Contents


```
vim.o.shiftwidth=4
vim.o.expandtab=true
vim.o.tabstop=4
vim.o.relativenumber=true
vim.cmd("colorscheme habamax")
```

1 Data Structure

1.1 Splay

```
struct node_t {
  node_t();
  void update();
  int dir() { return (this == p->ch[1]); }
  void setc(node_t *c, int d) { ch[d] = c, c->p = this; }
  node_t *p, *ch[2];
  int size, cnt; // maintain tag from top to bottom (via find).
} s[maxn], *nil = s, *root;
node_t::node_t() { p = ch[0] = ch[1] = nil; }

void node_t::update() {
  if (this == nil) return;
  size = ch[0]->size + ch[1]->size + cnt;
}

node_t *newNode(int cnt) {
  ++pt;
  s[pt].cnt = cnt; s[pt].p = s[pt].ch[0] = s[pt].ch[1] = nil;
  s[pt].update();
  return &s[pt];
}

void rotate(node_t *t) {
  node_t *p = t->p;
  p->p->update();
  p->update();
  t->update();
  int d = t->dir();
  p->p->setc(t, p->dir());
  p->setc(t->ch[!d], d);
  t->setc(p, !d);
  if (p == root) root = t;
  p->update(), t->update();
}

node_t *splay(node_t *t, node_t *dst = nil) {
  while (t->p != dst) {
    if (t->p->p == dst) rotate(t);
    else if (t->dir() == t->p->dir()) rotate(t->p), rotate(t);
    else rotate(t), rotate(t);
  }
  t->update();
  return t;
}

node_t *prev(node_t *p) {
  splay(p);
  p = p->ch[0], p->update();
  while (p->ch[1] != nil) p = p->ch[1], p->update();
  return p;
}

node_t *succ(node_t *p) {
  splay(p);
  p = p->ch[1], p->update();
  while (p->ch[0] != nil) p = p->ch[0], p->update();
  return p;
}

void insert(node_t *y, node_t *x) { // Insert node x after y
  splay(y);
  if (y->ch[1] == nil) {
    y->ch[1] = x;
    x->p = y;
    y->update();
  } else {
    y = y->ch[1], y->update();
    while (y->ch[0] != nil) y = y->ch[0], y->update();
    y->ch[0] = x;
    x->p = y;
    y->update();
  }
  splay(x);
}

void removeAll(node_t *x) { // Remove all the whole subtree of x
  x->p->update();
  x->p->ch[x->dir()] = nil;
  x->p->update();
  splay(x->p);
  x->p = nil;
}

void remove(node_t *x) { // Remove the single node x
  node_t *p = prev(x); node_t *s = succ(x);
  splay(p);
  splay(s, p);
  removeAll(s->ch[0]);
}

node_t *find(node_t *t, int k) {
  t->update();
  if (t->ch[0]->size < k && t->ch[0]->size + t->cnt >= k) return t;
```

```
  if (t->ch[0]->size >= k) return find(t->ch[0], k);
  return find(t->ch[1], k - t->ch[0]->size - t->cnt);
}

node_t *findAndSplit(node_t *t, int k) {
  t->update();
  if (t->ch[0]->size < k && t->ch[0]->size + t->cnt >= k) {
    int cnt = t->cnt;
    k -= t->ch[0]->size;
    t->cnt = 1;
    splay(t);
    node_t *p = prev(t);
    if (k - 1) insert(p, newNode(k - 1));
    if (cnt - k) insert(t, newNode(cnt - k));
    return t;
  }
  if (t->ch[0]->size >= k) return findAndSplit(t->ch[0], k);
  return findAndSplit(t->ch[1], k - t->ch[0]->size - t->cnt);
}

void init() {
  pt = 0, nil->p = nil->ch[0] = nil->ch[1] = nil;
}

node_t *expose(node_t *x, node_t *y) {
  x = prev(x), y = succ(y);
  return splay(y, splay(x))->ch[0];
}
```

1.2 Link-Cut Tree

```
struct node_t {
  node_t();
  node_t *ch[2], *p;
  int size, root;
  int dir() { return this == p->ch[1]; }
  void setc(node_t *c, int d) { ch[d] = c, c->p = this; }
  void update() { size = ch[0]->size + ch[1]->size + 1; }
} s[maxn], *nil = s;

node_t::node_t() {
  size = 1, root = true;
  ch[0] = ch[1] = p = nil;
}

void rotate(node_t *t) {
  node_t *p = t->p;
  int d = t->dir();
  if (!p->root) {
    p->p->setc(t, p->dir());
  } else {
    p->root = false, t->root = true;
    t->p = p->p; // Path Parent
  }
  p->setc(t->ch[!d], d);
  t->setc(p, !d);
  p->update(), t->update();
}

void splay(node_t *t) {
  // t->update(); // tag!
  while (!t->root) {
    // if (!t->p->root) t->p->p->update(); t->p->update(), t->update(); // !
    if (!t->p->root) rotate(t->dir() == t->p->dir() ? t->p : t);
    rotate(t);
  }
}

void access(node_t *x) { // Ask u, v: access(u), access(v, true), x = LCA
  node_t *y = nil;
  while (x != nil) {
    splay(x);
    // if (x->p == nil) at second call, x->ch[1](rev) + (x)_single + y
    x->ch[1]->root = true;
    x->ch[1] = y, y->root = false;
    x->update();
    y = x, x = x->p;
  }
}

void cut(node_t *x) {
  access(x);
  splay(x);
  x->ch[0]->root = true;
  x->ch[0]->p = nil;
  x->ch[0] = nil;
}

void link(node_t *x, node_t *y) {
  access(y);
  splay(y);
  y->p = x;
  access(y);
}

void init() { nil->size = 0; }
```

1.3 Treap

```
mt19937 rng{583427}; // any seed, crucial when testing on windows
struct item {
    int key, prior;
    item *l, *r;
    item () {}
    item (int key) : key(key), prior(rng()), l(nullptr), r(nullptr) { }
    item (int key, int prior) : key(key), prior(prior), l(nullptr), r(nullptr) { }
};
typedef item* pitem;

int cnt (pitem t) {
    return t ? t->cnt : 0;
}

void push (pitem p) {} // just in case

void pull (pitem t) {
    if (t)
        t->cnt = 1 + cnt(t->l) + cnt (t->r);
}

void split (pitem t, pitem & l, pitem & r, int key) {
    if (!t)
        return void( l = r = 0 );
    push(t);
    int lsize = cnt(t->l); //implicit key
    if (key <= lsize)
        split (t->l, l, t->l, key),  r = t;
    else
        split (t->r, t->r, r, key - 1 - lsize),  l = t;
    pull(t);
}

void merge (pitem & t, pitem l, pitem r) {
    push(l);
    push(r);
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge(l->r, l->r, r),  t = l;
    else
        merge(r->l, l, r->l),  t = r;
    pull(t);
}

pitem unite (pitem l, pitem r) {
    if (!l || !r)  return l ? l : r;
    push(l);
    push(r);
    if (l->prior < r->prior) swap(l, r);
    pitem lt, rt;
    split(r, l->key, lt, rt);
    l->l = unite(l->l, lt);
    l->r = unite(l->r, rt);
  pull(l);
    return l;
}
```

2 String Algorithms

2.1 Common

```
// please note that all strings are indexed from 0
void kmp(const char *s, int *next) {
  --s, --next;
  next[1] = 0;
  int j = 0, n = strlen(s + 1);
  for (int i = 2; i <= n; ++i) {
    while (j > 0 && s[j + 1] != s[i]) j = next[j];
    if (s[j + 1] == s[i]) j = j + 1;
    next[i] = j;
  }
}

// s: text, t: text being searched, ex[i]: maximum l satisfying s[i...i+l-1] = t
  [0...l-1]
void exkmp(const char *s, const char *t, int *next, int *ex) {
  int n = strlen(t), m = strlen(s), k, c;
  next[0] = n;
  k = 0, c = 1;
  while (k + 1 < n && t[k] == t[k + 1]) ++k;
  next[1] = k;
  for (int i = 2; i < n; ++i) {
    int p = next[c] + c - 1;
    int l = next[i - c];
    if (i + l < p + 1) next[i] = l;
    else {
      k = max(0, p - i + 1);
      while (i + k < n && t[i + k] == t[k]) ++k;
      next[i] = k;
      c = i;
    }
  }
  k = c = 0;
```

```
  while (k < m && k < n && s[k] == t[k]) ++k;
  ex[0] = k;
  for (int i = 1; i < m; ++i) {
    int p = ex[c] + c - 1;
    int l = next[i - c];
    if (l + i < p + 1) ex[i] = next[i - c];
    else {
      k = max(0, p - i + 1);
      while (i + k < m && k < n && s[i + k] == t[k]) ++k;
      ex[i] = k;
      c = i;
    }
  }
}

// minimum representation of a string
int minimum_representation(string s) {
  s += s;
  int l = s.length(), i = 0, j = 1, k = 0;
  while (i + k < l && j + k < l) {
    if (s[i + k] == s[j + k]) {
      ++k;
    } else {
      if (s[j + k] > s[i + k]) j += k + 1;
      else i += k + 1;
      k = 0;
      if (i == j) ++j;
    }
  }
  return min(i, j);
}

// l[i], the length of palindrome at the centre of i
int manacher(const char *s, int *l) {
  int n = strlen(s);
  for (int i = 0, j = 0, k; i < n * 2; i += k, j = max(j - k, 0)) {
    while (i >= j && i + j + 1 < n * 2 && s[(i - j) / 2] == s[(i + j + 1) / 2]) ++j;
    l[i] = j;
    for (k = 1; i >= k && j >= k && l[i - k] != j - k; ++k) {
      l[i + k] = min(l[i - k], j - k);
    }
  }
  return *max_element(l, l + n + n);
}

vector<int> z_function(string s) {
    int n = s.size();
    vector<int> z(n);
    int l = 0, r = 0;
    for(int i = 1; i < n; i++) {
        if(i < r)
            z[i] = min(r - i, z[i - l]);
        while(i + z[i] < n && s[z[i]] == s[i + z[i]]) {
            z[i]++;
        }
        if(i + z[i] > r) {
            l = i;
            r = i + z[i];
        }
    }
    return z;
}
```

