp q) const {
  double a = side (p[0], p[1], q), b = side(p[1], p[2], q), c = side(p[2], p[0], q);
  return sgn (a) >= 0 && sgn (b) >= 0 && sgn (c) >= 0; } };
void set_edge (edge a, edge b) {
 if (a.t) a.t -> e[a.side] = b;
 if (b.t) b.t -> e[b.side] = a; }
class trig {
 public:
  tri tpool[MAX_TRIS], *tot;
  trig() { const double LOTS = 1E6;
   the_root = new (tot++) tri (point (-LOTS, -LOTS), point (LOTS, -LOTS), point (0, LOTS)); }
  tri_r find (cp p) const { return find (the_root, p); }
  void add_point (cp p) { add_point (find (the_root, p), p); }
 private:
  tri_r the_root;
  static tri_r find (tri_r root, cp p) {
   for(; ; ) { if (!root -> has_child ()) return root;
   else for (int i = 0; i < 3 && root -> child[i]; ++i)
    if (root -> child[i] -> contains (p))
     { root = root->child[i]; break; } } }
  void add_point (tri_r root, cp p) {
   tri_r tab, tbc, tca;
   tab = new (tot++) tri (root -> p[0], root -> p[1], p);
   tbc = new (tot++) tri (root -> p[1], root -> p[2], p);
   tca = new (tot++) tri (root -> p[2], root -> p[0], p);
   set_edge (edge (tab, 0), edge (tbc, 1)); set_edge (edge (tbc, 0), edge (tca, 1));
   set_edge (edge (tca, 0), edge (tab, 1)); set_edge (edge (tab, 2), root -> e[2]);
   set_edge (edge (tbc, 2), root -> e[0]); set_edge (edge (tca, 2), root -> e[1]);
   root -> child[0] = tab; root -> child[1] = tbc; root -> child[2] = tca;
```

```
  flip (tab, 2); flip (tbc, 2); flip (tca, 2); }
 void flip (tri_r t, side_t pi) {
  tri_r trj = t -> e[pi].t; int pj = t -> e[pi].side;
  if(!trj || !in_circumcircle (t -> p[0], t -> p[1], t -> p[2], trj -> p[pj])) return;
  tri_r trk = new (tot++) tri (t -> p[(pi + 1) % 3], trj -> p[pj], t -> p[pi]);
  tri_r trl = new (tot++) tri (trj -> p[(pj + 1) % 3], t -> p[pi], trj -> p[pj]);
  set_edge (edge (trk, 0), edge (trl, 0));
  set_edge (edge (trk, 1), t -> e[(pi + 2) % 3]); set_edge (edge (trk, 2), trj -> e[(pj + 1) % 3]);
  set_edge (edge (trl, 1), trj -> e[(pj + 2) % 3]); set_edge (edge (trl, 2), t -> e[(pi + 1) % 3]);
  t -> child[0] = trk; t -> child[1] = trl; t -> child[2] = 0;
  trj -> child[0] = trk; trj -> child[1] = trl; trj -> child[2] = 0;
  flip (trk, 1); flip (trk, 2); flip (trl, 1); flip (trl, 2); } };
void build (std::vector <point> ps, trig &t) {
 t.tot = t.tpool; std::random_shuffle (ps.begin (), ps.end ());
 for (point &p : ps) t.add_point (p); }
```

3.4.3 Nearest pair of points

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

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

3.4.4 Fermat point

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

```
point fermat_point (cp a, cp b, cp c) {
 if (a == b) return a; if (b == c) return b; if (c == a) return c;
 double ab = dis (a, b), bc = dis (b, c), ca = dis (c, a);
 double cosa = dot (b - a, c - a) / ab / ca;
 double cosb = dot (a - b, c - b) / ab / bc;
 double cosc = dot (b - c, a - c) / ca / bc;
 double sq3 = PI / 3.0; point mid;
 if (sgn (cosa + 0.5) < 0) mid = a;
 else if (sgn (cosb + 0.5) < 0) mid = b;
 else if (sgn (cosc + 0.5) < 0) mid = c;
 else if (sgn (det (b - a, c - a)) < 0) mid = line_intersect (line (a, b + (c - b).rot (sq3)), line (b, c + (a - c).rot (sq3)));
 else mid = line_intersect (line (a, c + (b - c).rot (sq3)), line (c, b + (a - b).rot (sq3)));
 return mid; }
```

3.4.5 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: $\csc(A + \pi/3) : \csc(B + \pi/3) : \csc(C + \pi/3)$.

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

4 Graph

```
template <int MAXN = 100000, int MAXM = 100000>
struct edge_list {
 int size, begin[MAXN], dest[MAXM], next[MAXM];
 void clear (int n) { size = 0; std::fill (begin, begin + n, -1); }
 edge_list (int n = MAXN) { clear (n); }
 void add_edge (int u, int v) { dest[size] = v; next[size] = begin[u]; begin[u] = size++; } };
template <int MAXN = 100000, int MAXM = 100000>
struct cost_edge_list {
 int size, begin[MAXN], dest[MAXM], next[MAXM], cost[MAXM];
 void clear (int n) { size = 0; std::fill (begin, begin + n, -1); }
 cost_edge_list (int n = MAXN) { clear (n); }
 void add_edge (int u, int v, int c) { dest[size] = v; next[size] = begin[u]; cost[size] = c; begin[u] = size++; } };
```

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).

```
template <int MAXN = 100000, int MAXM = 100000>
struct chordal_graph {
 int n; edge_list <MAXN, MAXM> e;
 int id[MAXN], seq[MAXN];
 void init () {
  struct point {
   int lab, u;
   point (int lab = 0, int u = 0) : lab (lab), u (u) {}
   bool operator < (const point &a) const { return lab < a.lab; } };
  std::fill (id, id + n, -1);
  static int label[MAXN]; std::fill (label, label + n, 0);
  std::priority_queue <point> q;
  for (int i = 0; i < n; ++i) q.push (point (0, i));
  for (int i = n - 1; i >= 0; --i) {
   for (; ~id[q.top ().u]; ) q.pop ();
   int u = q.top ().u; q.pop (); id[u] = i;
   for (int j = e.begin[u], v; ~j; j = e.next[j])
    if (v = e.dest[j], !~id[v]) ++label[v], q.push ( point (label[v], v)); }
  for (int i = 0; i < n; ++i) seq[id[i]] = i; }
 bool is_chordal () {
  static int vis[MAXN], q[MAXN]; std::fill (vis, vis + n, -1);
  for (int i = n - 1; i >= 0; --i) {
   int u = seq[i], t = 0, v;
   for (int j = e.begin[u]; ~j; j = e.next[j])
    if (v = e.dest[j], id[v] > id[u]) q[t++] = v;
   if (!t) continue; int w = q[0];
   for (int j = 0; j < t; ++j) if (id[q[j]] < id[w]) w = q[j];
   for (int j = e.begin[w]; ~j; j = e.next[j]) vis[e.dest[j]] = i;
   for (int j = 0; j < t; ++j) if (q[j] != w && vis[q[j]] != i) return 0;
  }
  return 1; }
 int min_color () {
  int res = 0;
  static int vis[MAXN], c[MAXN];
  std::fill (vis, vis + n, -1);
  std::fill (c, c + n, n);
  for (int i = n - 1; i >= 0; --i) {
   int u = seq[i];
   for (int j = e.begin[u]; ~j; j = e.next[j]) vis[c[e.dest[j]]] = i;
   int k; for (k = 0; vis[k] == i; ++k);
   c[u] = k; res = std::max (res, k + 1);
  }
  return res; } };
```

4.1.2 Euler characteristic

The Euler characteristic χ 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:

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

```
typedef bool BB[N]; struct max_clique {
const BB *e; int pk, level; const float Tlimit;
struct vertex { int i, d; vertex (int i) : i(i), d(0) {} };
typedef std::vector <vertex> vertices; vertices V;
typedef std::vector <int> colors; colors QMAX, Q;
std::vector <colors> C;
static bool desc_degree (const vertex &vi,const vertex &vj) { return vi.d > vj.d; }
void init_colors (vertices &v) {
 const int max_degree = v[0].d;
 for (int i = 0; i < (int) v.size(); ++i) v[i].d = std::min (i, max_degree) + 1; }
void set_degrees (vertices &v) {
 for (int i = 0, j; i < (int) v.size (); ++i)
  for (v[i].d = j = 0; j < (int) v.size (); ++j)
   v[i].d += e[v[i].i][v[j].i]; }
struct steps { int i1, i2; steps (): i1 (0), i2 (0) {} };
std::vector <steps> S;
bool cut1 (const int pi, const colors &A) {
 for (int i = 0; i < (int) A.size (); ++i)
  if (e[pi][A[i]]) return true; return false; }
void cut2 (const vertices &A, vertices & B) {
 for (int i = 0; i < (int) A.size () - 1; ++i)
  if (e[A.back().i][A[i].i]) B.push_back(A[i].i); }
void color_sort (vertices & R) {
 int j = 0, maxno = 1, min_k = std::max ((int) QMAX.size () - (int) Q.size() + 1, 1);
 C[1].clear (); C[2].clear ();
 for (int i = 0; i < (int) R.size (); ++i) {
  int pi = R[i].i, k = 1; while (cut1(pi, C[k])) ++k;
  if (k > maxno) maxno = k, C[maxno + 1].clear();
  C[k].push_back (pi); if (k < min_k) R[j++].i = pi; }
 if (j > 0) R[j - 1].d = 0;
 for (int k = min_k; k <= maxno; ++k)
  for (int i = 0; i < (int) C[k].size (); ++i)
   R[j].i = C[k][i], R[j++].d = k; }
void expand_dyn (vertices &R) {
 S[level].i1 = S[level].i1 + S[level - 1].i1 - S[level].i2;
 S[level].i2 = S[level - 1].i1;
 while ((int) R.size ()) {
  if ((int) Q.size () + R.back ().d > (int) QMAX.size ()) {
   Q.push_back (R.back ().i); vertices Rp; cut2 (R, Rp);
   if ((int) Rp.size ()) {
    if((float) S[level].i1 / ++pk < Tlimit) degree_sort (Rp);
    color_sort (Rp); ++S[level].i1, ++level;
    expand_dyn (Rp); --level;
   } else if ((int) Q.size () > (int) QMAX.size ()) QMAX = Q;
   Q.pop_back (); } else return; R.pop_back (); } }
void mcqdyn (int *maxclique, int &sz) {
 set_degrees (V); std::sort(V.begin (), V.end (), desc_degree); init_colors (V);
 for (int i = 0; i < (int) V.size () + 1; ++i) S[i].i1 = S[i].i2 = 0;
 expand_dyn (V); sz = (int) QMAX.size ();
 for (int i = 0; i < (int) QMAX.size (); i++) maxclique[i] = QMAX[i]; }
void degree_sort (vertices & R) {
 set_degrees(R); std::sort(R.begin(), R.end(), desc_degree); }
max_clique (const BB *conn, const int sz, const float tt = .025) : pk (0), level (1), Tlimit (tt) {
 for(int i = 0; i < sz; i++) V.push_back (vertex (i));
 e = conn, C.resize (sz + 1), S.resize (sz + 1); } };

