Contents

1 Build and Snippet

1.1 Sublime Build

```
#for linux
 {
   "shell_cmd": "g++ $file -o $file_base_name && ./
   $file_base_name<input.txt> output.txt && rm
   $file_base_name",
   "working_dir": "$file_path",
   "selector": "source.c++"
 }
```

2 Data Structures

2.1 2D BIT

```
const int N = 1008; int bit[N][N], a[N][N], n, m, q;
void update(int x, int y, int val) { for (; x < N; x +=
-x & x) for (int j = y; j < N; j += -j & j) bit[x][j] +=
 val; } int get(int x, int y) { int ans = 0; for (; x; x
 -= x & -x) for (int j = y; j; j -= j & -j) ans += bit[x
][j]; return ans; } int get(int x1, int y1, int x2, int
y2) { return get(x2, y2) - get(x1 - 1, y2) - get(x2, y1
- 1) + get(x1 - 1, y1 - 1); }
```

2.2 BIT

```
class BIT { int *bin, N; public: BIT(int N) : N(N) { bin
 = new int[N + 1]; memset(bin, 0, (N + 1) * sizeof(int))
; } void update(int id, int val) { for (; id <= N; id +=
 id & -id) bin[id] += val; } int helper(int id) { int
sum = 0; for (; id > 0; id -= id & -id) sum += bin[id];
return sum; } int query(int l, int r) { return helper(r)
 - helper(l - 1); } ~BIT() { delete[] bin; } };
```

2.3 Lazy Propagation

```
class LazySegmentTree { vector<int> tree; vector<int>
lazy; int n, I; int merge(int a, int b) { return a + b;
} void propagate(int node, int le, int ri) { if (lazy[
node] != 0) { tree[node] += (ri - le + 1) * lazy[node];
if (le != ri) { lazy[2 * node + 1] += lazy[node]; lazy[2
 * node + 2] += lazy[node]; } lazy[node] = 0; } } void
build(int node, int le, int ri, vector<int> &arr) { if (
le == ri) { tree[node] = arr[le]; return; } int mid = (
le + ri) / 2; build(2 * node + 1, le, mid, arr); build(2
 * node + 2, mid + 1, ri, arr); tree[node] = merge(tree
[2 * node + 1] , tree[2 * node + 2]); } void update(int
node, int le, int ri, int l, int r, int val) { propagate
(node, le, ri); if (r < le || l > ri) { return; } if (le
 >= l && ri <= r) { tree[node] += (ri - le + 1) * val;
if (le != ri) { lazy[2 * node + 1] += val; lazy[2 * node
 + 2] += val; } return; } int mid = (le + ri) / 2;
update(2 * node + 1, le, mid, l, r, val); update(2 *
node + 2, mid + 1, ri, l, r, val); tree[node] = merge(
tree[2 * node + 1] , tree[2 * node + 2]); } int query(
int node, int le, int ri, int l, int r) { propagate(node
, le, ri); if (l <= le && r >= ri) { return tree[node];
} if (r < le || l > ri) { return I; } int mid = (le + ri
) / 2; return merge(query(2 * node + 1, le, mid, l, r) ,
 query(2 * node + 2, mid + 1, ri, l, r)); } public:
LazySegmentTree(vector<int> &arr, int I) { n = arr.size
(); this->I = I; tree.resize(4 * n, 0); lazy.resize(4 *
n, 0); build(0, 0, n - 1, arr); } void update(int l, int
 r, int val) { update(0, 0, n - 1, l, r, val); } int
query(int l, int r) { return query(0, 0, n - 1, l, r); }
 };
```

2.4 MO

```
struct Query { int l, r, idx; bool operator<(const Query
 &other) const { if (l / BLOCK != other.l / BLOCK) {
return l / BLOCK < other.l / BLOCK; } return r < other.r
; } }; vector<bool> mo_algorithm(vector<Query> &queries,
 int n, int q) { sort(queries.begin(), queries.end());
int currl = 0, curr_r = -1; vector<bool> result(q); for
(auto &query : queries) { int l = query.l, r = query.r,
idx = query.idx; while (curr_r < r) { add(++curr_r); }
while (curr_r > r) { remove(curr_r--); } while (currl <
l) { remove(currl++); } while (currl > l) { add(--currl)
; } } return result; }
```

2.5 MergeSortTree

```
class MergeSortTree { int n; vector<vector<int>> tree;
void build(int id, int le, int ri, vector<int> &a) { if
(le == ri) { tree[id].push_back(a[le]); return; } int
mid = (le + ri) >> 1; build(2 * id + 1, le, mid, a);
build(2 * id + 2, mid + 1, ri, a); auto &left = tree[2 *
 id + 1], &right = tree[2 * id + 2]; int i = 0, j = 0, n
 = left.size(), m = right.size(); while (i < n && j < m)
 { if (left[i] < right[j]) tree[id].push_back(left[i]),
i++; else tree[id].push_back(right[j]), j++; } while (i
< n) tree[id].push_back(left[i]), i++; while (j < m)
tree[id].push_back(right[j]), j++; } /* number of
element greater than val */ int queryL(int id, int le,
int ri, int l, int r, int val) { if (le > r || ri < l) {
 return 0; } if (le >= l && ri <= r) { return ri - le +
1 - (upper_bound(tree[id].begin(), tree[id].end(), val)
- tree[id].begin()); } int mid = (le + ri) >> 1; return
queryL(2 * id + 1, le, mid, l, r, val) + queryL(2 * id +
 2, mid + 1, ri, l, r, val); } /* number of element
smaller than val */ int queryS(int id, int le, int ri,
int l, int r, int val) { if (le > r || ri < l) { return
0; } if (le >= l && ri <= r) { return (upper_bound(tree[
id].begin(), tree[id].end(), val - 1) - tree[id].begin()
); } int mid = (le + ri) >> 1; return queryS(2 * id + 1,
 le, mid, l, r, val) + queryS(2 * id + 2, mid + 1, ri, l
, r, val); } public: MergeSortTree(vector<int> &a) { n =
 a.size(); tree.resize(n * 4); build(0, 0, n - 1, a); }
int queryS(int l, int r, int val) { return queryS(0, 0,
n - 1, l, r, val); } int queryL(int l, int r, int val) {
 return queryL(0, 0, n - 1, l, r, val); } };
```