2.2 Aho-Corasick Automaton

```
struct trie_t {
  bool flag;
  trie_t *child[C], *fail;
} trie[maxn], *root;
trie_t *new_trie() { return &trie[++pt]; }

void add(char *str) {
  int l = strlen(str);
  trie_t *p = root;
  for (int i = 0; i < l; ++i) {
    int ch = str[i]; // fixed to [0, C - 1], C = |SIGMA|
    if (!p->child[ch]) p->child[ch] = new_trie();
    p = p->child[ch];
  }
  p->flag = true;
}

void build() {
  queue<trie_t *> q;
  q.push(root);
  while (!q.empty()) {
    trie_t *p = q.front(), *t;
    q.pop();
    for (int i = 0; i < C; ++i) {
      t = p->fail;
      while (t && !t->child[i]) t = t->child[i];
      t = !t ? root : t->child[i];
      if (p->child[i]) {
        p->child[i]->fail = t;
        p->child[i]->flag |= t->flag;
        q.push(p->child[i]);
```

```
      } else p->child[i] = t;
    }
  }
}
```

2.3 Suffix Array

```
const int maxn = 100002, logn = 21, maxint = 0x7f7f7f7f;
int n, sa[maxn], r[maxn + maxn], h[maxn], mv[maxn][logn];
// r is `doubled`
void initlg() {
  _lg[1] = 0;
  for (int i = 2; i < maxn; ++i) _lg[i] = _lg[i - 1] + ((i & (i - 1)) == 0 ? 1 : 0);
}
// remember to clear r & call initlg() before calling
void construct(int n, int *s) { // initlg();
  static pair<int/* Type */, int> ord[maxn];
  for (int i = 1; i <= n; ++i) ord[i] = make_pair(s[i], i);
  sort(ord + 1, ord + 1 + n);
  for (int i = 1; i <= n; ++i) {
    sa[i] = ord[i].second;
    r[sa[i]] = (i == 1 ? 1 : r[sa[i - 1]] + (ord[i - 1].first != ord[i].first));
  }
  static int tr[maxn], tsa[maxn], c[maxn];
  for (int l = 1; l < n; l <<= 1) {
    int cnt = 0;
    for (int i = 1; i <= n; ++i) if (sa[i] + l > n) tsa[++cnt] = sa[i];
    for (int i = 1; i <= n; ++i) if (sa[i] > l) tsa[++cnt] = sa[i] - l;
    memset(c, 0, sizeof(c));
    for (int i = 1; i <= n; ++i) ++c[r[i]];
    for (int i = 1; i <= n; ++i) c[i] += c[i - 1];
    for (int i = n; i >= 1; --i) sa[c[r[tsa[i]]]--] = tsa[i];
    tr[sa[1]] = 1;
    for (int i = 2; i <= n; ++i) {
      tr[sa[i]] = tr[sa[i - 1]] + (r[sa[i]] != r[sa[i - 1]] || r[sa[i] + l] != r[sa[
        i - 1] + l]);
    }
    for (int i = 1; i <= n; ++i) r[i] = tr[i];
    if (r[sa[n]] == n) break;
  }
  int k = 0; /* Height && RMQ */
  for (int i = 1; i <= n; ++i) {
    if (--k < 0) k = 0;
    for (int j = sa[r[i] - 1]; r[i] != 1 && s[i + k] == s[j + k]; ++k);
    h[r[i]] = k;
  }
  for (int i = 1; i <= n; ++i) mv[i][0] = h[i];
  for (int k = 1; k < logn; ++k) {
    for (int i = 1, len = 1 << (k - 1); i + len <= n; ++i) {
      mv[i][k] = min(mv[i][k - 1], mv[i + len][k - 1]);
    }
  }
}

int askRMQ(int l, int r) {
  int len = r - l + 1, log = _lg[r - l + 1];
  return min(mv[l][log], mv[r - (1 << log) + 1][log]);
}

int LCP(int i, int j) {
  i = r[i], j = r[j];
  if (i > j) swap(i, j);
  return askRMQ(++i, j);
}
```

3 Math

3.1 Matrix Tricks

```
template<typename T, int N>
struct matrix_t {
  T x[N + 1][N + 1];
  matrix_t(T v) {
    memset(x, 0, sizeof(x));
    for (int i = 1; i <= N; ++i) x[i][i] = v;
  }
  T determinant() const {
    T r = 1, det[N + 1][N + 1];
    for (int i = 1; i <= N; ++i) memcpy(det[i], x[i], sizeof(x[i]));
    for (int i = 1; i <= N; ++i) {
      for (int j = i + 1; j <= N; ++j) {
        while (det[j][i] != 0) {
          T ratio = det[i][i] / det[j][i];
          for (int k = i; k <= N; ++k) {
            det[i][k] -= ratio * det[j][k];
            swap(det[i][k], det[j][k]);
          }
          r = -r;
        }
      }
      r = r * det[i][i];
    }
    return r;
  }
};
```

■ Optimization of recursion matrix

$h_n = a_1 h_{n-1} + a_2 h_{n-2} + a_3 h_{n-3} + \ldots + a_k h_{n-k}$, Construct matrix of $k * k$:

$$\mathbf{M} = \begin{bmatrix} a_1 & a_2 & a_3 & \cdots & a_{k-2} & a_{k-1} & a_k \\ 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 & 0 \end{bmatrix}$$

Then the characteristic polynomial of $\mathbf{M}$ is

$$f(\lambda) = |\lambda\mathbf{E} - \mathbf{M}| = \begin{bmatrix} \lambda - a_1 & -a_2 & -a_3 & \cdots & -a_{k-2} & -a_{k-1} & -a_k \\ -1 & \lambda & 0 & \cdots & 0 & 0 & 0 \\ 0 & -1 & \lambda & \cdots & 0 & 0 & 0 \\ 0 & 0 & -1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -1 & \lambda & 0 \\ 0 & 0 & 0 & \cdots & 0 & -1 & \lambda \end{bmatrix}$$
$$= \lambda^k - a_1\lambda^{k-1} - a_2\lambda^{k-2} - \ldots - a_k.$$

Apply Hamilton-Cayley theorem, we have $f(\mathbf{M}) = \mathbf{0}$.
And, $\forall i$, $\mathbf{M}^i$ can be written as a linear combination of $\mathbf{E}$,$\mathbf{M}$,$\mathbf{M}^2$,...,$\mathbf{M}^{k-1}$.
So the matrix multiplication is reduced to polynomial multiplication, which can be computed in $O(n^2)$.

```
int a[maxn], b[maxn], f[maxn], coef[maxn][maxn];

void mul(int k, int a[maxn], int b[maxn], int c[maxn]) {
  static int ret[maxn + maxn];
  for (int i = 0; i <= k + k - 2; ++i) ret[i] = 0;
  for (int i = 0; i < k; ++i) {
    if (a[i] == 0) continue;
    for (int j = 0; j < k; ++j) {
      ret[i + j] += (LL)a[i] * b[j] % mod;
      ret[i + j] %= mod;
    }
  }
  for (int i = 0; i < k; ++i) c[i] = ret[i];
  for (int i = k; i <= k + k - 2; ++i) {
    if (ret[i] == 0) continue;
    for (int j = 0; j < k; ++j) {
      c[j] += (LL)ret[i] * coef[i - (k - 1)][j] % mod;
      c[j] %= mod;
    }
  }
}

void pow(int k, LL p, int x[maxn], int r[maxn]) {
  int a[maxn];
  copy(x, x + maxn, a);
  for (int i = 1; i < k; ++i) r[i] = 0;
  r[0] = 1;
  for (; p; p >>= 1) {
    if (p & 1) mul(k, r, a, r);
    mul(k, a, a, a);
  }
}

int solve(int k, int a[maxn], int f[maxn], LL n) { // use f[0]
  copy(a + 1, a + 1 + k, b + 1);
  reverse(b, b + 1 + k);
  for (int i = 0; i < k; ++i) coef[1][i] = b[i];
  for (int j = 2; j < k; ++j) {
    coef[j][0] = ((LL)coef[j - 1][k - 1] * b[0]) % mod;
    for (int i = 1; i < k; ++i) {
      coef[j][i] = ((LL)coef[j - 1][k - 1] * b[i] + coef[j - 1][i - 1]) % mod;
    }
  }
  int ret[maxn] = {0, 1}, r = 0;
  pow(k, n, ret, ret);
  for (int i = 0; i < k; ++i) {
    r += (LL)ret[i] * f[i] % mod;
    r %= mod;
  }
  return r;
}
```

■ Spanning tree count

Laplacian matrix $L = (\ell_{i,j})_{n*n}$ is defined as: $L = D - A$, that is, it is the difference of the degree matrix D and the adjacency matrix A of the graph.
From the definition it follows that:

$$\ell_{i,j} = \begin{cases} deg(v_i) & \text{if } i = j \\ -1 & \text{if } i \neq j \text{ and } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise} \end{cases}$$

Then the number of spanning trees of a graph on n vertices is the determinant of any $n-1$ submatrix of L.