BB e[N]; int ans, sol[N]; for (...) e[x][y] = e[y][x] = true;
max_clique mc (e, n); mc.mcqdyn (sol, ans); //0-based.
for (int i = 0; i < ans; ++i) std::cout << sol[i] << 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 (`idom[]`) of each node, `idom[x]` will be x if x does not have a dominator, and will be -1 if x is not reachable from s.

```
template <int MAXN = 100000, int MAXM = 100000>
struct dominator_tree {
 int dfn[MAXN], sdom[MAXN], idom[MAXN], id[MAXN], f[MAXN], fa[MAXN], smin[MAXN], stamp;
 void predfs (int x, const edge_list <MAXN, MAXM> & succ) {
  id[dfn[x] = stamp++] = x;
  for (int i = succ.begin[x]; ~i; i = succ.next[i]) {
   int y = succ.dest[i];
   if (dfn[y] < 0) { f[y] = x; predfs (y, succ); } } }
 int getfa (int x) {
  if (fa[x] == x) return x;
  int ret = getfa (fa[x]);
  if (dfn[sdom[smin[fa[x]]]] < dfn[sdom[smin[x]]])
   smin[x] = smin[fa[x]];
  return fa[x] = ret; }
 void solve (int s, int n, const edge_list <MAXN, MAXM> &succ) {
  std::fill (dfn, dfn + n, -1); std::fill (idom, idom + n, -1);
  static edge_list <MAXN, MAXM> pred, tmp; pred.clear (n);
  for (int i = 0; i < n; ++i) for (int j = succ.begin[i]; ~j; j = succ.next[j])
   pred.add_edge (succ.dest[j], i);
  stamp = 0; tmp.clear (n); predfs (s, succ);
  for (int i = 0; i < stamp; ++i) fa[id[i]] = smin[id[i]] = id[i];
  for (int o = stamp - 1; o >= 0; --o) {
   int x = id[o];
   if (o) {
    sdom[x] = f[x];
    for (int i = pred.begin[x]; ~i; i = pred.next[i]) {
     int p = pred.dest[i];
     if (dfn[p] < 0) continue;
```

```
    if (dfn[p] > dfn[x]) { getfa (p); p = sdom[smin[p]]; }
    if (dfn[sdom[x]] > dfn[p]) sdom[x] = p; }
   tmp.add_edge (sdom[x], x); }
  while (~tmp.begin[x]) {
   int y = tmp.dest[tmp.begin[x]];
   tmp.begin[x] = tmp.next[tmp.begin[x]]; getfa (y);
   if (x != sdom[smin[y]]) idom[y] = smin[y];
   else idom[y] = x; }
  for (int v : succ[x]) if (f[v] == x) fa[v] = x; }
 idom[s] = s; for (int i = 1; i < stamp; ++i) {
  int x = id[i]; if (idom[x] != sdom[x]) idom[x] = idom[idom[x]]; } } };
```

4.3.3 Stoer Wagner algorithm

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

```
template <int MAXN = 500>
struct stoer_wagner {
 int n, edge[MAXN][MAXN];
 int dist[MAXN];
 bool vis[MAXN], bin[MAXN];
 stoer_wagner () {
  memset (edge, 0, sizeof (edge));
  memset (bin, false, sizeof (bin)); }
 int contract (int &s, int &t)  {
  memset (dist, 0, sizeof (dist));
  memset (vis, false, sizeof (vis));
  int i, j, k, mincut, maxc;
  for (i = 1; i <= n; i++) {
   k = -1; maxc = -1;
   for (j = 1; j <= n; j++)
    if (!bin[j] && !vis[j] && dist[j] > maxc) {
     k = j; maxc = dist[j]; }
   if (k == -1) return mincut;
   s = t; t = k; mincut = maxc; vis[k] = true;
   for (j = 1; j <= n; j++) if (!bin[j] && !vis[j])
    dist[j] += edge[k][j]; }
  return mincut; }
 int solve () {
  int mincut, i, j, s, t, ans;
  for (mincut = INF, i = 1; i < n; i++) {
   ans = contract (s, t); bin[t] = true;
   if (mincut > ans) mincut = ans;
   if (mincut == 0) return 0;
   for (j = 1; j <= n; j++) if (!bin[j])
    edge[s][j] = (edge[j][s] += edge[j][t]); }
  return mincut; } };
```

4.3.4 Tarjan

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

```
template <int MAXN = 1000000, int MAXM = 1000000>
struct tarjan {
 int comp[MAXN], size;
 int dfn[MAXN], ind, low[MAXN], ins[MAXN], stk[MAXN], stks;
 void dfs (const edge_list <MAXN, MAXM> &e, int i) {
  dfn[i] = low[i] = ind++;
  ins[i] = 1; stk[stks++] = i;
  for (int x = e.begin[i]; ~x; x = e.next[x]) {
   int j = e.dest[x]; if (!~dfn[j]) {
    dfs (e, j);
    if (low[i] > low[j]) low[i] = low[j];
    if (low[j] > dfn[i]); //edge-biconnected
    if (low[j] >= dfn[i]); //vertex-biconnected
   } else if (ins[j] && low[i] > dfn[j])
    low[i] = dfn[j]; }
  if (dfn[i] == low[i]) {    //strongly-connected
   for (int j = -1; j != i;
    j = stk[--stks], ins[j] = 0, comp[j] = size);
   ++size; } }
 void solve (const edge_list <MAXN, MAXM> &e, int n) {
  size = ind = stks = 0;
  std::fill (dfn, dfn + n, -1);
  for (int i = 0; i < n; ++i) if (!~dfn[i])
   dfs (e, i); } };
```

4.4 Flow

4.4.1 Maximum flow

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

```
template <int MAXN = 1000, int MAXM = 100000>
struct isap {
 struct flow_edge_list {
  int size, begin[MAXN], dest[MAXM], next[MAXM], flow[MAXM];
  void clear (int n) { size = 0; std::fill (begin, begin + n, -1); }
  flow_edge_list (int n = MAXN) { clear (n); }
  void add_edge (int u, int v, int f) {
   dest[size] = v; next[size] = begin[u]; flow[size] = f; begin[u] = size++;
   dest[size] = u; next[size] = begin[v]; flow[size] = 0; begin[v] = size++; } };
 int pre[MAXN], d[MAXN], gap[MAXN], cur[MAXN], que[MAXN], vis[MAXN];
 int solve (flow_edge_list &e, int n, int s, int t) {
  for (int i = 0; i < n; ++i) { pre[i] = d[i] = gap[i] = vis[i] = 0; cur[i] = e.begin[i]; }
  int l = 0, r = 0; que[0] = t; gap[0] = 1; vis[t] = true;
  while (l <= r) { int u = que[l++];
   for (int i = e.begin[u]; ~i; i = e.next[i])
    if (e.flow[i] == 0 && !vis[e.dest[i]]) {
     que[++r] = e.dest[i];
     vis[e.dest[i]] = true;
     d[e.dest[i]] = d[u] + 1;
     ++gap[d[e.dest[i]]]; } }
  for (int i = 0; i < n; ++i) if (!vis[i]) d[i] = n, ++gap[n];
  int u = pre[s] = s, v, maxflow = 0;
  while (d[s] < n) {
   v = n; for (int i = cur[u]; ~i; i = e.next[i])
    if (e.flow[i] && d[u] == d[e.dest[i]] + 1) {
     v = e.dest[i]; cur[u] = i; break; }
   if (v < n) {
    pre[v] = u; u = v;
    if (v == t) {
     int dflow = INF, p = t; u = s;
     while (p != s) { p = pre[p]; dflow = std::min (dflow, e.flow[cur[p]]); }
     maxflow += dflow; p = t;
     while (p != s) { p = pre[p]; e.flow[cur[p]] -= dflow; e.flow[cur[p] ^ 1] += dflow; } }
   } else {
    int mindist = n + 1;
    for (int i = e.begin[u]; ~i; i = e.next[i])
     if (e.flow[i] && mindist > d[e.dest[i]]) {
      mindist = d[e.dest[i]]; cur[u] = i; }
    if (!--gap[d[u]]) return maxflow;
    gap[d[u] = mindist + 1]++; u = pre[u]; } }
  return maxflow; } };
template <int MAXN = 1000, int MAXM = 100000>
struct dinic {
 struct flow_edge_list {
  int size, begin[MAXN], dest[MAXM], next[MAXM], flow[MAXM];
  void clear (int n) { size = 0; std::fill (begin, begin + n, -1); }
  flow_edge_list (int n = MAXN) { clear (n); }
  void add_edge (int u, int v, int f) {
   dest[size] = v; next[size] = begin[u]; flow[size] = f; begin[u] = size++;
   dest[size] = u; next[size] = begin[v]; flow[size] = 0; begin[v] = size++; } };
 int n, s, t, d[MAXN], w[MAXN], q[MAXN];
 int bfs (flow_edge_list &e) {
  std::fill (d, d + n, -1);
  int l, r; q[l = r = 0] = s, d[s] = 0;
  for (; l <= r; l ++)
   for (int k = e.begin[q[l]]; ~k; k = e.next[k])
    if (!~d[e.dest[k]] && e.flow[k] > 0) d[e.dest[k]] = d[q[l]] + 1, q[++r] = e.dest[k];
  return ~d[t] ? 1 : 0; }
 int dfs (flow_edge_list &e, int u, int ext) {
  if (u == t) return ext; int k = w[u], ret = 0;
  for (; ~k; k = e.next[k], w[u] = k) {
   if (ext == 0) break;
   if (d[e.dest[k]] == d[u] + 1 && e.flow[k] > 0) {
    int flow = dfs (e, e.dest[k], std::min (e.flow[k], ext));
    if (flow > 0) {
     e.flow[k] -= flow, e.flow[k ^ 1] += flow;
     ret += flow, ext -= flow; } } }
  if (!~k) d[u] = -1; return ret; }
 int solve (flow_edge_list &e, int n_, int s_, int t_) {
  int ans = 0; n = n_; s = s_; t = t_;
  while (bfs (e)) {
   for (int i = 0; i < n; ++i) w[i] = e.begin[i];
   ans += dfs (e, s, INF); }
  return ans; } };
```