2.6 PST

```
class PST{ #define lc(u) tree[u].left #define rc(u) tree
[u].right; struct node{ int left = 0, right = 0, val =
0; }; node *tree; int N, LG, time = 0, I = 0; node
create(int l, int r){ return {l, r, merge(tree[l].val,
tree[r].val)}; } int merge(LL a, LL b){ return a + b; }
int build(int le, int ri){ int id = ++time; if(le == ri)
 return tree[id] = node(), id; int m = (le + ri) / 2;
return tree[id] = create(build(le, m), build(m + 1, ri))
, id; } int update(int id, int le, int ri, int pos, int
val){ int nid = ++time; if(le == ri) return tree[nid] =
{0, 0, (tree[id].val + val)}, nid; /* // change here */
int m = (le + ri) / 2; if(pos <= m){ tree[nid] = create(
update(tree[id].left, le, m, pos, val), tree[id].right);
 }else{ tree[nid] = create(tree[id].left, update(tree[id
].right, m + 1, ri, pos, val)); } return nid; } int
query(int id, int le, int ri, int l, int r){ if(r < le
|| ri < l) return 0; if(l <= le && ri <= r) return tree[
id].val; int m = (le + ri) >> 1; return query(tree[id].
left, le, m, l, r) + query(tree[id].right, m + 1, ri, l,
 r); } public: PST(int N, int U){ /*U --> number of
expected updates */ this->N = N; LG = 33 - __builtin_clz
(N); tree = new node[N * 4 + U * LG]; build(0, N - 1); }
 int update(int id, int pos, int val){ return update(id,
 0, N - 1, pos, val); } int query(int id, int l, int r){
 if(l > r) return 0; return query(id, 0, N - 1, l, r); }
 ~PST(){ delete[] tree; } };
```

2.7 SegmentTree

```
template <typename DT> class segmentTree { DT *seg, I;
int n; DT (*merge)(DT, DT); void build(int idx, int le,
int ri, vector<DT> &v) { if (le == ri) { seg[idx] = v[le
]; return; } int mid = (le + ri) >> 1; build(2 * idx +
1, le, mid, v); build(2 * idx + 2, mid + 1, ri, v); seg[
idx] = merge(seg[2 * idx + 1], seg[2 * idx + 2]); } void
 update(int idx, int le, int ri, int pos, DT val) { if (
le == ri) { seg[idx] = val; return; } int mid = (le + ri
) >> 1; if (pos <= mid) update(2 * idx + 1, le, mid, pos
, val); else update(2 * idx + 2, mid + 1, ri, pos, val);
 seg[idx] = merge(seg[2 * idx + 1], seg[2 * idx + 2]); }
 DT query(int idx, int le, int ri, int l, int r) { if (l
 <= le && r >= ri) { return seg[idx]; } if (r < le || l
> ri) { return I; } int mid = (le + ri) >> 1; return
merge(query(2 * idx + 1, le, mid, l, r), query(2 * idx +
 2, mid + 1, ri, l, r)); } /* // finding the leftmost
appearence of value <= val in [l....r] range // need
minimum segment tree */ int walk(int idx, int le, int ri
, int l, int r, DT val) { if (r < le || l > ri) { return
 r; } if (le == ri) { if (seg[idx] <= val) return le;
return r; } if (l <= le && r >= ri) { int mid = (le + ri
) >> 1; if (seg[2 * idx + 1] <= val) return walk(2 * idx
 + 1, le, mid, l, r, val); return walk(2 * idx + 2, mid
+ 1, ri, l, r, val); } int mid = (le + ri) >> 1; return
merge(walk(2 * idx + 1, le, mid, l, r, val), walk(2 *
idx + 2, mid + 1, ri, l, r, val)); } public: segmentTree
() {} segmentTree(vector<DT> &v, DT (*fptr)(DT, DT), DT
_I) { n = v.size(); I = _I; merge = fptr; seg = new DT[4
 * n]; build(0, 0, n - 1, v); } void update(int pos, DT
val) { update(0, 0, n - 1, pos, val); } int walk(int l,
int r, DT val) { if (query(l, r) > val) return r; return
 walk(0, 0, n - 1, l, r, val); } DT query(int l, int r)
{ return query(0, 0, n - 1, l, r); } ~segmentTree(){
delete[] seg; } }; int fun(int a, int b) { return max(a,
 b); }
```

2.8 SegmentTreeBeats

```
class SegTreeBeats { const int INF = INT_MAX; const LL
NEG_INF = LLONG_MIN; vector<LL> mx, mn, smx, smn, sum,
add; vector<int> mxcnt, mncnt; int L, R; void applyMax(
int u, LL x) { sum[u] += mncnt[u] * (x - mn[u]); if (mx[
u] == mn[u]) mx[u] = x; if (smx[u] == mn[u]) smx[u] = x;
 mn[u] = x; } void applyMin(int u, LL x) { sum[u] -=
mxcnt[u] * (mx[u] - x); if (mn[u] == mx[u]) mn[u] = x;
if (smn[u] == mx[u]) smn[u] = x; mx[u] = x; } void
applyAdd(int u, LL x, int tl, int tr) { sum[u] += (tr -
tl + 1) * x; add[u] += x; mx[u] += x, mn[u] += x; if (
smx[u] != NEG_INF) smx[u] += x; if (smn[u] != INF) smn[u
] += x; } void push(int u, int tl, int tr) { int lft = u
 << 1, ryt = lft | 1, mid = (tl + tr) >> 1; if (add[u]
!= 0) { applyAdd(lft, add[u], tl, mid); applyAdd(ryt,
add[u], mid + 1, tr); add[u] = 0; } if (mx[u] < mx[lft])
 applyMin(lft, mx[u]); if (mx[u] < mx[ryt]) applyMin(ryt
, mx[u]); if (mn[u] > mn[lft]) applyMax(lft, mn[u]); if
(mn[u] > mn[ryt]) applyMax(ryt, mn[u]); } void merge(int
 u) { int lft = u << 1, ryt = lft | 1; sum[u] = sum[lft]
 + sum[ryt]; mx[u] = max(mx[lft], mx[ryt]); smx[u] = max
(smx[lft], smx[ryt]); if (mx[lft] != mx[ryt]) smx[u] =
max(smx[u], min(mx[lft], mx[ryt])); mxcnt[u] = (mx[u] ==
 mx[lft]) * mxcnt[lft] + (mx[u] == mx[ryt]) * mxcnt[ryt
]; mn[u] = min(mn[lft], mn[ryt]); smn[u] = min(smn[lft],
 smn[ryt]); if (mn[lft] != mn[ryt]) smn[u] = min(smn[u],
 max(mn[lft], mn[ryt])); mncnt[u] = (mn[u] == mn[lft]) *
 mncnt[lft] + (mn[u] == mn[ryt]) * mncnt[ryt]; } void
build(const vector<int> &a, int tl, int tr, int u) { if
(tl == tr) { sum[u] = mn[u] = mx[u] = a[tl]; mxcnt[u] =
mncnt[u] = 1; smx[u] = NEG_INF; smn[u] = INF; return; }
int mid = (tl + tr) >> 1, lft = u << 1, ryt = lft | 1;
build(a, tl, mid, lft); build(a, mid + 1, tr, ryt);
merge(u); } public: SegTreeBeats(const vector<int> &a) {
 int n = a.size(); L = 0; R = n - 1; mx.resize(4 * n, 0)
; mn.resize(4 * n, 0); smx.resize(4 * n, NEG_INF); smn.
resize(4 * n, INF); sum.resize(4 * n, 0); add.resize(4 *
 n, 0); mxcnt.resize(4 * n, 0); mncnt.resize(4 * n, 0);
build(a, L, R, 1); } /* // a[i] = min(x, a[i]); */ void
minimize(int l, int r, LL x) { minimize(l, r, x, L, R,
1); } /* // a[i] = max(x, a[i]); */ void maximize(int l,
 int r, LL x) { maximize(l, r, x, L, R, 1); } /* // a[i]
 = a[i] + x; */ void increase(int l, int r, LL x) {
increase(l, r, x, L, R, 1); } LL getSum(int l, int r) {
return getSum(l, r, L, R, 1); } private: void minimize(
int l, int r, LL x, int tl, int tr, int u) { if (l > tr
|| tl > r || mx[u] <= x) return; if (l <= tl && tr <= r
&& smx[u] < x) { applyMin(u, x); return; } push(u, tl,
tr); int mid = (tl + tr) >> 1, lft = u << 1, ryt = lft |
 1; minimize(l, r, x, tl, mid, lft); minimize(l, r, x,
mid + 1, tr, ryt); merge(u); } void maximize(int l, int
r, LL x, int tl, int tr, int u) { if (l > tr || tl > r
|| mn[u] >= x) return; if (l <= tl && tr <= r && smn[u]
> x) { applyMax(u, x); return; } push(u, tl, tr); int
mid = (tl + tr) >> 1, lft = u << 1, ryt = lft | 1;
maximize(l, r, x, tl, mid, lft); maximize(l, r, x, mid +
 1, tr, ryt); merge(u); } void increase(int l, int r, LL
 x, int tl, int tr, int u) { if (l > tr || tl > r)
return; if (l <= tl && tr <= r) { applyAdd(u, x, tl, tr)
; return; } push(u, tl, tr); int mid = (tl + tr) >> 1,
lft = u << 1, ryt = lft | 1; increase(l, r, x, tl, mid,
lft); increase(l, r, x, mid + 1, tr, ryt); merge(u); }
LL getSum(int l, int r, int tl, int tr, int u) { if (l >
 tr || tl > r) return 0; if (l <= tl && tr <= r) return
sum[u]; push(u, tl, tr); int mid = (tl + tr) >> 1, lft =
 (u << 1), ryt = (lft | 1); LL x = getSum(l, r, tl, mid,
 lft), y = getSum(l, r, mid + 1, tr, ryt); return x + y;
 } };
```