3.2 Gauss Elimiation

```
void gauss(int n, double g[maxn][maxn]) { // input: N * (N + 1) Matrix
  for (int i = 1; i <= n; ++i) {
    double temp = 0;
    int pos = -1;
    for (int j = i; j <= n; ++j) {
      if (fabs(g[j][i]) > temp) temp = fabs(g[j][i]), pos = j;
    }
    if (pos == -1) continue;
    for (int k = 1; k <= n + 1; ++k) swap(g[pos][k], g[i][k]);
    temp = g[i][i];
    for (int k = 1; k <= n + 1; ++k) g[i][k] /= temp;
    for (int j = i + 1; j <= n; ++j) {
      temp = g[j][i];
      for (int k = 1; k <= n + 1; ++k) g[j][k] -= temp * g[i][k];
    }
  }
  for (int i = n; i >= 1; --i) {
    for (int j = 1; j < i; ++j) {
      g[j][n + 1] -= g[i][n + 1] * g[j][i];
      g[j][i] = 0;
    }
  }
}
```

3.3 Polynomial Root

```
double cal(const vector<double> &coef, double x) {
  double e = 1, s = 0;
  for (int i = 0; i < coef.size(); ++i) s += coef[i] * e, e *= x;
  return s;
}

double find(const vector<double> &coef, double l, double r, int sl, int sr) {
  int sl = dblcmp(cal(coef, l)), sr = dblcmp(cal(coef, r));
  if (sl == 0) return l;
  if (sr == 0) return r;
  for (int tt = 0; tt < 100 && r - l > eps; ++tt) {
    double mid = (l + r) / 2;
    int smid = dblcmp(cal(coef, mid));
    if (smid == 0) return mid;
    if (sl * smid < 0) r = mid;
    else l = mid;
  }
  return (l + r) / 2;
}

vector<double> rec(const vector<double> &coef, int n) {
  vector<double> ret; // c[0]+c[1]*x+c[2]*x^2+...+c[n]*x^n, c[n]==1
  if (n == 1) {
    ret.push_back(-coef[0]);
    return ret;
  }
  vector<double> dcoef(n);
  for (int i = 0; i < n; ++i) dcoef[i] = coef[i + 1] * (i + 1) / n;
  double b = 2; // fujiwara bound
  for (int i = 0; i <= n; ++i) b = max(b, 2 * pow(fabs(coef[i]), 1.0 / (n - i)));
  vector<double> droot = rec(dcoef, n - 1);
  droot.insert(droot.begin(), -b);
  droot.push_back(b);
  for (int i = 0; i + 1 < droot.size(); ++i) {
    int sl = dblcmp(cal(coef, droot[i])), sr = dblcmp(cal(coef, droot[i + 1]));
    if (sl * sr > 0) continue;
    ret.push_back(find(coef, droot[i], droot[i + 1], sl, sr));
  }
  return ret;
}

vector<double> solve(vector<double> coef) {
  int n = coef.size() - 1;
  while (coef.back() == 0) coef.pop_back(), --n;
  for (int i = 0; i <= n; ++i) coef[i] /= coef[n];
  return rec(coef, n);
}
```

3.4 Number Theory Library

```
int exgcd(int x, int y, int &a, int &b) { // extended gcd, ax + by = g.
  int a0 = 1, a1 = 0, b0 = 0, b1 = 1;
  while (y != 0) {
    a0 -= x / y * a1; swap(a0, a1);
    b0 -= x / y * b1; swap(b0, b1);
    x %= y; swap(x, y);
  }
  if (x < 0) a0 = -a0, b0 = -b0, x = -x;
  a = a0, b = b0;
  return x;
}

LL CRT(int cnt, int *p, int *b) { // chinese remainder theorem
  LL N = 1, ans = 0;
  for (int i = 0; i < k; ++i) N *= p[i];
  for (int i = 0; i < k; ++i) {
    LL mult = (inverse(N / p[i], p[i]) * (N / p[i])) % N;
    mult = (mult * b[i]) % N;
    ans += mult; ans %= N;
```

```
  }
  if (ans < 0) ans += N;
  return ans;
}

bool miller_rabin(LL n, LL b) { // miller-rabin prime test
  LL m = n - 1, cnt = 0;
  while (m % 2 == 0) m >>= 1, ++cnt;
  LL ret = fpow(b, m, n);
  if (ret == 1 || ret == n - 1) return true;
  --cnt;
  while (cnt >= 0) {
    ret = mult(ret, ret, n);
    if (ret == n - 1) return true;
    --cnt;
  }
  return false;
}

bool prime_test(LL n) {
  if (n < 2) return false;
  if (n < 4) return true;
  if (n == 3215031751LL) return false;
  const int BASIC[12] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
  for (int i = 0; i < 12 && BASIC[i] < n; ++ i) {
    if (!miller_rabin(n, BASIC[i])) return false;
  }
  return true;
}

LL pollard_rho(LL n, LL seed) { // pollard-rho divisors factorization
  LL x, y;
  x = y = rand() % (n - 1) + 1;
  LL head = 1, tail = 2;
  while (true) {
    x = mult(x, x, n);
    x = (x + seed) % n;
    if (x == y) return n;
    LL d = gcd(max(x - y, y - x), n);
    if (1 < d && d < n) return d;
    if (++head == tail) y = x, tail <<= 1;
  }
}

void factorize(LL n, vector<LL> &divisor) {
  if (n == 1) return;
  if (prime_test(n)) divisor.push_back(n);
  else {
    LL d = n;
    while (d >= n) d = pollard_rho(n, rand() % (n - 1) + 1);
    factorize(n / d, divisor);
    factorize(d, divisor);
  }
}

// primtive root, finding the number with order p-1
int primtive_root(int p) {
  vector<int> factor;
  int tmp = p - 1;
  for (int i = 2; i * i <= tmp; ++i) {
    if (tmp % i == 0) {
      factor.push_back(i);
      while (tmp % i == 0) tmp /= i;
    }
  }
  if (tmp != 1) factor.push_back(tmp);
  for (int root = 1; ; ++root) {
    bool flag = true;
    for (int i = 0; i < factor.size(); ++i) {
      if (fpow(root, (p - 1) / factor[i], p) == 1) {
        flag = false;
        break;
      }
    }
    if (flag) return root;
  }
}
```

3.5 Number Partition

```
// number of ways to divide n to integers(unordered), O(n^(3/2))
int partition(int n) {
  int dp[n + 1];
  dp[0] = 1;
  for (int i = 1; i <= n; i++) {
    dp[i] = 0;
    for (int j = 1, r = 1; i - (3 * j * j - j) / 2 >= 0; ++j, r *= -1) {
      dp[i] += dp[i - (3 * j * j - j) / 2] * r;
      if (i - (3 * j * j + j) / 2 >= 0) dp[i] += dp[i - (3 * j * j + j) / 2] * r;
    }
  }
  return dp[n];
}
```

4 Computational Geometry

4.1 Common 2D

```
// implementation of (dblcmp,dist,cross,dot) is trivial

// count-clock wise is positive direction
double angle(point_t p1, point_t p2) {
  double x1 = p1.x, y1 = p1.y, x2 = p2.x, y2 = p2.y;
  double a1 = atan2(y1, x1), a2 = atan2(y2, x2);
  double a = a2 - a1;
  while (a < -pi) a += 2 * pi;
  while (a >= pi) a -= 2 * pi;
  return a;
}

bool onSeg(point_t p, point_t a, point_t b) {
  return dblcmp(cross(a - p, b - p)) == 0 && dblcmp(dot(a - p, b - p)) <= 0;
}

// 1 normal intersected, -1 denormal intersected, 0 not intersected
int testSS(point_t a, point_t b, point_t c, point_t d) {
  if (dblcmp(max(a.x, b.x) - min(c.x, d.x)) < 0) return 0;
  if (dblcmp(max(c.x, d.x) - min(a.x, b.x)) < 0) return 0;
  if (dblcmp(max(a.y, b.y) - min(c.y, d.y)) < 0) return 0;
  if (dblcmp(max(c.y, d.y) - min(a.y, b.y)) < 0) return 0;
  int d1 = dblcmp(cross(c - a, b - a));
  int d2 = dblcmp(cross(d - a, b - a));
  int d3 = dblcmp(cross(a - c, d - c));
  int d4 = dblcmp(cross(b - c, d - c));
  if ((d1 * d2 < 0) && (d3 * d4 < 0)) return 1;
  if ((d1 * d2 <= 0 && d3 * d4 == 0) || (d1 * d2 == 0 && d3 * d4 <= 0)) return -1;
  return 0;
}

vector<point_t> isLL(point_t a, point_t b, point_t c, point_t d) {
  point_t p1 = b - a, p2 = d - c;
  vector<point_t> ret;
  double a1 = p1.y, b1 = -p1.x, c1;
  double a2 = p2.y, b2 = -p2.x, c2;
  if (dblcmp(a1 * b2 - a2 * b1) == 0) return ret; // colined <=> a1*c2-a2*c1=0 && b1
    *c2-b2*c1=0
  else {
    c1 = a1 * a.x + b1 * a.y;
    c2 = a2 * c.x + b2 * c.y;
    ret.push_back(point_t((c1 * b2 - c2 * b1) / (a1 * b2 - a2 * b1), (c1 * a2 - c2 *
       a1) / (b1 * a2 - b2 * a1)));
    return ret;
  }
}

point_t angle_bisector(point_t p0, point_t p1, point_t p2) {
  point_t v1 = p1 - p0, v2 = p2 - p0;
  v1 = v1 / dist(v1) * dist(v2);
  return v1 + v2 + p0;
}

point_t perpendicular_bisector(point_t p1, point_t p2) {
  point_t v = p2 - p1;
  swap(v.x, v.y);
  v.x = -v.x;
  return v + (p1 + p2) / 2;
}

point_t circumcenter(point_t p0, point_t p1, point_t p2) {
  point_t v1 = perpendicular_bisector(p0, p1);
  point_t v2 = perpendicular_bisector(p1, p2);
  return isLL((p0 + p1) / 2, v1, (p1 + p2) / 2, v2);
}

point_t incenter(point_t p0, point_t p1, point_t p2) {
  point_t v1 = angle_bisector(p0, p1, p2);
  point_t v2 = angle_bisector(p1, p2, p0);
  return isLL(p0, v1, p1, v2);
}

point_t orthocenter(point_t p0, point_t p1, point_t p2) {
  return p0 + p1 + p2 - circumcenter(p0, p1, p2) * 2;
}

// count-clock wise is positive direction
point_t rotate(point_t p, double a) {
  double s = sin(a), c = cos(a);
  return point_t(p.x * c - p.y * s, p.y * c + p.x * s);
}

bool insidePoly(point_t *p, int n, point_t t) {
  p[0] = p[n];
  for (int i = 0; i < n; ++i) if (onSeg(t, p[i], p[i + 1])) return true;
  point_t r = point_t(-3863456.6633422, 5126546.243); // random point
  int cnt = 0;
  for (int i = 0; i < n; ++i) {
    if (testSS(t, r, p[i], p[i + 1]) != 0) ++cnt;
  }
  return cnt & 1;
}
```

```
bool insideConvex(point_t *convex, int n, point_t t) { // O(logN), convex polygen, cross(p[2] - p[1], p[3] - p[1]) > 0
  if (n == 2) return onSeg(t, convex[1], convex[2]);
  int l = 2, r = n;
  while (l < r) {
    int mid = (l + r) / 2 + 1;
    int side = dblcmp(cross(convex[mid] - convex[1], t - convex[1]));
    if (side == 1) l = mid;
    else r = mid - 1;
  }
  int s = dblcmp(cross(convex[l] - convex[1], t - convex[1]));
  if (s == -1 || l == n) return false;
  point_t v = convex[l + 1] - convex[l];
  if (dblcmp(cross(v, t - convex[l])) >= 0) return true;
  return false;
}
```