4.4.2 Minimum cost flow

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

```
template <int MAXN = 1000, int MAXM = 100000>
struct minimum_cost_flow {
 struct cost_flow_edge_list {
  int size, begin[MAXN], dest[MAXM], next[MAXM], cost[MAXM], flow[MAXM];
  void clear (int n) { size = 0; std::fill (begin, begin + n, -1); }
  cost_flow_edge_list (int n = MAXN) { clear (n); }
  void add_edge (int u, int v, int c, int f) {
   dest[size] = v; next[size] = begin[u]; cost[size] = c; flow[size] = f; begin[u] = size++;
   dest[size] = u; next[size] = begin[v]; cost[size] = -c; flow[size] = 0; begin[v] = size++; } };
 int n, s, t, prev[MAXN], dist[MAXN], occur[MAXN];
 bool augment (cost_flow_edge_list &e) {
  std::vector <int> queue;
  std::fill (dist, dist + n, INF); std::fill (occur, occur + n, 0);
  dist[s] = 0; occur[s] = true; queue.push_back (s);
  for (int head = 0; head < (int)queue.size(); ++head) {
   int x = queue[head];
   for (int i = e.begin[x]; ~i; i = e.next[i]) {
    int y = e.dest[i];
```

```
    if (e.flow[i] && dist[y] > dist[x] + e.cost[i]) {
     dist[y] = dist[x] + e.cost[i]; prev[y] = i;
     if (!occur[y]) {
      occur[y] = true; queue.push_back (y); } } }
   occur[x] = false; }
  return dist[t] < INF; }
 std::pair <int, int> solve (cost_flow_edge_list &e, int n_, int s_, int t_) {
  n = n_; s = s_; t = t_; std::pair <int, int> ans = std::make_pair (0, 0);
  while (augment (e)) {
   int num = INF;
   for (int i = t; i != s; i = e.dest[prev[i] ^ 1])
    num = std::min (num, e.flow[prev[i]]);
   ans.first += num;
   for (int i = t; i != s; i = e.dest[prev[i] ^ 1]) {
    e.flow[prev[i]] -= num; e.flow[prev[i] ^ 1] += num;
    ans.second += num * e.cost[prev[i]]; } }
  return ans; } };
template <int MAXN = 1000, int MAXM = 100000>
struct zkw_flow {
 struct cost_flow_edge_list {
  int size, begin[MAXN], dest[MAXM], next[MAXM], cost[MAXM], flow[MAXM];
  void clear (int n) { size = 0; std::fill (begin, begin + n, -1); }
  cost_flow_edge_list (int n = MAXN) { clear (n); }
  void add_edge (int u, int v, int c, int f) {
   dest[size] = v; next[size] = begin[u]; cost[size] = c; flow[size] = f; begin[u] = size++;
   dest[size] = u; next[size] = begin[v]; cost[size] = -c; flow[size] = 0; begin[v] = size++; } };
 int n, s, t, tf, tc, dis[MAXN], slack[MAXN], visit[MAXN];
 int modlable() {
  int delta = INF;
  for (int i = 0; i < n; i++) {
   if (!visit[i] && slack[i] < delta) delta = slack[i];
   slack[i] = INF; }
  if (delta == INF) return 1;
  for (int i = 0; i < n; i++) if (visit[i]) dis[i] += delta;
  return 0; }
 int dfs (cost_flow_edge_list &e, int x, int flow) {
  if (x == t) { tf += flow; tc += flow * (dis[s] - dis[t]); return flow; }
  visit[x] = 1; int left = flow;
  for (int i = e.begin[x]; ~i; i = e.next[i])
   if (e.flow[i] > 0 && !visit[e.dest[i]]) {
    int y = e.dest[i];
    if (dis[y] + e.cost[i] == dis[x]) {
     int delta = dfs (e, y, std::min (left, e.flow[i]));
     e.flow[i] -= delta; e.flow[i ^ 1] += delta; left -= delta;
     if (!left) { visit[x] = false; return flow; }
    } else
     slack[y] = std::min (slack[y], dis[y] + e.cost[i] - dis[x]); }
  return flow - left; }
 std::pair <int, int> solve (cost_flow_edge_list &e, int n_, int s_, int t_) {
  n = n_; s = s_; t = t_; tf = tc = 0;
  std::fill (dis + 1, dis + t + 1, 0);
  do { do {
   std::fill (visit + 1, visit + t + 1, 0);
  } while (dfs (e, s, INF)); } while (!modlable ());
  return std::make_pair (tf, tc);
 } };
```

4.5 Matching

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

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

where $\mathrm{odd}(H)$ 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.

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

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[]`.

```
struct weighted_blossom {
 static const int INF = INT_MAX, MAXN = 400;
 struct edge{ int u, v, w; edge (int u = 0, int v = 0, int w = 0): u(u), v(v), w(w) {} };
 int n, n_x;
 edge g[MAXN * 2 + 1][MAXN * 2 + 1];
 int lab[MAXN * 2 + 1], match[MAXN * 2 + 1], slack[MAXN * 2 + 1], st[MAXN * 2 + 1], pa[MAXN * 2 + 1];
 int flower_from[MAXN * 2 + 1][MAXN + 1], S[MAXN * 2 + 1], vis[MAXN * 2 + 1];
 std::vector <int> flower[MAXN * 2 + 1]; std::queue <int> q;
 int e_delta (const edge &e) { return lab[e.u] + lab[e.v] - g[e.u][e.v].w * 2; }
 void update_slack (int u, int x) { if (!slack[x] || e_delta (g[u][x]) < e_delta (g[slack[x]][x])) slack[x] = u; }
 void set_slack (int x) { slack[x] = 0; for (int u = 1; u <= n; ++u) if(g[u][x].w > 0 && st[u] != x && S[st[u]] == 0)
  update_slack(u, x); }
 void q_push (int x) {
  if (x <= n) q.push (x);
  else for (size_t i = 0; i < flower[x].size (); i++) q_push (flower[x][i]); }
 void set_st (int x, int b) {
  st[x] = b; if (x > n) for (size_t i = 0; i < flower[x].size (); ++i) set_st (flower[x][i], b); }
 int get_pr (int b, int xr) {
  int pr = std::find (flower[b].begin (), flower[b].end (), xr) - flower[b].begin ();
  if (pr % 2 == 1) { std::reverse (flower[b].begin () + 1, flower[b].end ()); return (int) flower[b].size () - pr;
```

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

4.5.3 Hopcoft-Karp algorithm

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

```
template <int MAXN = 100000, int MAXM = 100000>
struct hopcoft_karp {
 int mx[MAXN], my[MAXM], lv[MAXN];
 bool dfs (edge_list <MAXN, MAXM> &e, int x) {
  for (int i = e.begin[x]; ~i; i = e.next[i]) {
   int y = e.dest[i], w = my[y];
   if (!~w || (lv[w] + 1 == lv[x] && dfs (e, w))) {
    mx[x] = y; my[y] = x; return true; } }
  lv[x] = -1; return false; }
 int solve (edge_list <MAXN, MAXM> &e, int n, int m) {
  std::fill (mx, mx + n, -1); std::fill (my, my + m, -1);
  for (int ans = 0; ; ) {
   std::vector <int> q;
   for (int i = 0; i < n; ++i)
    if (mx[i] == -1) {
     lv[i] = 0; q.push_back (i);
    } else lv[i] = -1;
   for (int head = 0; head < (int) q.size(); ++head) {
    int x = q[head];
    for (int i = e.begin[x]; ~i; i = e.next[i]) {
     int y = e.dest[i], w = my[y];
     if (~w && lv[w] < 0) { lv[w] = lv[x] + 1; q.push_back (w); } } }
   int d = 0; for (int i = 0; i < n; ++i) if (!~mx[i] && dfs (e, i)) ++d;
   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.

```
template <int MAXN = 500>
struct kuhn_munkres {
 int n, w[MAXN][MAXN], lx[MAXN], ly[MAXN], m[MAXN], way[MAXN], sl[MAXN];
 bool u[MAXN];
 void hungary(int x) {
  m[0] = x; int j0 = 0;
  std::fill (sl, sl + n + 1, INF); std::fill (u, u + n + 1, false);
  do {
   u[j0] = true; int i0 = m[j0], d = INF, j1 = 0;
   for (int j = 1; j <= n; ++j)
    if (u[j] == false) {
     int cur = -w[i0][j] - lx[i0] - ly[j];
     if (cur < sl[j]) { sl[j] = cur; way[j] = j0; }
     if (sl[j] < d) { d = sl[j]; j1 = j; } }
   for (int j = 0; j <= n; ++j) {
    if (u[j]) { lx[m[j]] += d; ly[j] -= d; }
    else sl[j] -= d; }
   j0 = j1; } while (m[j0] != 0);
  do {
   int j1 = way[j0]; m[j0] = m[j1]; j0 = j1;
  } while (j0); }
 int solve() {
  for (int i = 1; i <= n; ++i) m[i] = lx[i] = ly[i] = way[i] = 0;
  for (int i = 1; i <= n; ++i) hungary (i);
  int sum = 0; for (int i = 1; i <= n; ++i) sum += w[m[i]][i];
  return sum; } };
```

4.6 Path

4.6.1 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 $A = \{a_1, \ldots, a_n\}$ and destination vertices

$B = \{b_1, \ldots, b_n\}$, and also assign a weight ω_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 $e(a,b) = \sum_{P:a\to b} \omega(P)$ 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 $(P_1, \ldots, P_n)$ of paths in G with the following properties:

1. There exists a permutation σ of $\{1, 2, ..., n\}$ 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 $(P_1, \ldots, P_n)$, we denote by $\sigma(P)$ the permutation of σ from the first condition.

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

$$\det(M) = \sum_{(P_1,\ldots,P_n)\colon A\to B} \operatorname{sign}(\sigma(P)) \prod_{i=1}^{n} \omega(P_i).$$

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 $(1, 2, \ldots, n)$, 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

4.8 Prufer sequence

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

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

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

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

4.9 Spanning tree counting

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

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

4.10 Tree hash

`A[n]` is the hash of the sub-tree with root `n`.
`B[n]` is the hash of the whole tree with root `n`.

```
template <int MAXN = 100000, int MAXM = 200000, long long MOD = 1000000000000000031l>
struct tree_hash {
 static long long ra[MAXN];
 tree_hash () {
  std::mt19937_64 mt (time (0));
  std::uniform_int_distribution <long long> uid (0, MOD - 1);
  for (int i = 0; i < MAXN; ++i) ra[i] = uid (mt); }
 struct node {
  std::vector <long long> s; int d1, d2; long long h1, h2;
  node () { d1 = d2 = 0; }
  void add (int d, long long v) {
   s.push_back (v);
   if (d > d1) d2 = d1, d1 = d; else if (d > d2) d2 = d; }
  long long hash () {
   h1 = h2 = 1; for (long long i : s) {
    h1 = mul_mod (h1, ra[d1] + i, MOD);
    h2 = mul_mod (h2, ra[d2] + i, MOD); } return h1; }
 std::pair <int, long long> del (int d, long long v) {
  if (d == d1) return { d2 + 1, mul_mod (h2, inverse (ra[d2] + v, MOD), MOD) };
  return { d1 + 1, mul_mod (h1, inverse (ra[d1] + v, MOD), MOD) }; } };
 std::pair <int, long long> u[MAXN]; node tree[MAXN];
 long long A[MAXN], B[MAXN];
 void dfs1 (const edge_list <MAXN, MAXM> &e, int x, int p = -1) {
  tree[x] = node ();
    for (int i = e.begin[x]; ~i; i = e.next[i]) {
   int c = e.dest[i]; if (c != p) {
    dfs1 (e, c, x); tree[x].add (tree[c].d1 + 1, tree[c].h1); } }
   A[x] = tree[x].hash (); }
 void dfs2 (const edge_list <MAXN, MAXM> &e, int x, int p = -1) {
  if (~p) tree[x].add (u[x].first, u[x].second);
  B[x] = tree[x].hash ();
   for (int i = e.begin[x]; ~i; i = e.next[i]) {
   int c = e.dest[i]; if (c != p) {
    u[c] = tree[x].del (tree[c].d1 + 1, tree[c].h1);
    dfs2 (e, c, x); } } }
 void solve (const edge_list <MAXN, MAXM> &e, int root) {
  dfs1 (e, root); dfs2 (e, root); } };
template <int MAXN, int MAXM, long long MOD>
long long tree_hash <MAXN, MAXM, MOD>::ra[MAXN];
```

5 Mathematics

5.1 Computation

5.1.1 Adaptive Simpson's method

Compute $\int_l^r f(x)dx$ with error less than ϵ.

```
struct simpson {
 double area (double (*f) (double), double l, double r) {
  double m = l + (r - l) / 2;
  return (f (l) + 4 * f (m) + f (r)) * (r - l) / 6; }
 double solve (double (*f) (double), double l, double r, double eps, double a) {
  double m = l + (r - l) / 2;
  double left = area (f, l, m), right = area (f, m, r);
  if (fabs (left + right - a) <= 15 * eps) return left + right + (left + right - a) / 15.0;
  return solve (f, l, m, eps / 2, left) + solve (f, m, r, eps / 2, right); }
 double solve (double (*f) (double), double l, double r, double eps) {
  return solve (f, l, r, eps, area (f, l, r)); } };
```

5.1.2 Dirichlet convolution

5.1.3 Dirichlet inversion

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

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

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

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

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

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

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

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

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

Usage:

1. `CUBEN` should be $N^{\frac{1}{3}}$.
2. Pass `p_f` that returns the prefix sum of $f(x)(1 \le x < th)$.
3. Pass `p_g` that returns the prefix sum of $g(x)(0 \le x \le N)$.
4. Pass `p_c` that returns the prefix sum of $f * g(x)(0 \le x \le N)$.
5. Pass `th` as the thereshold, 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 $\frac{n}{x}$.