2.9 Sparse Table

```
class SparseTable { private: vector<vector<int>> table;
vector<int> log; int n; public: SparseTable(const vector
<int>& arr) { n = arr.size(); log.resize(n + 1);
buildLog(); table = vector<vector<int>>(n, vector<int>(
log[n] + 1)); for (int i = 0; i < n; i++) { table[i][0]
= arr[i]; } for (int j = 1; (1 << j) <= n; j++) { for (
int i = 0; i + (1 << j) <= n; i++) { table[i][j] = merge
(table[i][j - 1], table[i + (1 << (j - 1))][j - 1]); } }
 } int merge(int a, int b) { return max(a, b); } void
buildLog() { log[1] = 0; for (int i = 2; i <= n; i++)
log[i] = log[i / 2] + 1; } int Query(int L, int R) { int
 j = log[R - L + 1]; return merge(table[L][j], table[R -
 (1 << j) + 1][j]); } int query(int L, int R) { int sum
= 0; for (int j = log[R - L + 1]; L <= R; j = log[R - L
+ 1]) { sum = merge(sum, table[L][j]); L += (1 << j); }
return sum; } };
```

2.10 Trie

```
struct node{ int path, leaf; vector<int> child; node(int
 n = 0) : child(n, -1), path(0), leaf(0) {} }; class
Trie{ int n, ptr; vector<node> tree; public: Trie(int n)
 : n(n), ptr(0){ tree.emplace_back(node(n)); } void
insert(string &s){ int cur = 0; for(auto u: s){ int &
next = tree[cur].child[u - '0']; if(next == -1){ tree.
emplace_back(node(n)); next = ++ptr; } tree[cur].path++;
 cur = next; } tree[cur].path++; tree[cur].leaf++; }
void erase(string &s){ int cur = 0; for(auto u: s){ tree
[cur].path--; cur = tree[cur].child[u - '0']; } tree[cur
].path--; tree[cur].leaf--; } bool find(string &s){ int
cur = 0; for(auto u: s){ cur = tree[cur].child[u - '0'];
 if(cur == -1 || !tree[cur].path) return 0; } return
tree[cur].leaf > 0; } };
```

2.11 sparse table 2D

```
/* // rectangle query */ namespace st2 { const int N = 2
e3 + 5, B = 12; using Ti = long long; Ti Id = LLONG_MAX;
 Ti f(Ti a, Ti b) { return max(a, b); } Ti tbl[N][N][B];
 void init(int n, int m) { for(int k = 1; k < B; k++) {
for(int i = 0; i + (1 << k) - 1 < n; i++) { for(int j =
0; j + (1 << k) - 1 < m; j++) { tbl[i][j][k] = tbl[i][j
][k - 1]; tbl[i][j][k] = f(tbl[i][j][k], tbl[i][j + (1
<< k - 1)][k - 1]); tbl[i][j][k] = f(tbl[i][j][k], tbl[i
 + (1 << k - 1)][j][k - 1]); tbl[i][j][k] = f(tbl[i][j][
k], tbl[i + (1 << k - 1)][j + (1 << k - 1)][k - 1]); } }
 } } Ti query(int i, int j, int len) { int k = __lg(len)
; LL ret = tbl[i][j][k]; ret = f(ret, tbl[i + len - (1
<< k)][j][k]); ret = f(ret, tbl[i][j + len - (1 << k)][k
]); ret = f(ret, tbl[i + len - (1 << k)][j + len - (1 <<
 k)][k]); return ret; } } int main() { for(int i = 0; i
< n; i++) for(int j = 0; j < m; j++) cin >> st2 :: tbl[i
][j][0]; st2 :: init(n, m); cout << st2 :: query(x, y, s
); // x, y, x + s - 1, y + s - 1 }
```