4.2 Graham Convex Hull

```
bool cmp(const point_t p1, const point_t p2) {
  return dblcmp(p1.y - p2.y) == 0 ? p1.x < p2.x : p1.y < p2.y;
}

int graham(point_t *p) { // Points co-lined are ignored.
  int top = 2; static point_t sk[maxn];
  sort(p + 1, p + 1 + n, cmp);
  sk[1] = p[1], sk[2] = p[2];
  for (int i = 3; i <= n; ++i) {
    while (top >= 2 && dblcmp(cross(p[i] - sk[top - 1], sk[top] - sk[top - 1])) >= 0) --top;
    sk[++top] = p[i];
  }
  int ttop = top;
  for (int i = n - 1; i >= 1; --i) {
    while (top > ttop && dblcmp(cross(p[i] - sk[top - 1], sk[top] - sk[top - 1])) >= 0) --top;
    sk[++top] = p[i];
  }
  for (int i = 1; i < top; ++i) p[i] = sk[i];
  return --top;
}
```

4.3 Minkowski Sum of Convex Hull

Wiki:
The Minkowski sum of two sets of position vectors A and B in Euclidean space is formed by adding each vector in A to each vector in B, i.e. the set

$$A + B = \{\vec{a} + \vec{b} \,|\, \vec{a} \in A, \vec{b} \in B\}.$$

For all subsets S_1 and S_2 of a real vector-space, the convex hull of their Minkowski sum is the Minkowski sum of their convex hulls $\mathrm{Conv}(S_1 + S_2) = \mathrm{Conv}(S_1) + \mathrm{Conv}(S_2)$. Minkowski sums are used in motion planning of an object among obstacles. They are used for the computation of the configuration space, which is the set of all admissible positions of the object. In the simple model of translational motion of an object in the plane, where the position of an object may be uniquely specified by the position of a fixed point of this object, the configuration space are the Minkowski sum of the set of obstacles and the movable object placed at the origin and rotated 180 degrees.

```
int minkowski(point_t *h, point_t *h1, point_t *h2, int n, int m) {
  point_t c = point_t(0, 0);
  for (int i = 1; i <= m; ++i) c = c + h2[i];
  c = c / m;
  for (int i = 1; i <= m; ++i) h2[i] = h2[i] - c;
  int cur = -1;
  for (int i = 1; i <= m; ++i) {
    if (dblcmp(cross(h2[i], h1[1] - h1[n])) >= 0) {
      if (cur == -1 || cross(h2[i], h1[1] - h1[n]) > cross(h2[cur], h1[1] - h1[n]))
        cur = i;
    }
  }
  int cnt = 0;
  h1[n + 1] = h1[1];
  for (int i = 1; i <= n; ++i) {
    while (true) {
      h[++cnt] = h1[i] + h2[cur];
      int next = (cur == m ? 1 : cur + 1);
      if (dblcmp(cross(h2[cur], h1[i + 1] - h1[i])) < 0) cur = next;
      else {
        if (cross(h2[next], h1[i + 1] - h1[i]) > cross(h2[cur], h1[i + 1] - h1[i]))
          cur = next;
        else break;
      }
    }
  }
  for (int i = 1; i <= cnt; ++i) h[i] = h[i] + c;
  for (int i = 1; i <= m; ++i) h2[i] = h2[i] + c;
  return graham(h, cnt);
}
```

4.4 Rotating Calipers

```
// Calculate the maximum distance of a point set.
```

```
double rotate_caliper() {
  p[0] = p[n];
  int to = 0;
  double ans = 0;
  for (int i = 0; i < n; ++i) {
    while ((to + 1) % n != i) {
      if (dblcmp(cross(p[i + 1] - p[i], p[to + 1] - p[i]) - cross(p[i + 1] - p[i], p
        [to] - p[i])) >= 0) to = (to + 1) % n;
      else break;
    }
    ans = max(ans, dist(p[i], p[to]));
    ans = max(ans, dist(p[i + 1], p[to]));
  }
  return ans;
}
```

4.5 Closest Pair Points

```
double dac(point_t *p, int l, int r) {
  double d = 10e100;
  if (r - l <= 3) {
    for (int i = l; i <= r; ++i) {
      for (int j = i + 1; j <= r; ++j) {
        d = min(d, dist2(p[i], p[j]));
      }
    }
    sort(p + l, p + r + 1, cmpY);
  } else {
    int mid = (l + r) / 2;
    d = min(dac(p, l, mid), dac(p, mid + 1, r));
    inplace_merge(p + l, p + mid + 1, p + r + 1, cmpY);
    static point_t tmp[maxn]; int cnt = 0;
    for (int i = l; i <= r; ++i) {
      if ((p[i].x - p[mid].x) * (p[i].x - p[mid].x) <= d) tmp[++cnt] = p[i];
    }
    for (int i = 1; i <= cnt; ++i) {
      for (int j = 1; j <= 8 && j + i <= cnt; ++j) {
        d = min(d, dist2(tmp[i], tmp[j + i]));
      }
    }
  }
  return d;
}

double cal(point_t *p, int n) {
  sort(p + 1, p + 1 + n, cmpX);
  return sqrt(dac(p, 1, n));
}
```

4.6 Halfplane Intersection

```
// O(N^2) sol, polygon counterclockwise order
// i.e., left side of vector v1->v2 is the valid half plane
const double maxd = 1e5;
int n, cnt;
point_t p[maxn];

void init() { // order reversed if right side
  cnt = 4;
  p[1] = point_t(-maxd, -maxd);
  p[2] = point_t(maxd, -maxd);
  p[3] = point_t(maxd, maxd);
  p[4] = point_t(-maxd, maxd);
}

void cut(point_t p1, point_t p2) {
  int tcnt = 0;
  static point_t tp[maxn];
  p[cnt + 1] = p[1];
  for (int i = 1; i <= cnt; ++i) {
    double v1 = cross(p2 - p1, p[i] - p1);
    double v2 = cross(p2 - p1, p[i + 1] - p1);
    if (dblcmp(v1) >= 0) tp[++tcnt] = p[i]; // <= if right side
    if (dblcmp(v1) * dblcmp(v2) < 0) tp[++tcnt] = isLL(p1, p2, p[i], p[i + 1]);
  }
  cnt = tcnt;
  for (int i = 1; i <= cnt; ++i) p[i] = tp[i];
}
```

```
// O(NlogN) sol, Left is valid half plane. Note that the edge of hull may degenerate
   to a point.
struct hp_t {
  point_t p1, p2;
  double a;
  hp_t() { }
  hp_t(point_t tp1, point_t tp2) : p1(tp1), p2(tp2) {
    tp2 = tp2 - tp1;
    a = atan2(tp2.y, tp2.x);
  }
  bool operator==(const hp_t &r) const {
    return dblcmp(a - r.a) == 0;
  }
  bool operator<(const hp_t &r) const {
    if (dblcmp(a - r.a) == 0) return dblcmp(cross(r.p2 - r.p1, p2 - r.p1)) >= 0;
    else return a < r.a;
```

```
  }
} hp[maxn];

void addhp(point_t p1, point_t p2) {
  hp[++cnt] = hp_t(p1, p2);
}

void init() {
  cnt = 0;
  addhp(point_t(-maxd, -maxd), point_t(maxd, -maxd));
  addhp(point_t(maxd, -maxd), point_t(maxd, maxd));
  addhp(point_t(maxd, maxd), point_t(-maxd, maxd));
  addhp(point_t(-maxd, maxd), point_t(-maxd, -maxd));
}

bool checkhp(hp_t h1, hp_t h2, hp_t h3) {
  point_t p = isLL(h1.p1, h1.p2, h2.p1, h2.p2);
  return dblcmp(cross(p - h3.p1, h3.p2 - h3.p1)) > 0;
}

vector<point_t> hp_inter() {
  sort(hp + 1, hp + 1 + cnt);
  cnt = unique(hp + 1, hp + 1 + cnt) - hp - 1;
  deque<hp_t> DQ;
  DQ.push_back(hp[1]);
  DQ.push_back(hp[2]);
  for (int i = 3; i <= cnt; ++i) {
    while (DQ.size() > 1 && checkhp(*----DQ.end(), *--DQ.end(), hp[i])) DQ.pop_back
      ();
    while (DQ.size() > 1 && checkhp(*++DQ.begin(), *DQ.begin(), hp[i])) DQ.pop_front
      ();
    DQ.push_back(hp[i]);
  }
  while (DQ.size() > 1 && checkhp(*----DQ.end(), *--DQ.end(), DQ.front())) DQ.
    pop_back();
  while (DQ.size() > 1 && checkhp(*++DQ.begin(), *DQ.begin(), DQ.back())) DQ.
    pop_front();
  DQ.push_front(DQ.back());
  vector<point_t> res;
  while (DQ.size() > 1) {
    hp_t tmp = DQ.front();
    DQ.pop_front();
    res.push_back(isLL(tmp.p1, tmp.p2, DQ.front().p1, DQ.front().p2));
  }
  return res;
}
```