```
template <int CUBEN = 3000>
struct prefix_mul {
 typedef long long (*func) (long long);
 func p_f, p_g, p_c;
 long long mod, th, inv, n, mem[CUBEN];
 prefix_mul (func p_f, func p_g, func p_c, long long th, long long mod, long long inv) :
  p_f (p_f), p_g (p_g), p_c (p_c), th (th), mod (mod), inv (inv) {}
 void init (long long n) {
  prefix_mul::n = n;
  for (long long i = 1, la; i <= n; i = la + 1) {
   if ((la = n / (n / i)) < th) continue;
   long long &ans = mem[n / la] = p_c (la);
   for (long long j = 2, ne; j <= la; j = ne + 1) {
    ne = la / (la / j);
    ans = (ans + mod - (p_g (ne) - p_g (j - 1) + mod) *
      (la / j < th ? p_f (la / j) : mem[n / (la / j)]) % mod);
    if (ans >= mod) ans -= mod; }
   if (inv != 1) ans = ans * inv % mod; } }
 long long ans (long long x) {
  if (n / x < th) return p_f (n / x);
  return mem[n / (n / x)]; } };
```

5.1.4 Euclidean-like algorithm

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

```
long long solve(long long n, long long a, long long b, long long m){
 if (b == 0) return n * (a / m);
 if (a >= m) return n * (a / m) + solve (n, a % m, b, m);
 if (b >= m) return (n - 1) * n / 2 * (b / m) + solve (n, a, b % m, m);
 return solve ((a + b * n) / m, (a + b * n) % m, m, b); }
```

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.

```
template <int SN = 110000, int D = 2>
struct ees {
 int co[SN], prime[SN], psize, sn;
 long long powa[D + 1][SN], powb[D + 1][SN];
 long long funca[SN], funcb[SN];
 long long pow (long long x, int n) {
  long long res = 1;
  for (int i = 0; i < n; ++i) res *= x;
  return res; }
 long long pre_pow (long long x, int n) {
  if (n == 0) return x;
  if (n == 1) return (1 + x) * x / 2;
  if (n == 2) return (1 + 2 * x) * (1 + x) * x / 6;
  return 0; }
 long long pfunc (long long p) { return -1; }
 long long cfunc (long long k, long long p) { return 0; }
 void assemble () {
  for (int i = 1; i <= sn; ++i) {
   funca[i] = -powa[0][i];
   funcb[i] = -powb[0][i]; } }
 void init (long long n) {
  sn = std::max ((int) (ceil (sqrt (n)) + 1), 2);
  psize = 0; for (int i = 2; i <= sn; ++i) {
   if (!co[i]) prime[psize++] = i;
   for (int j = 0; 1LL * i * prime[j] <= sn; ++j) {
    co[i * prime[j]] = 1;
    if (i % prime[j] == 0) break; } }
  for (int d = 0; d <= D; ++d) {
   long long *pa = powa[d], *pb = powb[d];
   for (int i = 1; i <= sn; ++i) pa[i] = pre_pow (i, d) - 1;
   for (int i = 1; i <= sn; ++i) pb[i] = pre_pow (n / i, d) - 1;
   for (int i = 0; i < psize; ++i) { int &pi = prime[i];
    for (int j = 1; j <= sn; ++j) if (n / j >= 1LL * pi * pi) {
     long long ch = n / j / pi;
     pb[j] -= ((ch <= sn ? pa[ch] : pb[j * pi]) - pa[pi - 1]) * pow (pi, d);
    } else break;
    for (int j = sn; j >= 1; --j) if (j >= 1LL * pi * pi)
     pa[j] -= (pa[j / pi] - pa[pi - 1]) * pow (pi, d);
    else break; } }
  assemble (); }
 void dfs (int x, int f, long long mul, long long val, long long n, long long &res) {
  for (; x < psize && mul * prime[x] * prime[x] <= n; ++x) {
   long long nmul = mul * prime[x], nval = val * pfunc (prime[x]);
   for (; nmul <= n; nmul *= prime[x], nval = cfunc (nval, prime[x]))
    dfs (x + 1, prime[x], nmul, nval, n, res); }
  if (n / mul > f) res += val * ((n / mul <= sn ? funca[n / mul] : funcb[mul]) - funca[f]);
  if (f > 1 && mul % (f * f) == 0) res += val; }
 long long solve (long long n) {
  if (n == 0) return 0;
  long long res = 1;
  init (n); dfs (0, 1, 1, 1, n, res);
  return res; } };
```

5.1.6 Fast power module

Compute $x^n \mod mod$.

```
int fpm (int x, int n, int mod) {
 int ans = 1, mul = x; while (n) {
  if (n & 1) ans = int (1ll * ans * mul % mod);
  mul = int (1ll * mul * mul % mod); n >>= 1; }
 return ans; }
long long mul_mod (long long x, long long y, long long mod) {
 long long t = (x * y - (long long) ((long double) x / mod * y + 1E-3) * mod) % mod;
 return t < 0 ? t + mod : t; }
long long llfpm (long long x, long long n, long long mod) {
 long long ans = 1, mul = x; while (n) {
  if (n & 1) ans = mul_mod (ans, mul, mod);
  mul = mul_mod (mul, mul, mod); n >>= 1; }
 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

$$\sum_{a=1}^{n} \sum_{b=1}^{n} gcd^2(a,b) = \sum_{d=1}^{n} d^2 \sum_{i=1}^{\lfloor \frac{n}{d} \rfloor} \sum_{j=1}^{\lfloor \frac{n}{d} \rfloor} [gcd(i,j) = 1]$$

$$= \sum_{d=1}^{n} d^2 \sum_{i=1}^{\lfloor \frac{n}{d} \rfloor} \sum_{j=1}^{\lfloor \frac{n}{d} \rfloor} \sum_{t|gcd(i,j)} \mu(t)$$

$$= \sum_{d=1}^{n} d^2 \sum_{t=1}^{\lfloor \frac{n}{d} \rfloor} \mu(t) \sum_{i=1}^{\lfloor \frac{n}{d} \rfloor} [t|i] \sum_{j=1}^{\lfloor \frac{n}{d} \rfloor} [t|j]$$

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

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

Moreover, let $l = dt$, then

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

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

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

5.1.9 Pólya enumeration theorem

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

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

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 seperating the graphs with the number of edges, we let $f(t) = 1 + t$, and examine the coefficient of t^i for a graph with i edges, and when seperating 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 $x^i y^j z^k$.

5.1.10 Zeller's congruence

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

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

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

```
struct bsgs {
 int solve (int a, int b, int c) {
  std::unordered_map <int, int> bs;
  int m = (int) sqrt ((double) c) + 1, res = 1;
  for (int i = 0; i < m; ++i) {
   if (bs.find (res) == bs.end ()) bs[res] = i;
   res = int (1LL * res * a % c); }
  int mul = 1, inv = (int) inverse (a, c);
  for (int i = 0; i < m; ++i) mul = int (1LL * mul *
      inv % c);
  res = b % c;
  for (int i = 0; i < m; ++i) {
   if (bs.find (res) != bs.end ()) return i * m + bs[
       res];
   res = int (1LL * res * mul % c); }
  return -1; } };
```

5.3.2 Chinese remainder theorem

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

```
struct crt {
 long long fix (const long long &a, const long long &b
     ) { return (a % b + b) % b; }
 bool solve (const std::vector <std::pair <long long,
     long long>> &in, std::pair <long long, long long>
      &out) {
  out = std::make_pair (1LL, 1LL);
  for (int i = 0; i < (int) in.size (); ++i) {
   long long n, u;
   euclid (out.second, in[i].second, n, u);
   long long divisor = std::__gcd (out.second, in[i].
       second);
   if ((in[i].first - out.first) % divisor) return
       false;
   n *= (in[i].first - out.first) / divisor;
   n = fix (n, in[i].second);
   out.first += out.second * n;
   out.second *= in[i].second / divisor;
   out.first = fix (out.first, out.second); }
  return true; } };
```

5.3.3 Extended Euclidean algorithm

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

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

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 $x_{k+1} = x_0 x_k + n y_0 y_k, y_{k+1} = x_0 y_k + y_0 x_k$.

```
template <int MAXN = 100000>
struct pell {
 std::pair <long long, long long> solve (long long n)
     {
  static long long p[MAXN], q[MAXN], g[MAXN], h[MAXN],
      a[MAXN];
  p[1] = q[0] = h[1] = 1; p[0] = q[1] = g[1] = 0;
  a[2] = (long long) (floor (sqrtl (n) + 1e-7L));
  for (int i = 2; ; ++i) {
   g[i] = -g[i - 1] + a[i] * h[i - 1];
   h[i] = (n - g[i] * g[i]) / h[i - 1];
   a[i + 1] = (g[i] + a[2]) / h[i];
   p[i] = a[i] * p[i - 1] + p[i - 2];
   q[i] = a[i] * q[i - 1] + q[i - 2];
   if (p[i] * p[i] - n * q[i] * q[i] == 1)
    return { p[i], q[i] }; } } };
```

5.3.5 Quadric residue

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

```
struct quadric {
 void multiply(long long &c, long long &d, long long a
     , long long b, long long w, long long p) {
  int cc = (a * c + b * d % p * w) % p;
  int dd = (a * d + b * c) % p; c = cc, d = dd; }
 bool solve(int n, int p, int &x) {
  if (n == 0) return x = 0, true; if (p == 2) return x
      = 1, true;
  if (power (n, p / 2, p) == p - 1) return false;
  long long c = 1, d = 0, b = 1, a, w;
  do { a = rand() % p; w = (a * a - n + p) % p;
   if (w == 0) return x = a, true;
  } while (power (w, p / 2, p) != p - 1);
  for (int times = (p + 1) / 2; times; times >>= 1) {
   if (times & 1) multiply (c, d, a, b, w, p);
   multiply (a, b, a, b, w, p); }
  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 $\mathbf{c^T x}$ subject to $\mathbf{Ax} \leq \mathbf{b}, \mathbf{x} \geq \mathbf{0}$ is equivalent to minimizing $\mathbf{b^T y}$ subject to $\mathbf{A^T x} \geq \mathbf{c}, \mathbf{y} \geq \mathbf{0}$.

```
template <int MAXN = 100, int MAXM = 100>
struct simplex {
 int n, m; double a[MAXM][MAXN], b[MAXM], c[MAXN];
 bool infeasible, unbounded;
 double v, an[MAXN + MAXM]; int q[MAXN + MAXM];
 void pivot (int l, int e) {
  std::swap (q[e], q[l + n]);
  double t = a[l][e]; a[l][e] = 1; b[l] /= t;
  for (int i = 0; i < n; ++i) a[l][i] /= t;
  for (int i = 0; i < m; ++i) if (i != l && std::abs (
      a[i][e]) > EPS) {
   t = a[i][e]; a[i][e] = 0; b[i] -= t * b[l];
   for (int j = 0; j < n; ++j) a[i][j] -= t * a[l][j];
       }
  if (std::abs (c[e]) > EPS) {
   t = c[e]; c[e] = 0; v += t * b[l];
   for (int j = 0; j < n; ++j) c[j] -= t * a[l][j]; }
       }
 bool pre () {
  for (int l, e; ; ) {
   l = e = -1;
   for (int i = 0; i < m; ++i) if (b[i] < -EPS && (!~l
        || rand () & 1)) l = i;
   if (!~l) return false;
   for (int i = 0; i < n; ++i) if (a[l][i] < -EPS &&
       (!~e || rand () & 1)) e = i;
   if (!~e) return infeasible = true;
   pivot (l, e); } }
```

```
double solve () {
 double p; std::fill (q, q + n + m, -1);
 for (int i = 0; i < n; ++i) q[i] = i;
 v = 0; infeasible = unbounded = false;
 if (pre ()) return 0;
 for (int l, e; ; pivot (l, e)) {
  l = e = -1; for (int i = 0; i < n; ++i) if (c[i] > EPS) { e = i; break; }
  if (!~e) break; p = INF;
  for (int i = 0; i < m; ++i) if (a[i][e] > EPS && p > b[i] / a[i][e])
   p = b[i] / a[i][e], l = i;
  if (!~l) return unbounded = true, 0; }
 for (int i = n; i < n + m; ++i) if (~q[i]) an[q[i]] = b[i - n];
 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,

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

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) $M = \bigcup_{i=1}^{n} M_i\{x | x > 1 \land a_i\%x = 0\}$.
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, $M = \bigcup\{M_{x_1} \times M_{x_2} \cdots \times M_{x_l} | l \leq k \land \forall i(x_i < x_{i+1})\}$.
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 $0, 1, 2, ..., n-1$. $f_{1,m} = 0, f_{n,m} = (f_{n-1,m} + m) \mod n$.