3 Number Theory

3.1 2D FFT

```
const int N = 1 << 13; const int mod = 998244353; const
int root = 3; using Mat = vector<vector<int>>; int lim,
rev[N], w[N], wn[N], inv_lim; void reduce(int &x) { x =
(x + mod) % mod; } int POW(int x, int y, int ans = 1) {
for (; y; y >>= 1, x = (long long) x * x % mod) if (y &
1) ans = (long long) ans * x % mod; return ans; } void
precompute(int len) { lim = wn[0] = 1; int s = -1; while
 (lim < len) lim <<= 1, ++s; for (int i = 0; i < lim; ++
i) rev[i] = rev[i >> 1] >> 1 | (i & 1) << s; const int g
 = POW(root, (mod - 1) / lim); inv_lim = POW(lim, mod -
2); for (int i = 1; i < lim; ++i) wn[i] = (long long) wn
[i - 1] * g % mod; } void ntt(vector<int> &a, int typ) {
 for (int i = 0; i < lim; ++i) if (i < rev[i]) swap(a[i
], a[rev[i]]); for (int i = 1; i < lim; i <<= 1) { for (
int j = 0, t = lim / i / 2; j < i; ++j) w[j] = wn[j * t
]; for (int j = 0; j < lim; j += i << 1) { for (int k =
0; k < i; ++k) { const int x = a[k + j], y = (long long)
 a[k + j + i] * w[k] % mod; reduce(a[k + j] += y - mod),
 reduce(a[k + j + i] = x - y); } } } if (!typ) { reverse
(a.begin() + 1, a.begin() + lim); for (int i = 0; i <
lim; ++i) a[i] = (long long) a[i] * inv_lim % mod; } }
/* // a is of size n * n // b is of size m * m // max(n,
 m)^2 * log(max(n, m)); */ Mat multiply(Mat a, Mat b) {
int n = a.size(), m = b.size(); int len = n + m - 1;
precompute(len); a.resize(lim); for (int i = 0; i < lim;
 i++) { a[i].resize(lim, 0); } b.resize(lim); for (int i
 = 0; i < lim; i++) { b[i].resize(lim, 0); } /* //
convert rows to point value form */ for (int i = 0; i <
lim; i++) { ntt(a[i], 1); ntt(b[i], 1); } Mat ans(lim,
vector<int> (lim, 0)); for (int j = 0; j < lim; j++) {
vector<int> col_a(lim), col_b(lim); for (int i = 0; i <
lim; i++) { col_a[i] = a[i][j]; col_b[i] = b[i][j]; } /*
 // convert columns to point value form */ ntt(col_a, 1)
; ntt(col_b, 1); /* // so everything is in point value
form, // so compute the product easily */ for (int i =
0; i < lim; i++) { col_a[i] = 1LL * col_a[i] * col_b[i]
% mod; } /* // inverse fft on columns */ ntt(col_a, 0);
for (int i = 0; i < lim; i++) { a[i][j] = col_a[i]; } }
/* // inverse fft on rows */ for (int i = 0; i < lim; i
++) { ntt(a[i], 0); } a.resize(n + m - 1); for (int i =
0; i < n + m - 1; i++) { a[i].resize(n + m - 1); }
return a; } Mat multiply_brute(Mat a, Mat b) { int n = a
.size(), m = b.size(); Mat ans(n + m - 1, vector<int> (n
 + m - 1, 0)); for (int i = 0; i < n; i++) { for (int j
= 0; j < n; j++) { for (int r = 0; r < m; r++) { for (
int c = 0; c < m; c++) { ans[i + r][j + c] += 1LL * a[i
][j] * b[r][c] % mod; ans[i + r][j + c] %= mod; } } } }
return ans; }
```

3.2 Bitwise Sieve

```
const int nmax = 1e8 + 1; int mark[(nmax >> 6) + 1];
vector<int> primes; #define isSet(n, pos) (bool)((n) &
(1LL << (pos))) #define Set(n, pos) ((n) | (1LL << (pos)
)) void sieve(int n) { for (int i = 3; i * i <= n; i +=
2) { if (isSet(mark[i >> 6], (i >> 1) & 31) == 0) { for
(int j = i * i; j <= n; j += (i << 1)) mark[j >> 6] =
Set(mark[j >> 6], (j >> 1) & 31); } } primes.push_back
(2); for (int i = 3; i <= n; i += 2) { if (isSet(mark[i
>> 6], (i >> 1) & 31) == 0) primes.push_back(i); } }
```

3.3 Chinese Reminder Theorem

```
/* // given a, b will find solutions for, ax + by = 1 */
 tuple<LL, LL, LL> EGCD(LL a, LL b) { if (b == 0) return
 {1, 0, a}; else { auto [x, y, g] = EGCD(b, a % b);
return {y, x - a / b * y, g}; } } /* // given modulo
equations, will apply CRT */ PLL CRT(vector<PLL> &v) {
LL V = 0, M = 1; for (auto &[v, m] : v) { /* // value %
mod */ auto [x, y, g] = EGCD(M, m); if ((v - V) % g !=
0) return {-1, 0}; V += x * (v - V) / g % (m / g) * M, M
 *= m / g; V = (V % M + M) % M; } return make_pair(V, M)
; }
```

3.4 Divisor

```
/* // calculate divisor in range[1,n] */ LL sum_in_range
(LL n) { return n * (n + 1) / 2; } LL sum_all_divisors(
LL n) { LL ans = 0; for(LL i=1;i*i<=n;i++) { LL hello =
i * (n / i - i + 1); LL world = sum_in_range(n / i) -
sum_in_range(i); ans += hello + world; } return ans; }
```

3.5 Eulars Totient Function

```
void phi_in_range() { int N = 1e6, phi[N + 1]; for (int
i = 0; i <= N; i++) phi[i] = i; for (int i = 2; i <= N;
i++) { if (phi[i] != i) continue; for (int j = i; j <= N
; j += i) { phi[j] -= phi[j] / i; } } } #some important
properties of phi phi(a*b) = phi(a)*phi(b)*(gcd(a,b)/phi
(gcd(a,b))) phi(p^k) = p^k - p^(k-1) ,where p is a prime
 number SUM{phi(d)} = n, d|n
```