4.7 Tri-Cir Intersection & Tangent

```
vector<point_t> tanCP(point_t c, double r, point_t p) {
  double x = dot(p - c, p - c);
  double d = x - r * r;
  vector<point_t> res;
  if (d < -eps) return res;
  if (d < 0) d = 0;
  point_t q1 = (p - c) * (r * r / x);
  point_t q2 = ((p - c) * (-r * sqrt(d) / x)).rot90(); // rot90: (-y, x)
  res.push_back(c + q1 - q2);
  res.push_back(c + q1 + q2);
  return res;
}

vector<seg_t> tanCC(point_t c1, double r1, point_t c2, double r2) {
  vector<seg_t> res;
  if (abs(r1 - r2) < eps) {
    point_t dir = c2 - c1;
    dir = (dir * (r1 / dir.l())).rot90();
    res.push_back(seg_t(c1 + dir, c2 + dir));
    res.push_back(seg_t(c1 - dir, c2 - dir));
  } else {
    point_t p = ((c1 * -r2) + (c2 * r1)) / (r1 - r2);
    vector<point_t> ps = tanCP(c1, r1, p), qs = tanCP(c2, r2, p);
    for (int i = 0; i < ps.size() && i < qs.size(); ++i) {
      res.push_back(seg_t(ps[i], qs[i]));
    }
  }
  point_t p = ((c1 * r2) + (c2 * r1)) / (r1 + r2);
  vector<point_t> ps = tanCP(c1, r1, p), qs = tanCP(c2, r2, p);
  // point_t tmp = (c2 - c1).rot90().rot90().rot90();
  for (int i = 0; i < ps.size() && i < qs.size(); ++i) {
    /* if (dblcmp(dist(ps[i], qs[i])) == 0) {
      qs[i] = qs[i] + tmp;
      tmp = tmp.rot90().rot90();
    }*/
    res.push_back(seg_t(ps[i], qs[i]));
  }
  return res;
}
```

```
// Assume d <= r1 + r2 && d >= |r1 - r2|
pair<point_t, point_t> isCC(point_t c1, point_t c2, double r1, double r2) {
  if (r1 < r2) swap(c1, c2), swap(r1, r2);
  double d = dist(c1, c2);
  double x1 = c1.x, x2 = c2.x, y1 = c1.y, y2 = c2.y;
  double mid = atan2(y2 - y1, x2 - x1);
  double a = r1, c = r2;
```

```
  double t = acos(max(0.0, a * a + d * d - c * c) / (2 * a * d));
  point_t p1 = point_t(cos(mid - t) * r1, sin(mid - t) * r1) + c1;
  point_t p2 = point_t(cos(mid + t) * r1, sin(mid + t) * r1) + c1;
  return make_pair(p1, p2);
}

int testCC(point_t c1, point_t c2, double r1, double r2) {
  double d = dist(c1, c2);
  if (dblcmp(r1 + r2 - d) <= 0) return 1; // not intersected or tged
  if (dblcmp(r1 + d - r2) <= 0) return 2; // C1 inside C2
  if (dblcmp(r2 + d - r1) <= 0) return 3; // C2 inside C1
  return 0; // intersected
}

point_t isCL(point_t a, point_t b, point_t o, double r) {
  double x0 = o.x, y0 = o.y;
  double x1 = a.x, y1 = a.y;
  double x2 = b.x, y2 = b.y;
  double dx = x2 - x1, dy = y2 - y1;
  double A = dx * dx + dy * dy;
  double B = 2 * dx * (x1 - x0) + 2 * dy * (y1 - y0);
  double C = (x1 - x0) * (x1 - x0) + (y1 - y0) * (y1 - y0) - r * r;
  double delta = B * B - 4 * A * C;
  if (delta >= 0) {
    delta = sqrt(delta);
    double t1 = (-B - delta) / 2 / A;
    double t2 = (-B + delta) / 2 / A;
    if (dblcmp(t1) >= 0) return point_t(x1 + t1 * dx, y1 + t1 * dy); // Ray
    if (dblcmp(t2) >= 0) return point_t(x1 + t2 * dx, y1 + t2 * dy);
  }
  return point_t();
}
```

```
double areaTC(point_t ct, double r, point_t p1, point_t p2) { // intersected area
  double a, b, c, x, y, s = cross(p1 - ct, p2 - ct) / 2;
  a = dist(ct, p2), b = dist(ct, p1), c = dist(p1, p2);
  if (a <= r && b <= r) {
    return s;
  } else if (a < r && b >= r) {
    x = (dot(p1 - p2, ct - p2) + sqrt(c * c * r * r - sqr(cross(p1 - p2, ct - p2))))
       / c;
    return asin(s * (c - x) * 2 / c / b / r) * r * r / 2 + s * x / c;
  } else if (a >= r && b < r) {
    y = (dot(p2 - p1, ct - p1) + sqrt(c * c * r * r - sqr(cross(p2 - p1, ct - p1))))
       / c;
    return asin(s * (c - y) * 2 / c / a / r) * r * r / 2 + s * y / c;
  } else {
    if (fabs(2 * s) >= r * c || dot(p2 - p1, ct - p1) <= 0 || dot(p1 - p2, ct - p2)
      <= 0) {
      if (dot(p1 - ct, p2 - ct) < 0) {
        if (cross(p1 - ct, p2 - ct) < 0) {
          return (-pi - asin(s * 2 / a / b)) * r * r / 2;
        } else {
          return (pi - asin(s * 2 / a / b)) * r * r / 2;
        }
      } else {
        return asin(s * 2 / a / b) * r * r / 2;
      }
    } else {
      x = (dot(p1 - p2, ct - p2) + sqrt(c * c * r * r - sqr(cross(p1 - p2, ct - p2))
        )) / c;
      y = (dot(p2 - p1, ct - p1) + sqrt(c * c * r * r - sqr(cross(p2 - p1, ct - p1))
        )) / c;
      return (asin(s * (1 - x / c) * 2 / r / b) + asin(s * (1 - y / c) * 2 / r / a))
         * r * r / 2 + s * ((y + x) / c - 1);
    }
  }
}

double areaTC(point_t ct, double r, point_t p1, point_t p2, point_t p3) {
  return areaTC(ct, r, p1, p2) + areaTC(ct, r, p2, p3) + areaTC(ct, r, p3, p1);
}
```

4.8 Minimum Covering Circle

```
void set_circle(point_t &p, double &r, point_t a, point_t b) {
  r = dist(a, b) / 2;
  p = (a + b) / 2;
}

void set_circle(point_t &p, double &r, point_t a, point_t b, point_t c) {
  if (dblcmp(cross(b - a, c - a)) == 0) {
    if (dist(a, c) > dist(b, c)) {
      r = dist(a, c) / 2;
      p = (a + c) / 2;
    } else {
      r = dist(b, c) / 2;
      p = (b + c) / 2;
    }
  } else {
    p = circumcenter(a, b, c);
    r = dist(p, a);
  }
}

bool in_circle(point_t &p, double &r, point_t x) {
  return dblcmp(dist(x, p) - r) <= 0;
```

```
}

pair<point_t, double> minimum_circle(int n, point_t *p) {
  point_t c = point_t(0, 0);
  double r = 0;
  random_shuffle(p + 1, p + 1 + n);
  set_circle(c, r, p[1], p[2]);
  for (int i = 3; i <= n; ++i) {
    if (in_circle(c, r, p[i])) continue;
    set_circle(c, r, p[i], p[1]);
    for (int j = 2; j < i; ++j) {
      if (in_circle(c, r, p[j])) continue;
      set_circle(c, r, p[i], p[j]);
      for (int k = 1; k < j; ++k) {
        if (in_circle(c, r, p[k])) continue;
        set_circle(c, r, p[i], p[j], p[k]);
      }
    }
  }
  return make_pair(c, r);
}
```

4.9 Convex Polygon Area Union

```
// modified from syntax_error's code
bool operator<(const point_t &a, const point_t &b) {
  if (dblcmp(a.x - b.x) == 0) return a.y < b.y;
  return a.x < b.x;
}

bool operator==(const point_t &a, const point_t &b) {
  return dblcmp(a.x - b.x) == 0 && dblcmp(a.y - b.y) == 0;
}

struct segment_t {
  point_t a, b;
  segment_t() { a = b = point_t(); }
  segment_t(point_t ta, point_t tb) : a(ta), b(tb) { }
  double len() const { return dist(a, b); }
  double k() const { return (a.y - b.y) / (a.x - b.x); }
  double l() const { return a.y - k() * a.x; }
};

struct line_t {
  double a, b, c;
  line_t(point_t p) { a = p.x, b = -1.0, c = -p.y; }
  line_t(point_t p, point_t q) {
    a = p.y - q.y;
    b = q.x - p.x;
    c = a * p.x + b * p.y;
  }
};

bool ccutl(line_t p, line_t q) {
  if (dblcmp(p.a * q.b - q.a * p.b) == 0) return false;
  return true;
}

point_t cutl(line_t p, line_t q) {
  double x = (p.c * q.b - q.c * p.b) / (p.a * q.b - q.a * p.b);
  double y = (p.c * q.a - q.c * p.a) / (p.b * q.a - q.b * p.a);
  return point_t(x, y);
}

bool onseg(point_t p, segment_t s) {
  if (dblcmp(p.x - min(s.a.x, s.b.x)) < 0 || dblcmp(p.x - max(s.a.x, s.b.x)) > 0)
    return false;
  if (dblcmp(p.y - min(s.a.y, s.b.y)) < 0 || dblcmp(p.y - max(s.a.y, s.b.y)) > 0)
    return false;
  return true;
}

bool ccut(segment_t p, segment_t q) {
  if (!ccutl(line_t(p.a, p.b), line_t(q.a, q.b))) return false;
  point_t r = cutl(line_t(p.a, p.b), line_t(q.a, q.b));
  if (!onseg(r, p) || !onseg(r, q)) return false;
  return true;
}

point_t cut(segment_t p, segment_t q) {
  return cutl(line_t(p.a, p.b), line_t(q.a, q.b));
}

struct event_t {
  double x;
  int type;
  event_t() { x = 0, type = 0; }
  event_t(double _x, int _t) : x(_x), type(_t) { }
  bool operator<(const event_t &r) const {
    return x < r.x;
  }
};

vector<segment_t> s;

double solve(const vector<segment_t> &v, const vector<int> &sl) {
```

```
  double ret = 0;
  vector<point_t> lines;
  for (int i = 0; i < v.size(); ++i) lines.push_back(point_t(v[i].k(), v[i].l()));
  sort(lines.begin(), lines.end());
  lines.erase(unique(lines.begin(), lines.end()), lines.end());
  for(int i = 0; i < lines.size(); ++i) {
    vector<event_t> e;
    vector<int>::const_iterator it = sl.begin();
    for(int j = 0; j < s.size(); j += *it++) {
      bool touch = false;
      for (int k = 0; k < *it; ++k) if (lines[i] == point_t(s[j + k].k(), s[j + k].l
        ())) touch = true;
      if (touch) continue;
      vector<point_t> cuts;
      for (int k = 0; k < *it; ++k) {
        if (!ccutl(line_t(lines[i]), line_t(s[j + k].a, s[j + k].b))) continue;
        point_t r = cutl(line_t(lines[i]), line_t(s[j + k].a, s[j + k].b));
        if (onseg(r, s[j + k])) cuts.push_back(r);
      }
      sort(cuts.begin(), cuts.end());
      cuts.erase(unique(cuts.begin(), cuts.end()), cuts.end());
      if (cuts.size() == 2) {
        e.push_back(event_t(cuts[0].x, 0));
        e.push_back(event_t(cuts[1].x, 1));
      }
    }
    for (int j = 0; j < v.size(); ++j) {
      if (lines[i] == point_t(v[j].k(), v[j].l())) {
        e.push_back(event_t(min(v[j].a.x, v[j].b.x), 2));
        e.push_back(event_t(max(v[j].a.x, v[j].b.x), 3));
      }
    }
    sort(e.begin(), e.end());
    double last = e[0].x;
    int cntg = 0, cntb = 0;
    for (int j = 0; j < e.size(); ++j) {
      double y0 = lines[i].x * last + lines[i].y;
      double y1 = lines[i].x * e[j].x + lines[i].y;
      if (cntb == 0 && cntg) ret += (y0 + y1) * (e[j].x - last) / 2;
      last = e[j].x;
      if (e[j].type == 0) ++cntb;
      if (e[j].type == 1) --cntb;
      if (e[j].type == 2) ++cntg;
      if (e[j].type == 3) --cntg;
    }
  }
  return ret;
}
```

```

double polyUnion(vector<vector<point_t> > polys) {
  s.clear();
  vector<segment_t> A, B;
  vector<int> sl;
  for (int i = 0; i < polys.size(); ++i) {
    double area = 0;
    int tot = polys[i].size();
    for (int j = 0; j < tot; ++j) {
      area += cross(polys[i][j], polys[i][(j + 1) % tot]);
    }
    if (dblcmp(area) > 0) reverse(polys[i].begin(), polys[i].end());
    if (dblcmp(area) != 0) {
      sl.push_back(tot);
      for (int j = 0; j < tot; ++j) s.push_back(segment_t(polys[i][j], polys[i][(j +
        1) % tot]));
    }
  }
  for (int i = 0; i < s.size(); ++i) {
    int sgn = dblcmp(s[i].a.x - s[i].b.x);
    if (sgn == 0) continue;
    else if (sgn < 0) A.push_back(s[i]);
    else B.push_back(s[i]);
  }
  return solve(A, sl) - solve(B, sl);
}
```