5.5 Machine learning

5.5.1 Neural network

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

```
template <int ft = 3, int n = 2, int m = 3, int MAXDATA = 100000>
struct network {
 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];
 network () {
  std::mt19937_64 mt (time (0));
  std::uniform_real_distribution <double> urdn (0, 2 * sqrt (m));
  for (int i = 0; i < n; ++i) for (int j = 0; j < m; ++j) for (int k = 0; k < (i ? m : ft); ++k)
   wp[i][j][k] = urdn (mt);
  for (int i = 0; i < n; ++i) for (int j = 0; j < m; ++j) bp[i][j] = urdn (mt);
  for (int i = 0; i < m; ++i) w[i] = urdn (mt); b = urdn (mt);
  for (int i = 0; i < ft + 1; ++i) avg[i] = sig[i] = 0; }
 double compute (double *x) {
  for (int j = 0; j < m; ++j) {
   val[0][j] = bp[0][j]; for (int k = 0; k < ft; ++k) val[0][j] += wp[0][j][k] * x[k];
   val[0][j] = 1 / (1 + exp (-val[0][j]));
  }
  for (int i = 1; i < n; ++i) for (int j = 0; j < m; ++j) {
   val[i][j] = bp[i][j]; for (int k = 0; k < m; ++k) val[i][j] += wp[i][j][k] * val[i - 1][k];
   val[i][j] = 1 / (1 + exp (-val[i][j]));
  }
  double res = b; for (int i = 0; i < m; ++i) res += val[n - 1][i] * w[i];
//  return 1 / (1 + exp (-res));
  return res; }
 void desc (double *x, double t, double eta) {
  double o = compute (x), delo = (o - t); // * o * (1 - o)
  for (int j = 0; j < m; ++j) del[n - 1][j] = w[j] * delo * val[n - 1][j] * (1 - val[n - 1][j]);
  for (int i = n - 2; i >= 0; --i) for (int j = 0; j < m; ++j) {
   del[i][j] = 0; for (int k = 0; k < m; ++k)
    del[i][j] += wp[i + 1][k][j] * del[i + 1][k] * val[i][j] * (1 - val[i][j]);
  }
  for (int j = 0; j < m; ++j) bp[0][j] -= eta * del[0][j];
  for (int j = 0; j < m; ++j) for (int k = 0; k < ft; ++k) wp[0][j][k] -= eta * del[0][j] * x[k];
  for (int i = 1; i < n; ++i) for (int j = 0; j < m; ++j) bp[i][j] -= eta * del[i][j];
  for (int i = 1; i < n; ++i) for (int j = 0; j < m; ++j) for (int k = 0; k < m; ++k)
   wp[i][j][k] -= eta * del[i][j] * val[i - 1][k];
  b -= eta * delo;
//  for (int i = 0; i < m; ++i) w[i] -= eta * delo * o * (1 - o) * val[i];
  for (int i = 0; i < m; ++i) w[i] -= eta * delo * val[n - 1][i]; }
 void train (double data[MAXDATA][ft + 1], int dn, int epoch, double eta) {
  for (int i = 0; i < ft + 1; ++i) for (int j = 0; j < dn; ++j) avg[i] += data[j][i];
  for (int i = 0; i < ft + 1; ++i) avg[i] /= dn;
  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]);
  for (int i = 0; i < ft + 1; ++i) sig[i] = sqrt (sig[i] / dn);
  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];
  for (int cnt = 0; cnt < epoch; ++cnt) for (int test = 0; test < dn; ++test)
   desc (data[test], data[test][ft], eta); }
 double predict (double *x) {
  for (int i = 0; i < ft; ++i) x[i] = (x[i] - avg[i]) / sig[i];
  return compute (x) * sig[ft] + avg[ft]; }
 std::string to_string () {
```

```
  std::ostringstream os; os << std::fixed << std::setprecision (16);
  for (int i = 0; i < n; ++i) for (int j = 0; j < m; ++j) for (int k = 0; k < (i ? m : ft); ++k)
   os << wp[i][j][k] << "␣";
  for (int i = 0; i < n; ++i) for (int j = 0; j < m; ++j) os << bp[i][j] << "␣";
  for (int i = 0; i < m; ++i) os << w[i] << "␣"; os << b << "␣";
  for (int i = 0; i < ft + 1; ++i) os << avg[i] << "␣";
  for (int i = 0; i < ft + 1; ++i) os << sig[i] << "␣";
  return os.str (); }
 void read (const std::string &str) {
  std::istringstream is (str);
  for (int i = 0; i < n; ++i) for (int j = 0; j < m; ++j) for (int k = 0; k < (i ? m : ft); ++k)
   is >> wp[i][j][k];
  for (int i = 0; i < n; ++i) for (int j = 0; j < m; ++j) is >> bp[i][j];
  for (int i = 0; i < m; ++i) is >> w[i]; is >> b;
  for (int i = 0; i < ft + 1; ++i) is >> avg[i];
  for (int i = 0; i < ft + 1; ++i) is >> sig[i]; } };
```

5.6 Primality

5.6.1 Miller Rabin primality test

Test whether a certain integer is prime.

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

5.6.2 Pollard's Rho algorithm

Factorize an integer.

```
struct pollard_rho {
 miller_rabin is_prime; const long long thr = 13E9;
 long long factor (const long long &n, const long long &seed) {
  long long x = rand () % (n - 1) + 1, y = x;
  for (int head = 1, tail = 2; ; ) {
   x = mul_mod (x, x, n);
   x = (x + seed) % n;
   if (x == y) return n;
   long long ans = std::__gcd (std::abs (x - y), n);
   if (ans > 1 && ans < n) return ans;
   if (++head == tail) { y = x; tail <<= 1; } } }
 void search (const long long &n, std::vector <long long> &div) {
  if (n > 1)   {
   if (is_prime.solve (n)) div.push_back (n);
   else {
    long long fac = n;
    for (; fac >= n; fac = factor (n, rand () % (n - 1) + 1));
    search (n / fac, div); search (fac, div); } } }
 std::vector <long long> solve (const long long &n) {
  std::vector <long long> ans;
  if (n > thr) search (n, ans);
  else {
   long long rem = n;
   for (long long i = 2; i * i <= rem; ++i)
    while (!(rem % i)) { ans.push_back (i); rem /= i; }
   if (rem > 1) ans.push_back (rem); }
  return ans; } };
```

5.7 Recurrence relation

5.7.1 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. $a_i - a_{i-1} - a_{i-2} = 0$.

```
struct berlekamp-massey {
 struct poly { std::vector <int> a; poly() { a.clear() ; }
  poly (std::vector <int> &a) : a (a) {}
  int length () const { return a.size(); }
  poly move (int d) { std::vector <int> na (d, 0);
   na.insert (na.end (), a.begin (), a.end ());
   return poly (na); }
  int calc(std::vector <int> &d, int pos) { int ret = 0;
   for (int i = 0; i < (int) a.size (); ++i) {
    if ((ret += 1LL * d[pos - i] * a[i] % MOD) >= MOD) {
     ret -= MOD; } }
   return ret; }
  poly operator - (const poly &b) {
   std::vector <int> na (std::max (this -> length (), b.length ()));
   for (int i = 0; i < (int) na.size (); ++i) {
    int aa = i < this -> length () ? this -> a[i] : 0,
     bb = i < b.length () ? b.a[i] : 0;
    na[i] = (aa + MOD - bb) % MOD; }
   return poly (na); } };
 poly operator * (const int &c, const poly &p) {
  std::vector <int> na (p.length ());
  for (int i = 0; i < (int) na.size (); ++i) {
   na[i] = 1LL * c * p.a[i] % MOD; }
  return na; }
 std::vector <int> solve(vector<int> a) {
  int n = a.size (); poly s, b;
  s.a.push_back (1), b.a.push_back (1);
  for (int i = 0, j = -1, ld = 1; i < n; ++i) {
   int d = s.calc(a, i); if (d) {
    if ((s.length () - 1) * 2 <= i) {
     poly ob = b; b = s;
     s = s - 1LL * d * inverse (ld) % MOD * ob.move (i - j);
     j = i; ld = d;
    } else {
     s = s - 1LL * d * inverse (ld) % MOD * b.move (i - j); } } }
  return s.a; } };
```

5.7.2 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: $calc(3) = 5, calc(10007) = 959155122 \mod 10^9 + 7$.

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

5.8 Sequence manipulation

5.8.1 Discrete Fourier transform

Complexity $O(n \log n)$.

```
template <int MAXN = 1000000>
struct dft {
  typedef std::complex <double> complex;
  complex e[2][MAXN];
  int init (int n) {
   int len = 1;
   for (; len <= 2 * n; len <<= 1);
   for (int i = 0; i < len; ++i) {
    e[0][i] = complex (cos (2 * PI * i / len), sin (2 * PI * i / len));
    e[1][i] = complex (cos (2 * PI * i / len), -sin (2 * PI * i / len)); }
   return len; }
  void solve (complex *a, int n, int f) {
   for (int i = 0, j = 0; i < n; ++i) {
    if (i > j) std::swap (a[i], a[j]);
    for (int t = n >> 1; (j ^= t) < t; t >>= 1); }
   for (int i = 2; i <= n; i <<= 1)
    for (int j = 0; j < n; j += i)
     for (int k = 0; k < (i >> 1); ++k) {
      complex A = a[j + k];
      complex B = e[f][n / i * k] * a[j + k + (i >> 1)];
      a[j + k] = A + B;
```

```
      a[j + k + (i >> 1)] = A - B; }
  if (f == 1) {
  for (int i = 0; i < n; ++i) a[i] = complex (a[i].real () / n, a[i].imag ()); } } };
```

5.8.2 Fast Walsh-Hadamard transform

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

```
void fwt (int *a, int n, int w) {
 for (int i = 1; i < n; i <<= 1)
  for(int j = 0; j < n; j += i << 1) {
   for(int k = 0; k < i; ++k) {
    int x = a[j + k], y = a[i + j + k];
    if (w) {
    /* xor : a[j + k] = (x + y) / 2, a[i + j + k] = (x - y) / 2; and : a[j + k] = x - y; or : a[i + j + k] = y - x; */
    }else{
    /* xor : a[j + k] = x + y, a[i + j + k] = x - y; and : a[j + k] = x + y; or : a[i + j + k] = x + y; */
    } } } }
```