3.6 FFT

```
using CD = complex <double>; typedef long long LL; const
 double PI = acos(-1.0L); int N; vector<int> perm;
vector<CD> wp[2]; void precalculate(int n) { assert((n &
 (n - 1)) == 0), N = n; perm = vector<int>(N, 0); for (
int k = 1; k < N; k <<= 1) { for (int i = 0; i < k; i++)
 { perm[i] <<= 1; perm[i + k] = 1 + perm[i]; } } wp[0] =
 wp[1] = vector<CD>(N); for (int i = 0; i < N; i++) { wp
[0][i] = CD(cos(2 * PI * i / N), sin(2 * PI * i / N));
wp[1][i] = CD(cos(2 * PI * i / N), -sin(2 * PI * i / N))
; } } void fft(vector<CD> &v, bool invert = false) { if
(v.size() != perm.size()) precalculate(v.size()); for (
int i = 0; i < N; i++) if (i < perm[i]) swap(v[i], v[
perm[i]]); for (int len = 2; len <= N; len *= 2) { for (
int i = 0, d = N / len; i < N; i += len) { for (int j =
0, idx = 0; j < len / 2; j++, idx += d) { CD x = v[i + j
]; CD y = wp[invert][idx] * v[i + j + len / 2]; v[i + j]
 = x + y; v[i + j + len / 2] = x - y; } } } if (invert)
{ for (int i = 0; i < N; i++) v[i] /= N; } } void
pairfft(vector<CD> &a, vector<CD> &b, bool invert =
false) { int N = a.size(); vector<CD> p(N); for (int i =
 0; i < N; i++) p[i] = a[i] + b[i] * CD(0, 1); fft(p,
invert); p.push_back(p[0]); for (int i = 0; i < N; i++)
{ if (invert) { a[i] = CD(p[i].real(), 0); b[i] = CD(p[i
].imag(), 0); } else { a[i] = (p[i] + conj(p[N - i])) *
CD(0.5, 0); b[i] = (p[i] - conj(p[N - i])) * CD(0, -0.5)
; } } } vector<LL> multiply(const vector<LL> &a, const
```

```
vector<LL> &b) { int n = 1; while (n < a.size() + b.size
()) n <<= 1; vector<CD> fa(a.begin(), a.end()), fb(b.
begin(), b.end()); fa.resize(n); fb.resize(n); /* // fft
(fa); fft(fb); */ pairfft(fa, fb); for (int i = 0; i < n
; i++) fa[i] = fa[i] * fb[i]; fft(fa, true); vector<LL>
ans(n); for (int i = 0; i < n; i++) ans[i] = round(fa[i
].real()); return ans; } const int M = 1e9 + 7, B = sqrt
(M) + 1; vector<LL> anyMod(const vector<LL> &a, const
vector<LL> &b) { int n = 1; while (n < a.size() + b.size
()) n <<= 1; vector<CD> al(n), ar(n), bl(n), br(n); for
(int i = 0; i < a.size(); i++) al[i] = a[i] % M / B, ar[
i] = a[i] % M % B; for (int i = 0; i < b.size(); i++) bl
[i] = b[i] % M / B, br[i] = b[i] % M % B; pairfft(al, ar
); pairfft(bl, br); /* // fft(al); fft(ar); fft(bl); fft
(br); */ for (int i = 0; i < n; i++) { CD ll = (al[i] *
bl[i]), lr = (al[i] * br[i]); CD rl = (ar[i] * bl[i]),
rr = (ar[i] * br[i]); al[i] = ll; ar[i] = lr; bl[i] = rl
; br[i] = rr; } pairfft(al, ar, true); pairfft(bl, br,
true); /* // fft(al, true); fft(ar, true); fft(bl, true)
; fft(br, true); */ vector<LL> ans(n); for (int i = 0; i
 < n; i++) { LL right = round(br[i].real()), left =
round(al[i].real()); ; LL mid = round(round(bl[i].real()
) + round(ar[i].real())); ans[i] = ((left % M) * B * B +
 (mid % M) * B + right) % M; } return ans; }
```

3.7 FWHT

```
#include<bits/stdc++.h> using namespace std; const int N
 = 3e5 + 9, mod = 1e9 + 7; int POW(long long n, long
long k) { int ans = 1 % mod; n %= mod; if (n < 0) n +=
mod; while (k) { if (k & 1) ans = (long long) ans * n %
mod; n = (long long) n * n % mod; k >>= 1; } return ans;
 } const int inv2 = (mod + 1) >> 1; #define M (1 << 20)
#define OR 0 #define AND 1 #define XOR 2 struct FWHT{
int P1[M], P2[M]; void wt(int *a, int n, int flag = XOR)
 { if (n == 0) return; int m = n / 2; wt(a, m, flag); wt
(a + m, m, flag); for (int i = 0; i < m; i++){ int x = a
[i], y = a[i + m]; if (flag == OR) a[i] = x, a[i + m] =
(x + y) % mod; if (flag == AND) a[i] = (x + y) % mod, a[
i + m] = y; if (flag == XOR) a[i] = (x + y) % mod, a[i +
 m] = (x - y + mod) % mod; } } void iwt(int* a, int n,
int flag = XOR) { if (n == 0) return; int m = n / 2; iwt
(a, m, flag); iwt(a + m, m, flag); for (int i = 0; i < m
; i++){ int x = a[i], y = a[i + m]; if (flag == OR) a[i]
 = x, a[i + m] = (y - x + mod) % mod; if (flag == AND) a
[i] = (x - y + mod) % mod, a[i + m] = y; if (flag == XOR
) a[i] = 1LL * (x + y) * inv2 % mod, a[i + m] = 1LL * (x
 - y + mod) * inv2 % mod; /* // replace inv2 by >>1 if
not required */ } } vector<int> multiply(int n, vector<
int> A, vector<int> B, int flag = XOR) { assert(
__builtin_popcount(n) == 1); A.resize(n); B.resize(n);
for (int i = 0; i < n; i++) P1[i] = A[i]; for (int i =
0; i < n; i++) P2[i] = B[i]; wt(P1, n, flag); wt(P2, n,
flag); for (int i = 0; i < n; i++) P1[i] = 1LL * P1[i] *
 P2[i] % mod; iwt(P1, n, flag); return vector<int> (P1,
P1 + n); } vector<int> pow(int n, vector<int> A, long
long k, int flag = XOR) { assert(__builtin_popcount(n)
== 1); A.resize(n); for (int i = 0; i < n; i++) P1[i] =
A[i]; wt(P1, n, flag); for(int i = 0; i < n; i++) P1[i]
= POW(P1[i], k); iwt(P1, n, flag); return vector<int> (
P1, P1 + n); } }t;
```