4.10 3D Common

```
double dot(point3_t p1, point3_t p2) {
  return p1.x * p2.x + p1.y * p2.y + p1.z * p2.z;
}

point3_t cross(point3_t p1, point3_t p2) {
  return point3_t(p1.y * p2.z - p1.z * p2.y, p1.z * p2.x - p1.x * p2.z, p1.x * p2.y
    - p1.y * p2.x);
}

double volume(point3_t p1, point3_t p2, point3_t p3, point3_t p4) {
  point3_t v1 = cross(p2 - p1, p3 - p1);
  p4 = p4 - p1;
  return dot(v1, p4) / 6;
}

double area(point3_t p1, point3_t p2, point3_t p3) {
  return cross(p2 - p1, p3 - p1).length() / 2;
}

```

```
pair<point3_t, point3_t> isFF(point3_t p1, point3_t o1, point3_t p2, point3_t o2) {
  point3_t e = cross(o1, o2), v = cross(o1, e);
  double d = dot(o2, v);
  if (fabs(d) < eps) throw -1;
  point3_t q = p1 + (v * (dot(o2, p2 - p1) / d));
  return make_pair(q, q + e);
}

double distLL(point3_t p1, point3_t u, point3_t p2, point3_t v) {
  double s = dot(u, v) * dot(v, p1 - p2) - dot(v, v) * dot(u, p1 - p2);
  double t = dot(u, u) * dot(v, p1 - p2) - dot(u, v) * dot(u, p1 - p2);
  double deno = dot(u, u) * dot(v, v) - dot(u, v) * dot(u, v);
  if (dblcmp(deno) == 0) return dist(p1, p2 + v * (dot(p1 - p2, u) / dot(u, v)));
  s /= deno; t /= deno;
  point3_t a = p1 + u * s, b = p2 + v * t;
  return dist(a, b);
}
```

5 Graph

5.1 Maximum Flow

```
// assuming the sink has the maximum vertex ID => t = n
int n, m, s, t, ec, d[maxn], vd[maxn];
struct edge_link {
  int v, r;
  edge_link *next, *pair;
} edge[maxm], *header[maxn], *current[maxn];
void add(int u, int v, int r) // (u, v, r), (v, u, 0)

int augment(int u, int flow) {
  if (u == t) return flow;
  int temp, res = 0;
  for (edge_link *&e = current[u]; e != NULL; e = e->next) {
    if (e->r && d[u] == d[e->v] + 1) {
      temp = augment(e->v, min(e->r, flow - res));
      e->r -= temp, e->pair->r += temp, res += temp;
      if (d[s] == t || res == flow) return res;
    }
  }
  if (--vd[d[u]] == 0) d[s] = t;
  else current[u] = header[u], ++vd[++d[u]];
  return res;
}

int sap() {
  int flow = 0;
  memset(d, 0, sizeof(d)), memset(vd, 0, sizeof(vd));
  vd[0] = t;
  for (int i = 1; i <= t; ++i) current[i] = header[i];
  while (d[s] < t) flow += augment(s, maxint);
  return flow;
}
```

Notes on vertex covering and independent set on bipartite graph:
Minimum Vertex Covering Set V': $\forall (u, v) \in E$, $u \in V'$ or $v \in V'$ holds.
Maximum Vertex Independent Set V': $\forall u, v \in V'$, $(u, v) \notin E$ holds.
Construct a flow graph G, run DFS from s on reduction graph, the vertices not visited in left side and visited in right side form the minimum vertex covering set.
Maximum vertex independent set vice versa.

5.2 Minimum Cost, Maximum Flow

```
int n, m, s, t, ec, d[maxn]; // Minimum cost, maximum flow(ZKW version), t=n
struct edge_link {
  int v, r, w;
  edge_link *next, *pair;
} edge[maxm], *header[maxn];
bool vis[maxn];
void add(int u, int v, int r, int w) // (u, v, r, w), (v, u, 0, -w)

void spfa() {
  queue<int> q;
  for (int i = 1; i <= t; ++i) d[i] = maxint, vis[i] = false;
  d[s] = 0, q.push(s), vis[s] = true;
  while (!q.empty()) {
    int u = q.front();
    q.pop(), vis[u] = false;
    for (edge_link *e = header[u]; e != NULL; e = e->next) {
      if (e->r && d[u] + e->w < d[e->v]) {
        d[e->v] = d[u] + e->w;
        if (!vis[e->v]) q.push(e->v), vis[e->v] = true;
      }
    }
  }
  for (int i = 1; i <= t; ++i) d[i] = d[t] - d[i];
}

int augment(int u, int flow) {
  if (u == t) return flow;
  vis[u] = true;
  for (edge_link *e = header[u]; e != NULL; e = e->next) {
    if (e->r && !vis[e->v] && d[e->v] + e->w == d[u]) {
      int temp = augment(e->v, min(flow, e->r));
```

```
      if (temp) {
        e->r -= temp, e->pair->r += temp;
        return temp;
      }
    }
  }
  return 0;
}

bool adjust() {
  int delta = maxint;
  for (int u = 1; u <= t; ++u) {
    if (!vis[u]) continue;
    for (edge_link *e = header[u]; e != NULL; e = e->next) {
      if (e->r && !vis[e->v] && d[e->v] + e->w > d[u]) {
        delta = min(delta, d[e->v] + e->w - d[u]);
      }
    }
  }
  if (delta == maxint) return false;
  for (int i = 1; i <= t; ++i) {
    if (vis[i]) d[i] += delta;
  }
  memset(vis, 0, sizeof(vis));
  return true;
}

pair<int, int> cost_flow() {
  int temp, flow = 0, cost = 0;
  spfa();
  do {
    while (temp = augment(s, maxint)) {
      flow += temp;
      memset(vis, 0, sizeof(vis));
    }
  } while (adjust());
  for (int i = 2; i <= ec; i += 2) cost += edge[i].r * edge[i - 1].w;
  return make_pair(flow, cost);
}
```

5.3 Minimum Mean Cycle

```
int dp[maxn][maxn]; // minimum mean cycle(allow negative weight)
double mmc(int n) {
  for (int i = 0; i < n; ++i) {
    memset(dp[i + 1], 0x7f, sizeof(dp[i + 1]));
    for (int j = 1; j <= ec; ++j) {
      int u = edge[j].u, v = edge[j].v, w = edge[j].w;
      if (dp[i][u] != maxint) dp[i + 1][v] = min(dp[i + 1][v], dp[i][u] + w);
    }
  }
  double res = maxdbl;
  for (int i = 1; i <= n; ++i) {
    if (dp[n][i] == maxint) continue;
    double value = -maxdbl;
    for (int j = 0; j < n; ++j) {
      value = max(value, (double)(dp[n][i] - dp[j][i]) / (n - j));
    }
    res = min(res, value);
  }
  return res;
}
```

6 Miscellaneous

6.1 Convex Hull Trick

```
typedef int ftype;
typedef complex<ftype> point;
#define x real
#define y imag

ftype dot(point a, point b) {
    return (conj(a) * b).x();
}

ftype cross(point a, point b) {
    return (conj(a) * b).y();
}

vector<point> hull, vecs;

void add_line(ftype k, ftype b) {
    point nw = {k, b};
    while(!vecs.empty() && dot(vecs.back(), nw - hull.back()) < 0) {
        hull.pop_back();
        vecs.pop_back();
    }
    if(!hull.empty()) {
        vecs.push_back(1i * (nw - hull.back()));
    }
    hull.push_back(nw);
}

int get(ftype x) {
```

```
    point query = {x, 1};
    auto it = lower_bound(vecs.begin(), vecs.end(), query, [](point a, point b) {
        return cross(a, b) > 0;
    });
    return dot(query, hull[it - vecs.begin()]);
}
```