5.8.3 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.

```
template <int MAXN = 1000000>
struct ntt {
 int MOD[3] = {1045430273, 1051721729, 1053818881}, PRT[3] = {3, 6, 7};
 void solve (int *a, int n, int f = 0, int mod = 998244353, int prt = 3) {
  for (int i = 0, j = 0; i < n; ++i) {
   if (i > j) std::swap (a[i], a[j]);
   for (int t = n >> 1; (j ^= t) < t; t >>= 1); }
  for (int i = 2; i <= n; i <<= 1) {
   static int exp[MAXN]; exp[0] = 1;
   exp[1] = fpm (prt, (mod - 1) / i, mod);
   if (f == 1) exp[1] = fpm (exp[1], mod - 2, mod);
   for (int k = 2; k < (i >> 1); ++k)
    exp[k] = int (1ll * exp[k - 1] * exp[1] % mod); }
   for (int j = 0; j < n; j += i) {
    for (int k = 0; k < (i >> 1); ++k) {
     int &pA = a[j + k], &pB = a[j + k + (i >> 1)];
     int A = pA, B = int (1ll * pB * exp[k] % mod);
     pA = (A + B) % mod;
     pB = (A - B + mod) % mod; } } }
  if (f == 1) {
   int rev = fpm (n, mod - 2, mod);
   for (int i = 0; i < n; ++i) a[i] = int (1ll * a[i] * rev % mod); } }
 int crt (int *a, int mod) {
  static int inv[3][3];
  for (int i = 0; i < 3; ++i) for (int j = 0; j < 3; ++j)
   inv[i][j] = (int) inverse (MOD[i], MOD[j]);
  static int x[3];
  for (int i = 0; i < 3; ++i) { x[i] = a[i];
   for (int j = 0; j < i; ++j) {
    int t = (x[i] - x[j] + MOD[i]) % MOD[i];
    if (t < 0) t += MOD[i];
    x[i] = int (1LL * t * inv[j][i] % MOD[i]); } }
  int sum = 1, ret = x[0] % mod;
  for (int i = 1; i < 3; ++i) {
   sum = int (1LL * sum * MOD[i - 1] % mod);
   ret += int (1LL * x[i] * sum % mod);
   if (ret >= mod) ret -= mod; }
  return ret; } };
```

5.8.4 Polynomial operation

1. `inverse`: Find a polynomial b so that $a(x)b(x) \equiv 1 \mod x^n \mod mod$. 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 $b^2(x) \equiv a(x) \mod x^n \mod mod$. 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.

```
template <int MAXN = 1000000>
struct polynomial {
 ntt <MAXN> tr;
 void inverse (int *a, int *b, int n, int mod, int prt) {
  static int c[MAXN]; b[0] = ::inverse (a[0], mod); b[1] = 0;
  for (int m = 2, i; m <= n; m <<= 1) {
   std::copy (a, a + m, c);
   std::fill (b + m, b + m + m, 0); std::fill (c + m, c + m + m, 0);
   tr.solve (c, m + m, 0, mod, prt); tr.solve (b, m + m, 0, mod, prt);
   for (int i = 0; i < m + m; ++i) b[i] = 1LL * b[i] * (2 - 1LL * b[i] * c[i] % mod + mod) % mod;
   tr.solve (b, m + m, 1, mod, prt); std::fill (b + m, b + m + m, 0); } }
 void sqrt (int *a, int *b, int n, int mod, int prt) {
  static int d[MAXN], ib[MAXN]; b[0] = 1; b[1] = 0;
  int i2 = ::inverse (2, mod), m, i;
  for (int m = 2; m <= n; m <<= 1) {
   std::copy (a, a + m, d);
   std::fill (d + m, d + m + m, 0); std::fill (b + m, b + m + m, 0);
   tr.solve (d, m + m, 0, mod, prt); inverse (b, ib, m, mod, prt);
   tr.solve (ib, m + m, 0, mod, prt); tr.solve (b, m + m, 0, mod, prt);
   for (int i = 0; i < m + m; ++i) b[i] = (1LL * b[i] * i2 + 1LL * i2 * d[i] % mod * ib[i]) % mod;
   tr.solve (b, m + m, 1, mod, prt); std::fill (b + m, b + m + m, 0); } }
 void divide (int *a, int n, int *b, int m, int *d, int *r, int mod, int prt) {
  static int u[MAXN], v[MAXN]; while (!b[m - 1]) --m;
  int p = 1, t = n - m + 1; while (p < t << 1) p <<= 1;
  std::fill (u, u + p, 0); std::reverse_copy (b, b + m, u); inverse (u, v, p, mod, prt);
  std::fill (v + t, v + p, 0); tr.solve (v, p, 0, mod, prt); std::reverse_copy (a, a + n, u);
  std::fill (u + t, u + p, 0); tr.solve (u, p, 0, mod, prt);
  for (int i = 0; i < p; ++i) u[i] = 1LL * u[i] * v[i] % mod;
  tr.solve (u, p, 1, mod, prt); std::reverse (u, u + t); std::copy (u, u + t, d);
  for (p = 1; p < n; p <<= 1); std::fill (u + t, u + p, 0);
  tr.solve (u, p, 0, mod, prt); std::copy (b, b + m, v);
  std::fill (v + m, v + p, 0); tr.solve (v, p, 0, mod, prt);
  for (int i = 0; i < p; ++i) u[i] = 1LL * u[i] * v[i] % mod;
  tr.solve (u, p, 1, mod, prt);
  for (int i = 0; i < m; ++i) r[i] = (a[i] - u[i] + mod) % mod;
  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 ≥ 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 ith symbol of x is the same as the symbol at position ($i \bmod \mathit{length}(w)$) 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.

6.2 Matching

6.2.1 Minimal string rotation

Return the start index.

```
int min_rep (char *s, int l) {
 int i, j, k;
 i = 0; j = 1; k = 0;
 while (i < l && j < l) {
  k = 0; while (s[i + k] == s[j + k] && k < l) ++k;
  if (k == l) return i;
  if (s[i + k] > s[j + k])
   if (i + k + 1 > j) i = i + k + 1;
   else i = j + 1;
  else if (j + k + 1 > i) j = j + k + 1;
  else j = i + 1; }
 if (i < l) return i; else return j; }
```

6.3 Palindrome

6.3.1 Manacher

Odd palindromes only.

```
for (int i = 1, j = 0; i != (n << 1) - 1; ++i) {
 int p = i >> 1, q = i - p, r = ((j + 1) >> 1) + l[j] - 1;
 l[i] = r < q ? 0 : std::min (r - q + 1, l[(j << 1) - i]);
 while (p - l[i] != -1 && q + l[i] != n
  && s[p - l[i]] == s[q + l[i]]) l[i]++;
 if (q + l[i] - 1 > r) j = i;
 a += l[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.

```
template <int MAXN = 1000000, int MAXC = 26>
struct palindromic_tree {
 struct node {
  node *child[MAXC], *fail; int len;
  node (int len) : fail (NULL), len (len) {
   memset (child, NULL, sizeof (child)); }
 } node_pool[MAXN * 2], *tot_node;
 int size, text[MAXN];
 node *odd, *even, *last;
 node *match (node *now) {
  for (; text[size - now -> len - 1] != text[size]; now = now -> fail);
  return now; }
 bool extend (int token) {
  text[++size] = token; node *now = match (last);
  if (now -> child[token])
   return last = now -> child[token], false;
  last = now -> child[token] = new (tot_node++) node (now -> len + 2);
  if (now == odd) last -> fail = even;
  else {
   now = match (now -> fail);
   last -> fail = now -> child[token]; }
  return true; }
 void init() {
  text[size = 0] = -1; tot_node = node_pool;
  last = even = new (tot_node++) node (0); odd = new (tot_node++) node (-1);
  even -> fail = odd; }
 palindromic_tree () { init (); } };
```

6.4 Suffix

6.4.1 Suffix array

Usage:

1. `sa[i]`: The beginning position of the i-th smallest suffix.
2. `rk[i]`: The rank of the suffix beginning at position i.
3. `height[i]`: The longest common prefix of `sa[i]` and `sa[i - 1]`.

```
template <int MAXN = 1000000, int MAXC = 26>
struct suffix_array {
 int rk[MAXN], height[MAXN], sa[MAXN];
 int cmp (int *x, int a, int b, int d) {
  return x[a] == x[b] && x[a + d] == x[b + d]; }
 void doubling (int *a, int n) {
  static int sRank[MAXN], tmpA[MAXN], tmpB[MAXN];
  int m = MAXC, *x = tmpA, *y = tmpB;
  for (int i = 0; i < m; ++i) sRank[i] = 0;
  for (int i = 0; i < n; ++i) ++sRank[x[i] = a[i]];
  for (int i = 1; i < m; ++i) sRank[i] += sRank[i - 1];
  for (int i = n - 1; i >= 0; --i) sa[--sRank[x[i]]] = i;
  for (int d = 1, p = 0; p < n; m = p, d <<= 1) {
   p = 0; for (int i = n - d; i < n; ++i) y[p++] = i;
   for (int i = 0; i < n; ++i) if (sa[i] >= d) y[p++] = sa[i] - d;
   for (int i = 0; i < m; ++i) sRank[i] = 0;
   for (int i = 0; i < n; ++i) ++sRank[x[i]];
   for (int i = 1; i < m; ++i) sRank[i] += sRank[i - 1];
   for (int i = n - 1; i >= 0; --i) sa[--sRank[x[y[i]]]] = y[i];
   std::swap (x, y); x[sa[0]] = 0; p = 1; y[n] = -1;
   for (int i = 1; i < n; ++i)
    x[sa[i]] = cmp (y, sa[i], sa[i - 1], d) ? p - 1 : p++; } }
 void solve (int *a, int n) {
  a[n] = -1; doubling (a, n);
  for (int i = 0; i < n; ++i) rk[sa[i]] = i;
  int cur = 0;
  for (int i = 0; i < n; ++i)
   if (rk[i]) {
    if (cur) cur--;
    for (; a[i + cur] == a[sa[rk[i] - 1] + cur]; ++cur);
    height[rk[i]] = cur; } } };
```

6.4.2 Suffix array (SAIS)

Ensure that `str[n]` ≥ 0 is the unique lexicographically smallest character in `str`.