3.8 GCD convolution

```
template<int MOD> struct ModInt { private: int r; static
 ModInt Pow(ModInt x, size_t n) { ModInt ret = 1; for (;
 n; n >>= 1, x *= x) if (n & 1) ret *= x; return ret; }
public: constexpr ModInt() : r(0) {} constexpr ModInt(
int x) : r(x % MOD) { if (r < 0) r += MOD; } constexpr
ModInt(i64 x) : r(x % MOD) { if (r < 0) r += MOD; }
ModInt Inv() const { return Pow(*this, MOD - 2); }
ModInt operator- () const { return r ? MOD - r : 0; }
ModInt operator+ (const ModInt& x) const { return ModInt
(*this) += x; } ModInt operator- (const ModInt& x) const
 { return ModInt(*this) -= x; } ModInt operator* (const
ModInt& x) const { return ModInt(*this) *= x; } ModInt
operator/ (const ModInt& x) const { return ModInt(*this)
 /= x; } ModInt operator+= (const ModInt& x) { r += x.r;
 if (r >= MOD) r -= MOD; return *this; } ModInt operator
-= (const ModInt& x) { r -= x.r; if (r < 0) r += MOD;
return *this; } ModInt operator*= (const ModInt& x) { r
= (i64)r * x.r % MOD; return *this; } ModInt operator/=
(const ModInt& x) { return *this *= x.Inv(); } bool
operator== (const ModInt& x) const { return r == x.r; }
bool operator!= (const ModInt& x) const { return r != x.
r; } operator int() const { return r; } operator i64()
const { return r; } friend istream& operator>> (istream&
 in, ModInt& x) { i64 t; cin >> t; x = ModInt(t); return
 in; } friend ostream& operator<< (ostream& out, const
ModInt& x) { return out << x.r; } }; using mint = ModInt
<998'244'353>; /* Linear Sieve, O(n) */ vector<int>
PrimeEnumerate(int n) { vector<int> P; vector<bool> B(n
+ 1, 1); for (int i = 2; i <= n; i++) { if (B[i]) P.
push_back(i); for (int j : P) { if (i * j > n) break; B[
i * j] = 0; if (i % j == 0) break; } } return P; }
template<typename T> void MultipleZetaTransform(vector<T
>& v) { const int n = (int)v.size() - 1; for (int p :
PrimeEnumerate(n)) { for (int i = n / p; i; i--) v[i] +=
 v[i * p]; } } template<typename T> void
MultipleMobiusTransform(vector<T>& v) { const int n = (
int)v.size() - 1; for (int p : PrimeEnumerate(n)) { for
(int i = 1; i * p <= n; i++) v[i] -= v[i * p]; } }
template<typename T> vector<T> GCDConvolution(vector<T>
A, vector<T> B) { MultipleZetaTransform(A);
MultipleZetaTransform(B); for (int i = 0; i < A.size();
i++) A[i] *= B[i]; MultipleMobiusTransform(A); return A;
 }
```

3.9 LCM cnvolution

```
/* See GCD for MODint */ using mint = ModInt<998'244'
353>; /* Linear Sieve, O(n) */ vector<int>
PrimeEnumerate(int n) { vector<int> P; vector<bool> B(n
+ 1, 1); for (int i = 2; i <= n; i++) { if (B[i]) P.
push_back(i); for (int j : P) { if (i * j > n) break; B[
i * j] = 0; if (i % j == 0) break; } } return P; }
template<typename T> void DivisorZetaTransform(vector<T
>& v) { const int n = (int)v.size() - 1; for (int p :
PrimeEnumerate(n)) { for (int i = 1; i * p <= n; i++) v[
i * p] += v[i]; } } template<typename T> void
DivisorMobiusTransform(vector<T>& v) { const int n = (
int)v.size() - 1; for (int p : PrimeEnumerate(n)) { for
(int i = n / p; i; i--) v[i * p] -= v[i]; } } template<
typename T> vector<T> LCMConvolution(vector<T> A, vector
<T> B) { DivisorZetaTransform(A); DivisorZetaTransform(B
); for (int i = 0; i < A.size(); i++) A[i] *= B[i];
DivisorMobiusTransform(A); return A; } int main() {
fastio; int n; cin >> n; vector A(n + 1, mint(0)), B(n +
 1, mint(0)); for (int i = 1; i <= n; i++) cin >> A[i];
for (int i = 1; i <= n; i++) cin >> B[i]; auto C =
LCMConvolution(A, B); for (int i = 1; i <= n; i++) cout
<< C[i] << ' '; }
```

3.10 LargePrime

```
vector <int> sieve(const int N, const int Q = 17, const
int L = 1 << 15) { static const int rs[] = {1, 7, 11,
13, 17, 19, 23, 29}; struct P { P(int p) : p(p) {} int p
; int pos[8]; }; auto approx_prime_count = [] (const int
 N) -> int { return N > 60184 ? N / (log(N) - 1.1) : max
(1., N / (log(N) - 1.11)) + 1; }; const int v = sqrt(N),
 vv = sqrt(v); vector<bool> isp(v + 1, true); for (int i
```

```
 = 2; i <= vv; ++i) if (isp[i]) { for (int j = i * i; j
<= v; j += i) isp[j] = false; } const int rsize =
approx_prime_count(N + 30); vector<int> primes = {2, 3,
5}; int psize = 3; primes.resize(rsize); vector<P>
sprimes; size_t pbeg = 0; int prod = 1; for (int p = 7;
p <= v; ++p) { if (!isp[p]) continue; if (p <= Q) prod
*= p, ++pbeg, primes[psize++] = p; auto pp = P(p); for (
int t = 0; t < 8; ++t) { int j = (p <= Q) ? p : p * p;
while (j % 30 != rs[t]) j += p << 1; pp.pos[t] = j / 30;
 } sprimes.push_back(pp); } vector<unsigned char> pre(
prod, 0xFF); for (size_t pi = 0; pi < pbeg; ++pi) { auto
 pp = sprimes[pi]; const int p = pp.p; for (int t = 0; t
 < 8; ++t) { const unsigned char m = ~(1 << t); for (int
 i = pp.pos[t]; i < prod; i += p) pre[i] &= m; } } const
 int block_size = (L + prod - 1) / prod * prod; vector<
unsigned char> block(block_size); unsigned char* pblock
= block.data(); const int M = (N + 29) / 30; for (int
beg = 0; beg < M; beg += block_size, pblock -=
block_size) { int end = min(M, beg + block_size); for (
int i = beg; i < end; i += prod) { copy(pre.begin(), pre
.end(), pblock + i); } if (beg == 0) pblock[0] &= 0xFE;
for (size_t pi = pbeg; pi < sprimes.size(); ++pi) { auto
& pp = sprimes[pi]; const int p = pp.p; for (int t = 0;
t < 8; ++t) { int i = pp.pos[t]; const unsigned char m =
 ~(1 << t); for (; i < end; i += p) pblock[i] &= m; pp.
pos[t] = i; } } for (int i = beg; i < end; ++i) { for (
int m = pblock[i]; m > 0; m &= m - 1) { primes[psize++]
= i * 30 + rs[__builtin_ctz(m)]; } } } assert(psize <=
rsize); while (psize > 0 && primes[psize - 1] > N) --
psize; primes.resize(psize); return primes; }
```