6.2 Li Chao Tree

```
typedef long long ftype;
typedef complex<ftype> point;
#define x real
#define y imag

ftype dot(point a, point b) {
    return (conj(a) * b).x();
}

ftype f(point a,  ftype x) {
    return dot(a, {x, 1});
}

const int maxn = 2e5;

point line[4 * maxn];

void add_line(point nw, int v = 1, int l = 0, int r = maxn) {
    int m = (l + r) / 2;
    bool lef = f(nw, l) < f(line[v], l);
    bool mid = f(nw, m) < f(line[v], m);
    if(mid) {
        swap(line[v], nw);
    }
    if(r - l == 1) {
        return;
    } else if(lef != mid) {
        add_line(nw, 2 * v, l, m);
    } else {
        add_line(nw, 2 * v + 1, m, r);
    }
}

ftype get(int x, int v = 1, int l = 0, int r = maxn) {
    int m = (l + r) / 2;
    if(r - l == 1) {
        return f(line[v], x);
    } else if(x < m) {
        return min(f(line[v], x), get(x, 2 * v, l, m));
    } else {
        return min(f(line[v], x), get(x, 2 * v + 1, m, r));
    }
}
```

6.3 Strongly Connected Component

```
vector<bool> visited; // keeps track of which vertices are already visited

void dfs(int v, vector<vector<int>> const& adj, vector<int> &output) {
    visited[v] = true;
    for (auto u : adj[v])
        if (!visited[u])
            dfs(u, adj, output);
    output.push_back(v);
}

// input: adj -- adjacency list of G
// output: components -- the strongy connected components in G
// output: adj_cond -- adjacency list of G^SCC (by root vertices)
void strongly_connected_components(vector<vector<int>> const& adj,
                                   vector<vector<int>> &components,
                                   vector<vector<int>> &adj_cond) {
    int n = adj.size();
    components.clear(), adj_cond.clear();

    vector<int> order; // will be a sorted list of G's vertices by exit time

    visited.assign(n, false);

    // first series of depth first searches
    for (int i = 0; i < n; i++)
        if (!visited[i])
            dfs(i, adj, order);

    // create adjacency list of G^T
    vector<vector<int>> adj_rev(n);
    for (int v = 0; v < n; v++)
        for (int u : adj[v])
            adj_rev[u].push_back(v);

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

    vector<int> roots(n, 0); // gives the root vertex of a vertex's SCC

    // second series of depth first searches
    for (auto v : order)
```

```
        if (!visited[v]) {
            std::vector<int> component;
            dfs(v, adj_rev, component);
            components.push_back(component);
            int root = *min_element(begin(component), end(component));
            for (auto u : component)
                roots[u] = root;
        }

    // add edges to condensation graph
    adj_cond.assign(n, {});
    for (int v = 0; v < n; v++)
        for (auto u : adj[v])
            if (roots[v] != roots[u])
                adj_cond[roots[v]].push_back(roots[u]);
}
```

6.4 Linear Programming

```
double a[maxn][maxm], b[maxn], c[maxm], d[maxn][maxm];
int ix[maxn + maxm]; // !!! array all indexed from 0
// max{cx|Ax<=b,x>=0}, n: constraints, m: vars
double simplex(double a[maxn][maxm], double b[maxn], double c[maxm], int n, int m) {
  ++m;
  int r = n, s = m - 1;
  memset(d, 0, sizeof(d));
  for (int i = 0; i < n + m; ++i) ix[i] = i;
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < m - 1; ++j) d[i][j] = -a[i][j];
    d[i][m - 1] = 1;
    d[i][m] = b[i];
    if (d[r][m] > d[i][m]) r = i;
  }
  for (int j = 0; j < m - 1; ++j) d[n][j] = c[j];
  d[n + 1][m - 1] = -1;
  for (double dd;; ) {
    if (r < n) {
      int t = ix[s]; ix[s] = ix[r + m]; ix[r + m] = t;
      d[r][s] = 1.0 / d[r][s];
      for (int j = 0; j <= m; ++j) if (j != s) d[r][j] *= -d[r][s];
      for (int i = 0; i <= n + 1; ++i) if (i != r) {
        for (int j = 0; j <= m; ++j) if (j != s) d[i][j] += d[r][j] * d[i][s];
        d[i][s] *= d[r][s];
      }
    }
    r = -1; s = -1;
    for (int j = 0; j < m; ++j) if (s < 0 || ix[s] > ix[j]) {
      if (d[n + 1][j] > eps || (d[n + 1][j] > -eps && d[n][j] > eps)) s = j;
    }
    if (s < 0) break;
    for (int i = 0; i < n; ++i) if (d[i][s] < -eps) {
      if (r < 0 || (dd = d[r][m] / d[r][s] - d[i][m] / d[i][s]) < -eps || (dd < eps
        && ix[r + m] > ix[i + m])) r = i;
    }
    if (r < 0) return -1; // not bounded
  }
  if (d[n + 1][m] < -eps) return -1; // not executable
  double ans = 0;
  for (int i = m; i < n + m; ++i) { // the missing enumerated x[i] = 0
    if (ix[i] < m - 1) ans += d[i - m][m] * c[ix[i]];
  }
  return ans;
}
```

6.5 Simpson Integration

```
const double eps = 1e-15;
double f(double x) { return 0.0; }
double sim(double l, double r, double lv, double rv, double mv) {
  return (r - l) * (lv + rv + 4 * mv) / 6;
}
double rsim(double l, double r, double lv, double rv, double mv, double m1v, double
  m2v) {
  double mid = (l + r) / 2;
  if (fabs(sim(l, r, lv, rv, mv) - sim(l, mid, lv, mv, m1v) - sim(mid, r, mv, rv,
    m2v)) / 15 < eps) {
    return sim(l, r, lv, rv, mv);
  } else {
    double mid = (l + r) / 2, m1 = (l + (l + r) / 2) / 2, m2 = ((l + r) / 2 + r) /
      2;
    return rsim(l, mid, lv, mv, m1v, f((l + m1) / 2), f((m1 + mid) / 2)) + rsim(mid,
       r, mv, rv, m2v, f((mid + m2) / 2), f((m2 + r) / 2));
  }
}
double simpson(double l, double r) {
  double mid = (l + r) / 2;
  return rsim(l, r, f(l), f(r), f(mid), f((l + mid) / 2), f((mid + r) / 2));
}
```

6.6 Fast Fourier Transform

```
#define ld long double
#define ve vector

using namespace std;
```

```cpp
typedef ve<complex<ld>> vc;

const ld pi = acosl(-1);

const int LG = 22;
ve<complex<ld>> w[LG];

void fft(vc &a, int clg)
{
  int n = a.size();
  ve<int> dp(n, 0);
  for (int i = 1; i < n; i++)
    dp[i] = dp[i / 2] / 2 + (((i & 1) << (clg - 1)));
  for (int i = 0; i < n; i++)
    if (i < dp[i])
      swap(a[i], a[dp[i]]);
  for (int i = 1; i <= clg; i++)
  {
    int m = 1 << i;
    for (int j = 0; j < n; j += m)
      for (int k = 0; k < m / 2; k++)
      {
        auto x = a[j + k], y = a[j + k + m / 2];
        a[j + k] = x + w[i][k] * y;
        a[j + k + m / 2] = x - w[i][k] * y;
      }
  }
}

void ifft(vc &a, int clg)
{
  fft(a, clg);
  reverse(a.begin() + 1, a.end());
  for (auto &i : a)
    i /= a.size();
}

ve<ll> multiply(ve<int> a, ve<int> b)
{
  int n = 1, clg = 0;
  int m = a.size() + b.size() - 1;
  while (n < m)
  {
    n *= 2;
    clg++;
  }
  vc ca(n, 0);
  copy(all(a), ca.begin());
  for (int i = 0; i < b.size(); i++)
    ca[i] += complex<ld>(b[i]) * 1il;
  fft(ca, clg);
  for (int i = 0; i < n; i++)
    ca[i] *= ca[i];
  ifft(ca, clg);
  ve<ll> res(m);

  for (int i = 0; i < m; i++)
    res[i] = round(ca[i].imag() / 2);
  return res;
}
```

6.7 Number Theoretic Transform

$mod = c \cdot 2^k + 1 \rightarrow root = (primitive)^c$

```cpp
const int mod = 998244353;
const int root = 15311432;
const int root_1 = 469870224;
const int root_pw = 1 << 23;
void fft(vector<int> &a, bool invert) {
    int n = a.size();

    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1)
            j ^= bit;
        j ^= bit;

        if (i < j)
            swap(a[i], a[j]);
    }

    for (int len = 2; len <= n; len <<= 1) {
        int wlen = invert ? root_1 : root;
        for (int i = len; i < root_pw; i <<= 1)
            wlen = (int)(1LL * wlen * wlen % mod);

        for (int i = 0; i < n; i += len) {
            int w = 1;
            for (int j = 0; j < len / 2; j++) {
                int u = a[i + j], v = (int)(1LL * a[i + j + len / 2] * w % mod);
                a[i + j] = u + v < mod ? u + v : u + v - mod;
                a[i + j + len / 2] = u - v >= 0 ? u - v : u - v + mod;
                w = (int)(1LL * w * wlen % mod);
            }
        }
    }
```

```
33
34      if (invert) {
35          int n_1 = inverse(n, mod);
36          for (int &x : a)
37              x = (int)(1LL * x * n_1 % mod);
38      }
39  }
```

6.8 Prefix Sum of Multiplicative Function

```
1   Map Pi(ll n, VLL CR a) { // a = {n // 1, n // 2, n // 3, ..., 1}
2       Map res(n); // 2 arrays, s = sqrt(n): [0, s], and N / k \in [0, N / s + 1]
3       for (auto i : a)
4           res.at(i) = i - 1;
5
6       for (int i = 2; i <= res.s; i++) {
7           if (res.at(i) == res.at(i - 1))
8               continue;
9
10          ll f = res.at(i - 1);
11          for (auto j : a) {
12              if (j < (ll)i * i)
13                  break;
14
15              res.at(j) -= res.at(j / i) - f;
16          }
17      }
18      return res;
19  }
20
21  // cnt - global cnt for n // i, upto - precount on small primes
22  ll get(ll n, int ind) {
23      ll p = pr[ind];
24      if (p > n)
25          return 1;
26
27      if (p * p > n)
28          return (1 + (cnt.at(n) - upto[p] + 1) * pre[1]) % MOD;
29
30      __int128_t res = 0;
31      for (ll f = 1, c = 0; f <= n; f *= p, c++)
32          res += pre[c] * get(n / f, ind + 1);
33      return res % MOD;
34  }
```