Note that `sa[0]=n`.

```
template <int MAXN = 100000>
struct SA {
 int sa[MAXN], rk[MAXN], ht[MAXN], s[MAXN << 1], t[MAXN << 1], p[MAXN], cnt[MAXN], cur[MAXN];
#define pushS(x) sa[cur[s[x]]--] = x
#define pushL(x) sa[cur[s[x]]++] = x
#define inducedSort(v) std::fill_n(sa, n, -1); std::fill_n(cnt, m, 0);\
 for (int i = 0; i < n; i++) cnt[s[i]]++;\
 for (int i = 1; i < m; i++) cnt[i] += cnt[i-1];\
 for (int i = 0; i < m; i++) cur[i] = cnt[i]-1;\
 for (int i = n1-1; ~i; i--) pushS(v[i]);\
 for (int i = 1; i < m; i++) cur[i] = cnt[i-1];\
 for (int i = 0; i < n; i++) if (sa[i] > 0 &&  t[sa[i]-1]) pushL(sa[i]-1);\
 for (int i = 0; i < m; i++) cur[i] = cnt[i]-1;\
 for (int i = n-1;  ~i; i--) if (sa[i] > 0 && !t[sa[i]-1]) pushS(sa[i]-1)
 void sais(int n, int m, int *s, int *t, int *p) {
  int n1 = t[n-1] = 0, ch = rk[0] = -1, *s1 = s+n;
  for (int i = n-2; ~i; i--) t[i] = s[i] == s[i+1] ? t[i+1] : s[i] > s[i+1];
  for (int i = 1; i < n; i++) rk[i] = t[i-1] && !t[i] ? (p[n1] = i, n1++) : -1;
  inducedSort(p);
  for (int i = 0, x, y; i < n; i++) if (~(x = rk[sa[i]])) {
   if (ch < 1 || p[x+1] - p[x] != p[y+1] - p[y]) ch++;
   else for (int j = p[x], k = p[y]; j <= p[x+1]; j++, k++)
    if ((s[j]<<1|t[j]) != (s[k]<<1|t[k])) {ch++; break;}
   s1[y = x] = ch; }
  if (ch+1 < n1) sais(n1, ch+1, s1, t+n, p+n1);
  else for (int i = 0; i < n1; i++) sa[s1[i]] = i;
  for (int i = 0; i < n1; i++) s1[i] = p[sa[i]];
  inducedSort(s1); }
 template<typename T> int mapCharToInt(int n, const T *str) {
  int m = *std::max_element(str, str+n);
  std::fill_n(rk, m+1, 0);
  for (int i = 0; i < n; i++) rk[str[i]] = 1;
  for (int i = 0; i < m; i++) rk[i+1] += rk[i];
  for (int i = 0; i < n; i++) s[i] = rk[str[i]] - 1;
  return rk[m]; }
 template<typename T> void suffixArray(int n, const T *str) {
  int m = mapCharToInt(++n, str);
  sais(n, m, s, t, p);
  for (int i = 0; i < n; i++) rk[sa[i]] = i;
  for (int i = 0, h = ht[0] = 0; i < n-1; i++) {
   int j = sa[rk[i]-1];
   while (i+h < n && j+h < n && s[i+h] == s[j+h]) h++;
   if (ht[rk[i]] = h) h--; } } };
```

6.4.3 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. `state::len`: The longest length of the string in the state.
4. `state::right - 1`: The first location in the string where the state can be reached
5. `state::parent`: the parent link.
6. `state::dest`: the automaton link.

```
template <int MAXN = 1000000, int MAXC = 26>
struct suffix_automaton {
 struct state {
  int len, right; state *parent, *dest[MAXC];
  state (int len = 0, int right = 0) : len (len), right (right), parent (NULL) {
   memset (dest, 0, sizeof (dest)); }
 } node_pool[MAXN * 2], *tot_node, *null = new state();
 state *head, *tail;
 void extend (int token) {
  state *p = tail;
  state *np = tail -> dest[token] ? null : new (tot_node++) state (tail -> len + 1, tail -> len + 1);
  while (p && !p -> dest[token]) p -> dest[token] = np, p = p -> parent;
  if (!p) np -> parent = head;
  else {
   state *q = p -> dest[token];
   if (p -> len + 1 == q -> len) {
    np -> parent = q;
   } else {
    state *nq = new (tot_node++) state (*q);
    nq -> len = p -> len + 1;
    np -> parent = q -> parent = nq;
    while (p && p -> dest[token] == q) {
     p -> dest[token] = nq, p = p -> parent;
    } } }
  tail = np == null ? np -> parent : np; }
 void init () {
  tot_node = node_pool;
  head = tail = new (tot_node++) state (); }
 suffix_automaton () { init (); } };
```

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_clrsb`: 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 Fast IO

```
#define __ __attribute__ ((optimize ("-O3")))
#define _ __ __inline __attribute__ ((__gnu_inline__, __always_inline__, __artificial__))
namespace io {
 const int SIZE = 1000000; static char buf[SIZE + 1], *p = buf + SIZE;
 template <class t> _ bool read_int (t &x) {
  register int f = 0, sgn = 0; x = 0;
  while ((*p || (p = buf, buf[fread (buf, 1, SIZE, stdin)] = 0, buf[0])) &&
     (isdigit (*p) && (x = x * 10 + (*p - '0'), f = 1) || !f && (*p != '-' || (sgn = 1)))) ++p;
  if (sgn) x = -x;
  return f; }
 _ int read_str (char *x, int len, char d = '\n') {
  register int cnt = 0;
  while ((*p || (p = buf, buf[fread (buf, 1, SIZE, stdin)] = 0, buf[0])) &&
   cnt < len && *p != d) *(x++) = *(p++), ++cnt;
  if (*p == d) ++p;
  return cnt; }
//Set f to true to force an output (typically at the last write command).
 const int WSIZE = 1000000; static char wbuf[2 * WSIZE], *q = wbuf;
 __ void write (bool f, const char *str, ...) {
  va_list args; va_start (args, str);
  if ((q += vsprintf (q, str, args)) - wbuf >= WSIZE || f) fwrite (wbuf, 1, q - wbuf, stdout), q = wbuf;
  va_end (args); } }
```

7.3 Formatting

Faster cin and cout.

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

Examples on IO functions.

```
std::string str;
std::getline (std::cin, str, '#');
char ch[100];
std::cin.getline (ch, 100, '#');
fgets (ch, 100, stdin);
int c = std::cin.peek ();
std::cin.ignore (100, '#');
std::cin.ignore (100, EOF);
std::cin.seekg (0, std::cin.end);
int length = std::cin.tellg ();
std::cin.seekg (0, std::cin.beg);
char *buf = new char[length];
std::cin.read (buf, length);
std::cout << std::setw (10);
std::cout << std::setfill ('#');
std::cout << std::left << x << "\n";
std::cout << std::internal << x << "\n";
std::cout << std::right << x << "\n";
std::cout << std::setprecision (10);
std::cout << std::fixed; // std::cout << std::scientific;
```

7.4 Java

Import Libraries that are commonly used.

```
import java.io.*;
import java.lang.*;
import java.math.*;
import java.util.*;
```

Input `Scanner` is generally used to handle input.

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

Or:

```
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.

```
System.out.print (/*...*/);
System.out.println (/*...*/);
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:

```
public static BigInteger sqrt (BigInteger x) {
 if (x.equals (BigInteger.ZERO) || x.equals (BigInteger.ONE)) return x;
 BigInteger d = BigInteger.ZERO.setBit (x.bitLength () / 2);
 BigInteger d2 = d;
 for (; ; ) {
  BigInteger y = d.add (x.divide (d)).shiftRight (1);
  if (y.equals (d) || y.equals (d2)) return d.min (d2);
  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, Comperator <? super T> comperator);`

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.

```java
public class Main {
 public class Point {
  public int x; public int y;
  public Point () {
   x = 0;
   y = 0; }
  public Point (int xx, int yy) {
   x = xx;
   y = yy; } }
 public class Cmp implements Comparator <Point> {
  public int compare (Point a, Point b) {
   if (a.x < b.x) return -1;
   if (a.x == b.x) {
    if (a.y < b.y) return -1;
    if (a.y == b.y) return 0; }
   return 1; } }
 public static void main (String [] args) {
  Cmp c = new Cmp ();
  TreeMap <Point, Point> t = new TreeMap <Point, Point> (c);
  return; } }
```

Comparable An example to implement `Comparable`.

```java
public class Point implements Comparable <Point> {
 public int x; public int y;
 public Point () {
  x = 0;
  y = 0; }
 public Point (int xx, int yy) {
  x = xx;
  y = yy; }
 public int compareTo (Point p) {
  if (x < p.x) return -1;
  if (x == p.x) {
   if (y < p.y) return -1;
   if (y == p.y) return 0; }
  return 1; }
 public boolean equalTo (Point p) {
  return (x == p.x && y == p.y); }
 public int hashCode () {
  return x + y; } }
```

Fast IO A class for faster IO.

```java
public class Main {
 static class InputReader {
  public BufferedReader reader;
  public StringTokenizer tokenizer;
  public InputReader (InputStream stream) {
   reader = new BufferedReader (new InputStreamReader (stream), 32768);
   tokenizer = null; }
  public String next() {
   while (tokenizer == null || !tokenizer.hasMoreTokens()) {
    try {
     String line = reader.readLine();
     tokenizer = new StringTokenizer (line);
    } catch (IOException e) {
     throw new RuntimeException (e); } }
   return tokenizer.nextToken(); }
  public BigInteger nextBigInteger() {
   return new BigInteger (next (), 10); /* radix */ }
  public int nextInt() {
   return Integer.parseInt (next()); }
  public double nextDouble() {
   return Double.parseDouble (next()); } }
 public static void main (String[] args) {
  InputReader in = new InputReader (System.in);
 } }
```

7.5 Random numbers

An example on the usage of `generator` and `distribution`.

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

7.6 Regular expression

This is an example to construct a pattern:

```cpp
std::string str = ("The␣the␣there");
std::regex pattern ("(th|Th)[\\w]*", std::regex_constants::optimize | std::regex_constants::ECMAScript);
std::smatch match; //std::cmatch for char *
```

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

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

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

```cpp
auto mbegin = std::sregex_iterator (str.begin (), str.end (), pattern);
auto mend = std::sregex_iterator ();
std::cout << "Found␣" << std::distance (mbegin, mend) << "␣words:\n";
for (std::sregex_iterator i = mbegin; i != mend; ++i) {
 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`.

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

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

7.7 Stack hack

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

```cpp
//C++
#pragma comment (linker, "/STACK:36777216")
//G++
int __size__ = 256 << 20;
char *__p__ = (char*) malloc(__size__) + __size__;
__asm__ ("movl␣%0,␣%%esp\n" :: "r"(__p__));
```

7.8 Time hack

The following lines allow the program to check current time.

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

8 Appendix

8.1 Table of formulae

Binomial Coefficients

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

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

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

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

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

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

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

Fibonacci Numbers

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

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

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

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

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

$$f_1 + 2f_2 + 3f_3 + \cdots + nf_n = nf_{n+2} - f_{n+3} + 2]$$

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

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

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 $n=p_1^{k_1}...p_m^{k_m}$: $G(n)=lcm(G(p_1^{k_1}),...,G(p_m^{k_m}))$.

Lucas Numbers

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

Catlan Numbers

$$c_1=1,c_n=\sum_{i=0}^{n-1}c_ic_{n-1-i}=c_{n-1}\frac{4n-2}{n+1}=\frac{\binom{2n}{n}}{n+1}$$
$$=\binom{2n}{n}-\binom{2n}{n-1},c(x)=\frac{1-\sqrt{1-4x}}{2x}$$

Stirling Cycle Numbers

Divide n elements into k non-empty cycles.

$$s(n,0)=0,s(n,n)=1,s(n+1,k)=s(n,k-1)-ns(n,k)$$
$$s(n,k)=(-1)^{n-k}\begin{bmatrix}n\\k\end{bmatrix}$$
$$\begin{bmatrix}n+1\\k\end{bmatrix}=n\begin{bmatrix}n\\k\end{bmatrix}+\begin{bmatrix}n\\k-1\end{bmatrix},\begin{bmatrix}n+1\\2\end{bmatrix}=n!H_n$$
$$x^{\underline{n}}=x(x-1)...(x-n+1)=\sum_{k=0}^{n}\begin{bmatrix}n\\k\end{bmatrix}(-1)^{n-k}x^k$$
$$x^{\overline{n}}=x(x+1)...(x+n-1)=\sum_{k=0}^{n}\begin{bmatrix}n\\k\end{bmatrix}x^k$$

Stirling Subset Numbers

Divide n elements into k non-empty subsets.

$$\begin{Bmatrix}n+1\\k\end{Bmatrix}=k\begin{Bmatrix}n\\k\end{Bmatrix}+\begin{Bmatrix}n\\k-1\end{Bmatrix}$$
$$x^n=\sum_{k=0}^{n}\begin{Bmatrix}n\\k\end{Bmatrix}x^{\underline{k}}=\sum_{k=0}^{n}\begin{Bmatrix}n\\k\end{Bmatrix}(-1)^{n-k}x^{\overline{k}}$$
$$m!\begin{Bmatrix}n\\m\end{Bmatrix}=\sum_{k=0}^{m}\binom{m}{k}k^n(-1)^{m-k}$$
$$\sum_{k=1}^{n}k^p=\sum_{k=0}^{p}\begin{Bmatrix}p\\k\end{Bmatrix}(n+1)^{\underline{k}}$$

For a fixed k, generating functions :

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

Motzkin Numbers

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

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

$$M_{n+1}=M_n+\sum_{i}^{n-1}M_iM_{n-1-i}=\frac{(2n+3)M_n+3nM_{n-1}}{n+3}$$
$$M_n=\sum_{i=0}^{\lfloor\frac{n}{2}\rfloor}\binom{n}{2k}Catlan(k)$$
$$M(X)=\frac{1-x-\sqrt{1-2x-3x^2}}{2x^2}$$

Eulerian Numbers

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

$$\left\langle {n \atop k} \right\rangle=(k+1)\left\langle {n-1 \atop k} \right\rangle+(n-k)\left\langle {n-1 \atop k-1} \right\rangle$$
$$x^n=\sum_k\left\langle {n \atop k} \right\rangle\binom{x+k}{n}$$
$$\left\langle {n \atop m} \right\rangle=\sum_{k=0}^{m}\binom{n+1}{k}(m+1-k)^n(-1)^k$$

Harmonic Numbers

Sum of the reciprocals of the first n natural numbers.