3.11 Linear Recurance

```
const int N = 3e5 + 9, mod = 1e9 + 7; template <int32_t
MOD> struct modint { int32_t value; modint() = default;
modint(int32_t value_) : value(value_) {} inline modint<
MOD> operator + (modint<MOD> other) const { int32_t c =
this->value + other.value; return modint<MOD>(c >= MOD ?
 c - MOD : c); } inline modint<MOD> operator - (modint<
MOD> other) const { int32_t c = this->value - other.
value; return modint<MOD>(c < 0 ? c + MOD : c); } inline
 modint<MOD> operator * (modint<MOD> other) const {
int32_t c = (int64_t)this->value * other.value % MOD;
return modint<MOD>(c < 0 ? c + MOD : c); } inline modint
<MOD> & operator += (modint<MOD> other) { this->value +=
 other.value; if (this->value >= MOD) this->value -= MOD
; return *this; } inline modint<MOD> & operator -= (
modint<MOD> other) { this->value -= other.value; if (
this->value < 0) this->value += MOD; return *this; }
inline modint<MOD> & operator *= (modint<MOD> other) {
this->value = (int64_t)this->value * other.value % MOD;
if (this->value < 0) this->value += MOD; return *this; }
 inline modint<MOD> operator - () const { return modint<
MOD>(this->value ? MOD - this->value : 0); } modint<MOD>
 pow(uint64_t k) const { modint<MOD> x = *this, y = 1;
for (; k; k >>= 1) { if (k & 1) y *= x; x *= x; } return
 y; } modint<MOD> inv() const { return pow(MOD - 2); }
/* MOD must be a prime */ inline modint<MOD> operator /
(modint<MOD> other) const { return *this * other.inv();
} inline modint<MOD> operator /= (modint<MOD> other) {
return *this *= other.inv(); } inline bool operator == (
modint<MOD> other) const { return value == other.value;
} inline bool operator != (modint<MOD> other) const {
return value != other.value; } inline bool operator < (
modint<MOD> other) const { return value < other.value; }
 inline bool operator > (modint<MOD> other) const {
return value > other.value; } }; template <int32_t MOD>
modint<MOD> operator * (int64_t value, modint<MOD> n) {
return modint<MOD>(value) * n; } template <int32_t MOD>
modint<MOD> operator * (int32_t value, modint<MOD> n) {
return modint<MOD>(value % MOD) * n; } template <int32_t
 MOD> istream & operator >> (istream & in, modint<MOD> &
n) { return in >> n.value;} template <int32_t MOD>
ostream & operator << (ostream & out, modint<MOD> n) {
return out << n.value; } using mint = modint<mod>;
vector<mint> combine (int n, vector<mint> &a, vector<
mint> &b, vector<mint> &tr) { vector<mint> res(n * 2 +
1, 0); for (int i = 0; i < n + 1; i++) { for (int j = 0;
 j < n + 1; j++) res[i + j] += a[i] * b[j]; } for (int i
 = 2 * n; i > n; --i) { for (int j = 0; j < n; j++) res[
i - 1 - j] += res[i] * tr[j]; } res.resize(n + 1);
return res; }; // transition -> for(i = 0; i < x; i++) f
[n] += tr[i] * f[n-i-1] // S contains initial values, k
is 0 indexed mint LinearRecurrence(vector<mint> &S,
vector<mint> &tr, long long k) { int n = S.size();
assert(n == (int)tr.size()); if (n == 0) return 0; if (k
 < n) return S[k]; vector<mint> pol(n + 1), e(pol); pol
[0] = e[1] = 1; for (++k; k; k /= 2) { if (k % 2) pol =
combine(n, pol, e, tr); e = combine(n, e, e, tr); } mint
 res = 0; for (int i = 0; i < n; i++) res += pol[i + 1]
* S[i]; return res; }
```

3.12 Lucas

```
const int N = 1e6 + 3, mod = 1e6 + 3; using ll = long
long; template <const int32_t MOD> struct modint {
int32_t value; modint() = default; modint(int32_t value_
) : value(value_) {} inline modint<MOD> operator + (
modint<MOD> other) const { int32_t c = this->value +
other.value; return modint<MOD>(c >= MOD ? c - MOD : c);
 } inline modint<MOD> operator - (modint<MOD> other)
const { int32_t c = this->value - other.value; return
modint<MOD>(c < 0 ? c + MOD : c); } inline modint<MOD>
operator * (modint<MOD> other) const { int32_t c = (
int64_t)this->value * other.value % MOD; return modint<
MOD>(c < 0 ? c + MOD : c); } inline modint<MOD> &
operator += (modint<MOD> other) { this->value += other.
value; if (this->value >= MOD) this->value -= MOD;
return *this; } inline modint<MOD> & operator -= (modint
<MOD> other) { this->value -= other.value; if (this->
value < 0) this->value += MOD; return *this; } inline
modint<MOD> & operator *= (modint<MOD> other) { this->
value = (int64_t)this->value * other.value % MOD; if (
this->value < 0) this->value += MOD; return *this; }
inline modint<MOD> operator - () const { return modint<
MOD>(this->value ? MOD - this->value : 0); } modint<MOD>
 pow(uint64_t k) const { modint<MOD> x = *this, y = 1;
for (; k; k >>= 1) { if (k & 1) y *= x; x *= x; } return
 y; } modint<MOD> inv() const { return pow(MOD - 2); }
/* // MOD must be a prime */ inline modint<MOD> operator
 / (modint<MOD> other) const { return *this * other.inv
(); } inline modint<MOD> operator /= (modint<MOD> other)
 { return *this *= other.inv(); } inline bool operator
== (modint<MOD> other) const { return value == other.
value; } inline bool operator != (modint<MOD> other)
const { return value != other.value; } inline bool
operator < (modint<MOD> other) const { return value <
other.value; } inline bool operator > (modint<MOD> other
) const { return value > other.value; } }; template <
int32_t MOD> modint<MOD> operator * (int32_t value,
modint<MOD> n) { return modint<MOD>(value) * n; }
template <int32_t MOD> modint<MOD> operator * (int64_t
value, modint<MOD> n) { return modint<MOD>(value % MOD)
* n; } template <int32_t MOD> istream & operator >> (
istream & in, modint<MOD> &n) { return in >> n.value; }
template <int32_t MOD> ostream & operator << (ostream &
out, modint<MOD> n) { return out << n.value; } using
mint = modint<mod>; struct combi{ int n; vector<mint>
facts, finvs, invs; combi(int _n): n(_n), facts(_n),
finvs(_n), invs(_n){ facts[0] = finvs[0] = 1; invs[1] =
```

```
1; for (int i = 2; i < n; i++) invs[i] = invs[mod % i] *
 (-mod / i); for(int i = 1; i < n; i++){ facts[i] =
facts[i - 1] * i; finvs[i] = finvs[i - 1] * invs[i]; } }
 inline mint fact(int n) { return facts[n]; } inline
mint finv(int n) { return finvs[n]; } inline mint inv(
int n) { return invs[n]; } inline mint ncr(int n, int k)
 { return n < k or k < 0 ? 0 : facts[n] * finvs[k] *
finvs[n-k]; } }; combi C(N); /* // returns nCr modulo
mod where mod is a prime // Complexity: log(n) */ mint
lucas(ll n, ll r) { if (r > n) return 0; if (n < mod)
return C.ncr(n, r); return lucas(n / mod, r / mod) *
lucas(n % mod, r % mod); }
```