7 FPS

$$\exp(x) = \sum_{i=0} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots$$

$$\ln(1+x) = \sum_{i=1!!!} \frac{(-1)^{i+1} x^i}{i} = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \dots$$

$$\sqrt{1+x} = 1 + \sum_{i=1!!!} \frac{(-1)^{i+1} \cdot C_{i-1} \cdot x^i}{2 \cdot 4^{i-1}} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \dots$$

$$\sin x = \sum_{i=0} \frac{(-1)^i \cdot x^{2i+1}}{(2i+1)!} = x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \dots$$

$$\cos x = \sum_{i=0} \frac{(-1)^i \cdot x^{2i}}{(2i)!} = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \dots$$

7.1 Newton's Iteration

for finding the root of $F(Q) = 0$, our initial guess $Q_0 = const$ should fulfill $F(Q_0) \equiv 0 \pmod{x}$:

$$Q_{k+1} \equiv Q_k - \frac{F(Q_k)}{F'(Q_k)} \pmod{x^{2a}}$$

Base functions:

$$\ln P(x) = \int (\ln P(x))' dx = \int \frac{P'(x)}{P(x)} dx = \int (P'(x) \cdot P^{-1}(x)) dx$$

$$A = P^{-1} \Rightarrow Q_{k+1} \equiv Q_k(2 - PQ_k) \pmod{x^{2a}}$$

$$A = e^{P(x)} \Rightarrow Q_{k+1} \equiv Q_k(1 + P - \ln Q_k) \pmod{x^{2a}}$$

7.2 Lagrange Inverse formula

Let $F(G(X)) = x$, $F(0) = G(0) = 0$, $[x^1]F(x) \neq 0$, $[x^1]G(x) \neq 0$, then:

$$[x^n]G(x) = \frac{1}{n}[x^{-1}]\frac{1}{F(x)^n}$$

$$[x^n]H(G(x)) = \frac{1}{n}[x^{-1}]H'(x)\frac{1}{F(x)^n}$$

8 Tips

8.1 Useful Codes

- Enumerate all non-empty subsets

```
for (int sub = mask; sub > 0; sub = (sub - 1) & mask)
```

- Enumerate C_n^k

```
for (int comb = (1 << k) - 1; comb < 1 << n; ) {
  // ...
  int x = comb & -comb, y = comb + x;
  comb = ((comb & ~y) / x >> 1) | y;
}
```

8.2 Formulas

8.2.1 Geometry

■ Euler's Formula

For convex polyhedron: $V - E + F = 2$.
For planar graph: $|F| = |E| - |V| + n + 1$, n denotes the number of connected components.

■ Pick's Theorem

$$S = I + \frac{B}{2} - 1$$

S is the area of lattice polygon, I is the number of lattice interior points, and B is the number of lattice boundary points.

■ Heron's Formula

$$S = \sqrt{p(p-a)(p-b)(p-c)}$$
$$p = \frac{a+b+c}{2}$$

■ Volumes

- Pyramid $V = \frac{1}{3}Sh$.
- Sphere $V = \frac{4}{3}\pi R^3$.
- Frustum $V = \frac{1}{3}h(S_1 + \sqrt{S_1 S_2} + S_2)$.
- Ellipsoid $V = \frac{4}{3}\pi abc$.

■ Radius of Inscribedcircle & Circumcircle

$$r = \frac{2S}{a+b+c},\ R = \frac{abc}{4S}$$

■ Hypersphere

$$V_2 = \pi R^2,\ S_2 = 2\pi R$$
$$V_3 = \frac{4}{3}\pi R^3,\ S_3 = 4\pi R^2$$

■ Matrix of rotating θ about arbitrary axis A

$$\begin{bmatrix} c+(1-c)A_x^2 & (1-c)A_xA_y - sA_z & (1-c)A_xA_z + sA_y \\ (1-c)A_xA_y + sA_z & c+(1-c)A_y^2 & (1-c)A_yA_z - sA_x \\ (1-c)A_xA_z - sA_y & (1-c)A_yA_z + sA_x & c+(1-c)A_z^2 \end{bmatrix}$$

8.2.2 Math

■ Sums

$$1 + 2 + \ldots + n = \frac{n^2}{2} + \frac{n}{2}$$
$$1^2 + 2^2 + \ldots + n^2 = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$$
$$1^3 + 2^3 + \ldots + n^3 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4}$$
$$1^4 + 2^4 + \ldots + n^4 = \frac{n^5}{5} + \frac{n^4}{2} + \frac{n^3}{3} - \frac{n}{30}$$
$$1^5 + 2^5 + \ldots + n^5 = \frac{n^6}{6} + \frac{n^5}{2} + \frac{5n^4}{12} - \frac{n^2}{12}$$
$$1^6 + 2^6 + \ldots + n^6 = \frac{n^7}{7} + \frac{n^6}{2} + \frac{n^5}{2} - \frac{n^3}{6} + \frac{n}{42}$$
$$\mathrm{P}(k) = \frac{(n+1)^{k+1} - \sum_{i=0}^{k-1} \binom{k+1}{i} \mathrm{P}(i)}{k+1}, \mathrm{P}(0) = n\mathbf{+1}$$

$$\sum_{k=1}^{n} k(k+1) = \frac{n(n+1)(n+2)}{3}$$
$$\sum_{k=1}^{n} k(k+1)(k+2) = \frac{n(n+1)(n+2)(n+3)}{4}$$
$$\sum_{k=1}^{n} k(k+1)(k+2)(k+3) = \frac{n(n+1)(n+2)(n+3)(n+4)}{5}$$

■ Burnside's Lemma

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g|$$

$$\text{Polya} : X^g = t^{c(g)}$$

Let X^g denote the set of elements in X fixed by g.
$c(g)$ is the number of cycles of the group element g as a permutation of X.

■ Lucas' Theorem

For non-negative integers m and n and a prime p, holds the equation

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

where $m = m_k p^k + m_{k-1} p^{k-1} + \ldots + m_1 p + m_0$, and $n = n_k p^k + n_{k-1} p^{k-1} + \ldots + n_1 p + n_0$, are the base p expansions of m and n respectively.

■ Wilson's Theorem

p is a prime $\iff (p-1)! \equiv -1 \pmod p$.

■ Polynomial Congruence Equation

Solve the polynomial congruence equation $f(x) \equiv 0 \mod m$, $m = \prod_{i=1}^{k} p_i^{a_i}$.
We just simply consider the equation $f(x) \equiv 0 \mod p^a$, then use the Chinese Remainder theorem to merge the result.
If x is the root of the equation $f(x) \equiv 0 \mod p^a$, then x is also the root of $f(x) \equiv 0 \mod p^{a-1}$.
$f'(x') \equiv 0 \mod p$ **and** $f(x') \equiv 0 \mod p^a \Rightarrow x = x' + dp^{a-1}(d = 0, \ldots, p-1)$
$f'(x') \not\equiv 0 \mod p \Rightarrow x = x' - \frac{f(x')}{f'(x')}$

■ Binomial Coefficients

$$\mathrm{C}_r^k = \frac{r}{k}\mathrm{C}_{r-1}^{k-1} \qquad \mathrm{C}_r^k = \mathrm{C}_{r-1}^k + \mathrm{C}_{r-1}^{k-1}$$
$$\mathrm{C}_r^m \mathrm{C}_m^k = \mathrm{C}_r^k \mathrm{C}_{r-k}^{m-k} \qquad \sum_{k \le n} \mathrm{C}_{r+k}^k = \mathrm{C}_{r+n+1}^n$$
$$\sum_{0 \le k \le n} \mathrm{C}_k^m = \mathrm{C}_{n+1}^{m+1} \qquad \sum_k \mathrm{C}_r^k \mathrm{C}_s^{n-k} = \mathrm{C}_{r+s}^n$$

The number of non-negative solutions to equation $x_1 + x_2 + x_3 + \ldots + x_k = n$ is $\binom{n+k-1}{k-1}$.

■ Catalan Number

The number of sequences with 1 of m and -1 of n, and m$\geq n$, satifisying the constraint that any sum of the first k elements is always non-negative: $\mathrm{C}_{m+n}^m - \mathrm{C}_{m+n}^{m+1}$.
Specially, when $m = n$, it equals to $\mathrm{C}_n = \frac{\mathrm{C}_{2n}^n}{n+1}$, which is the Catalan number.
The first 10 Catalan numbers are 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, from $n = 1$, and $C_0 = 1$.
Besides, $C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$, for $n \geq 0$.

■ Stirling Number of 2nd

The Stirling number of the second kind is the number of ways to partition a set of n objects into k non-empty subsets, denoted by $S(n, k)$.
$S(n,k) = kS(n-1,k) + S(n-1,k-1)$, and $S(0,0) = 1$, $S(n,0) = 0$, $S(n,n) = 1$.

■ Bell Number

The Bell Number is the number of ways to partition a set of n objects into several subsets, denoted by B_n.
The first few Bell Numbers are 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975.
$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$, $B_n = \sum_{k=0}^{n} S(n,k)$, where $S(n,k)$ is Stirling Number of 2nd.
If p is a prime then, $B_{p+n} \equiv B_n + B_{n+1} (mod\ p)$, $B_{p^m+n} \equiv mB_n + B_{n+1} (mod\ p)$.

■ Derangement Number

The number of permutations of n elements with no fixed points, denotes as D_n.
$D_n = n!(1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \ldots + (-1)^n \frac{1}{n!})$, or $D_n = n * D_{n-1} + (-1)^n, D_n = (n-1) * (D_{n-1} + D_{n-2})$, with $D_1 = 1, D_2 = 1$. The first 10 derangement numbers are 0, 1, 2, 9, 44, 265, 1854, 14833, 133496, 41334961, from $n = 1$.

8.3 Compiler

```
gcc -g -O0 -pg -fsanitize=address,undefined -fstack-protector-strong -finstrument-functions -o your_program your_program.c
```

8.4 Lagrange

$A(x) = \sum_{i=1}^{n} y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$ - $O(n^2)$.
D&Q $O(n \log^2(n))$: $A_{l,r} = A_{l,m} \cdot P_{m+1,r} + A_{m+1,r} \cdot P_{l,m}$; $P_{l,r} = P_{l,m} \cdot P_{m+1,r}$