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

Pentagonal Number Theorem

$$\prod_{n=1}^{\infty}(1-x^n)=\sum_{n=-\infty}^{\infty}(-1)^kx^{k(3k-1)/2}$$
$$p(n)=p(n-1)+p(n-2)-p(n-5)-p(n-7)+\cdots$$
$$f(n,k)=p(n)-p(n-k)-p(n-2k)+p(n-5k)+p(n-7k)-\cdots$$

Bell Numbers

Divide a set that has exactly n elements.

$$B_n=\sum_{k=1}^{n}\begin{Bmatrix}n\\k\end{Bmatrix},\quad B_{n+1}=\sum_{k=0}^{n}\binom{n}{k}B_k$$
$$B_{p^m+n}\equiv mB_n+B_{n+1}\pmod p$$
$$B(x)=\sum_{n=0}^{\infty}\frac{B_n}{n!}x^n=e^{e^x-1}$$

Bernoulli Numbers

$$B_n=1-\sum_{k=0}^{n-1}\binom{n}{k}\frac{B_k}{n-k+1}$$
$$G(x)=\sum_{k=0}^{\infty}\frac{B_k}{k!}x^k=\frac{1}{\sum_{k=0}^{\infty}\frac{x^k}{(k+1)!}}$$
$$\sum_{k=1}^{n}k^m=\frac{1}{m+1}\sum_{k=0}^{m}\binom{m+1}{k}B_kn^{m-k+1}$$

Sum of Powers

$$\sum_{i=1}^{n}i^2=\frac{n(n+1)(2n+1)}{6},\quad\sum_{i=1}^{n}i^3=(\frac{n(n+1)}{2})^2$$
$$\sum_{i=1}^{n}i^4=\frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$
$$\sum_{i=1}^{n}i^5=\frac{n^2(n+1)^2(2n^2+2n-1)}{12}$$

Sum of Squares

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

$$n=2^{a_0}p_1^{2a_1}\cdots p_r^{2a_r}q_1b_1\cdots q_sb_s$$

where $p_i\equiv 3 \mod 4$, $q_i\equiv 1 \mod 4$, then

$$r_2(n)=\begin{cases}0 & \text{if any } a_i \text{ is a half-integer}\\ 4\prod_1^r(b_i+1) & \text{if all } a_i \text{ are integers}\end{cases}$$

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

Derangement

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

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

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

$$\iint_\Sigma f(x,y,z)\mathrm{d}S=\iint_D f(x(u,v),y(u,v),z(u,v))\sqrt{EG-F^2}\mathrm{d}u\mathrm{d}v\,,$$

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

$$\int_L P(x,y,z)\mathrm{d}x + Q(x,y,z)\mathrm{d}y + R(x,y,z)\mathrm{d}z$$
$$= \int_a^b [P(x(t),y(t),z(t))x'(t) + Q(x(t),y(t),z(t))y'(t) +$$
$$R(x(t),y(t),z(t))z'(t)]\mathrm{d}t$$

$$\iint_L P(x,y,z)\mathrm{d}y\mathrm{d}z + Q(x,y,z)\mathrm{d}z\mathrm{d}x + R(x,y,z)\mathrm{d}x\mathrm{d}y$$
$$= \pm\iint_D [P(x(u,v),y(u,v),z(u,v))\frac{\partial(y,z)}{\partial(u,v)} +$$
$$Q(x(u,v),y(u,v),z(u,v))\frac{\partial(z,x)}{\partial(u,v)} +$$
$$R(x(u,v),y(u,v),z(u,v))\frac{\partial(x,y)}{\partial(u,v)}]\mathrm{d}u\mathrm{d}v$$

Variable substitution

$$\iint_{T(D)} f(x,y)\mathrm{d}x\mathrm{d}y = \iint_D f(x(u,v),y(u,v))\left|\frac{\partial(x,y)}{\partial(u,v)}\right|\mathrm{d}u\mathrm{d}v$$

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

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

Integration

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

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

$\sqrt{ax+b}$

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

$x^2 \pm a^2$

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

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

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

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

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

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

1. $\int \frac{\mathrm{d}x}{\sqrt{x^2+a^2}} = \operatorname{arsh}\frac{x}{a} + C_1 = \ln(x + \sqrt{x^2+a^2}) + C$
2. $\int \frac{\mathrm{d}x}{\sqrt{(x^2+a^2)^3}} = \frac{x}{a^2\sqrt{x^2+a^2}} + C$
3. $\int \frac{x}{\sqrt{x^2+a^2}}\mathrm{d}x = \sqrt{x^2+a^2} + C$
4. $\int \frac{x}{\sqrt{(x^2+a^2)^3}}\mathrm{d}x = -\frac{1}{\sqrt{x^2+a^2}} + C$
5. $\int \frac{x^2}{\sqrt{x^2+a^2}}\mathrm{d}x = \frac{x}{2}\sqrt{x^2+a^2} - \frac{a^2}{2}\ln(x+\sqrt{x^2+a^2}) + C$
6. $\int \frac{x^2}{\sqrt{(x^2+a^2)^3}}\mathrm{d}x = -\frac{x}{\sqrt{x^2+a^2}} + \ln(x+\sqrt{x^2+a^2}) + C$
7. $\int \frac{\mathrm{d}x}{x\sqrt{x^2+a^2}} = \frac{1}{a}\ln\frac{\sqrt{x^2+a^2}-a}{|x|} + C$
8. $\int \frac{\mathrm{d}x}{x^2\sqrt{x^2+a^2}} = -\frac{\sqrt{x^2+a^2}}{a^2x} + C$
9. $\int \sqrt{x^2+a^2}\mathrm{d}x = \frac{x}{2}\sqrt{x^2+a^2} + \frac{a^2}{2}\ln(x+\sqrt{x^2+a^2}) + C$
10. $\int \sqrt{(x^2+a^2)^3}\mathrm{d}x = \frac{x}{8}(2x^2+5a^2)\sqrt{x^2+a^2} + \frac{3}{8}a^4\ln(x+\sqrt{x^2+a^2}) + C$
11. $\int x\sqrt{x^2+a^2}\mathrm{d}x = \frac{1}{3}\sqrt{(x^2+a^2)^3} + C$
12. $\int x^2\sqrt{x^2+a^2}\mathrm{d}x = \frac{x}{8}(2x^2+a^2)\sqrt{x^2+a^2} - \frac{a^4}{8}\ln(x+\sqrt{x^2+a^2}) + C$
13. $\int \frac{\sqrt{x^2+a^2}}{x}\mathrm{d}x = \sqrt{x^2+a^2} + a\ln\frac{\sqrt{x^2+a^2}-a}{|x|} + C$
14. $\int \frac{\sqrt{x^2+a^2}}{x^2}\mathrm{d}x = -\frac{\sqrt{x^2+a^2}}{x} + \ln(x+\sqrt{x^2+a^2}) + C$

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

1. $\int \frac{\mathrm{d}x}{\sqrt{x^2-a^2}} = \frac{x}{|x|}\operatorname{arch}\frac{|x|}{a} + C_1 = \ln\left|x+\sqrt{x^2-a^2}\right| + C$
2. $\int \frac{\mathrm{d}x}{\sqrt{(x^2-a^2)^3}} = -\frac{x}{a^2\sqrt{x^2-a^2}} + C$
3. $\int \frac{x}{\sqrt{x^2-a^2}}\mathrm{d}x = \sqrt{x^2-a^2} + C$
4. $\int \frac{x}{\sqrt{(x^2-a^2)^3}}\mathrm{d}x = -\frac{1}{\sqrt{x^2-a^2}} + C$
5. $\int \frac{x^2}{\sqrt{x^2-a^2}}\mathrm{d}x = \frac{x}{2}\sqrt{x^2-a^2} + \frac{a^2}{2}\ln|x+\sqrt{x^2-a^2}| + C$
6. $\int \frac{x^2}{\sqrt{(x^2-a^2)^3}}\mathrm{d}x = -\frac{x}{\sqrt{x^2-a^2}} + \ln|x+\sqrt{x^2-a^2}| + C$
7. $\int \frac{\mathrm{d}x}{x\sqrt{x^2-a^2}} = \frac{1}{a}\arccos\frac{a}{|x|} + C$
8. $\int \frac{\mathrm{d}x}{x^2\sqrt{x^2-a^2}} = \frac{\sqrt{x^2-a^2}}{a^2x} + C$
9. $\int \sqrt{x^2-a^2}\mathrm{d}x = \frac{x}{2}\sqrt{x^2-a^2} - \frac{a^2}{2}\ln|x+\sqrt{x^2-a^2}| + C$
10. $\int \sqrt{(x^2-a^2)^3}\mathrm{d}x = \frac{x}{8}(2x^2-5a^2)\sqrt{x^2-a^2} + \frac{3}{8}a^4\ln|x+\sqrt{x^2-a^2}| + C$
11. $\int x\sqrt{x^2-a^2}\mathrm{d}x = \frac{1}{3}\sqrt{(x^2-a^2)^3} + C$
12. $\int x^2\sqrt{x^2-a^2}\mathrm{d}x = \frac{x}{8}(2x^2-a^2)\sqrt{x^2-a^2} - \frac{a^4}{8}\ln|x+\sqrt{x^2-a^2}| + C$
13. $\int \frac{\sqrt{x^2-a^2}}{x}\mathrm{d}x = \sqrt{x^2-a^2} - a\arccos\frac{a}{|x|} + C$
14. $\int \frac{\sqrt{x^2-a^2}}{x^2}\mathrm{d}x = -\frac{\sqrt{x^2-a^2}}{x} + \ln|x+\sqrt{x^2-a^2}| + C$

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

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

11. $\int x\sqrt{a^2-x^2}\mathrm{d}x = -\frac{1}{3}\sqrt{(a^2-x^2)^3} + C$
12. $\int x^2\sqrt{a^2-x^2}\mathrm{d}x = \frac{x}{8}(2x^2-a^2)\sqrt{a^2-x^2} + \frac{a^4}{8}\arcsin\frac{x}{a} + C$
13. $\int \frac{\sqrt{a^2-x^2}}{x}\mathrm{d}x = \sqrt{a^2-x^2} + a\ln\frac{a-\sqrt{a^2-x^2}}{|x|} + C$
14. $\int \frac{\sqrt{a^2-x^2}}{x^2}\mathrm{d}x = -\frac{\sqrt{a^2-x^2}}{x} - \arcsin\frac{x}{a} + C$

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

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

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

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

Triangular function

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

Inverse triangular function $(a>0)$

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

Exponential function

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

Logarithmic function

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

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

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