3.13 Matrix

```
template<typename DT> struct Matrix{ vector<vector<DT>>
mat; Matrix(vector<vector<DT>> &mat) : mat(mat) {}
Matrix(int n) : mat(n, vector<DT> (n)){ for(int i = 0; i
 < n; i++){ mat[i][i] = 1; } } Matrix(int n, int m) :
mat(n, vector<DT> (m)) {} Matrix operator*(Matrix &other
){ auto &a = mat, &b = other.mat; assert(a[0].size() ==
b.size()); int n = a.size(), m = b[0].size(), s = a[0].
size(); Matrix<DT> ret(n, m); for(int i = 0; i < n; i++)
{ for(int j = 0; j < m; j++){ DT temp = 0; for(int k =
0; k < s; k++){ temp = (temp + 1LL * a[i][k] * b[k][j] %
 MOD) % MOD; } ret.mat[i][j] = temp; } } return ret; }
}; Matrix<int> pow(Matrix<int> a, LL p){ int n = a.mat.
size(); Matrix<int> ret(n); while(p){ if(p & 1) ret =
ret * a; a = a * a; p >>= 1; } return ret; }
```

3.14 NOD and SOD

```
/* // NUMBER = p_1^a_1 * p_2^a_2 .... p_n^a_n */ LL NOD
= 1, SOD = 1, POD = 1, POWER = 1; for(int i = 0; i < n;
i++) { LL p, a; cin >> p >> a; NOD = (NOD * (a + 1)) %
MOD; SOD = ((SOD * (bigmod(p, a + 1, MOD) + MOD - 1)) %
MOD * inv[p - 1]) % MOD; POD = bigmod(POD, a + 1, MOD) *
 bigmod(bigmod(x, a * (a + 1) / 2, MOD), POWER, MOD) %
MOD; POWER = (POWER * (a + 1)) % (MOD - 1); } cout <<
NOD << ' ' << SOD << ' ' << POD << '\n'; /* // CSOD */
LL csod(LL n) { LL ans = 0; for(LL i = 2; i * i <= n; ++
i) { LL j = n / i; ans += (i + j) * (j - i + 1) / 2; ans
 += i * (j - i); } return ans; } summation of NOD(d)[d|n
] = product of g(e_k + 1)[n=p_k^a_k] g(x) = x * (x + 1)
/ 2
```

3.15 Pollard rho

```
namespace rho{ inline LL mul(LL a, LL b, LL mod) { LL
result = 0; while (b) { if (b & 1) result = (result + a)
 % mod; a = (a + a) % mod; b >>= 1; } return result; }
inline LL bigmod(LL num,LL pow,LL mod){ LL ans = 1; for(
 ; pow > 0; pow >>= 1, num = mul(num, num, mod)) if(pow
&1) ans = mul(ans,num,mod); return ans; } inline bool
is_prime(LL n){ if(n < 2 or n % 6 % 4 != 1) return (n|1)
 == 3; LL a[] = {2, 325, 9375, 28178, 450775, 9780504,
1795265022}; LL s = __builtin_ctzll(n-1), d = n >> s;
for(LL x: a){ LL p = bigmod(x % n, d, n), i = s; for( ;
p != 1 and p != n-1 and x % n and i--; p = mul(p, p, n))
; if(p != n-1 and i != s) return false; } return true; }
 LL f(LL x, LL n) { return mul(x, x, n) + 1; } LL
get_factor(LL n) { LL x = 0, y = 0, t = 0, prod = 2, i =
 2, q; for( ; t++ %40 or __gcd(prod, n) == 1; x = f(x, n
), y = f(f(y, n), n) ){ (x == y) ? x = i++, y = f(x, n)
: 0; prod = (q = mul(prod, max(x,y) - min(x,y), n)) ? q
: prod; } return __gcd(prod, n); } void _factor(LL n,
map <LL, int> &res) { if(n == 1) return; if(is_prime(n))
 res[n]++; else { LL x = get_factor(n); _factor(x, res);
 _factor(n / x, res); } } map <LL, int> factorize(LL n){
 map <LL, int> res; if(n < 2) return res; LL
small_primes[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89,
97 }; for (LL p: small_primes) for( ; n % p == 0; n /= p
, res[p]++); _factor(n, res); return res; } }
```

3.16 Sieve

```
const int N = 10000000; vector <int> lp(N), pr; for (int
 i = 2; i < N; i++) { if (lp[i] == 0) { lp[i] = i; pr.
push_back (i); } for (int j = 0; i * pr[j] < N; j++) {
lp[i * pr[j]] = pr[j]; if (pr[j] == lp[i]) break; } }
```

3.17 XOR Basis

```
template<typename T = int, int B = 31> struct Basis { T
a[B]; Basis() { memset(a, 0, sizeof a); } void insert(T
x){ for (int i = B - 1; i >= 0; i--) { if (x >> i & 1) {
 if (a[i]) x ^= a[i]; else { a[i] = x; break; } } } }
bool can(T x) { for(int i = B - 1; i >= 0; i--) { x =
min(x, x ^ a[i]); } return x == 0; } T max_xor(T ans =
0) { for(int i = B - 1; i >= 0; i--) { ans = max(ans,
ans ^ a[i]); } return ans; } };
```

3.18 mobius function

```
const int N = 1e6 + 5; int mob[N]; void mobius() {
memset(mob, -1, sizeof mob); mob[1] = 1; for (int i = 2;
 i < N; i++) if (mob[i]) { for (int j = i + i; j < N; j
+= i) mob[j] -= mob[i]; } }
```

3.19 nCr

```
namespace com { LL fact[N], inv[N], inv_fact[N]; void
init() { fact[0] = inv_fact[0] = 1; for (LL i = 1; i < N
; i++) { inv[i] = i == 1 ? 1 : (LL)inv[i - mod % i] * (
mod / i + 1) % mod; fact[i] = (LL)fact[i - 1] * i % mod;
 inv_fact[i] = (LL)inv_fact[i - 1] * inv[i] % mod; } }
LL C(int n, int r) { return (r < 0 or r > n) ? 0 : fact[
n] * inv_fact[r] % mod * inv_fact[n - r] % mod; } }
```

3.20 ntt

```
const LL N = 1 << 18; const LL MOD = 786433; vector<LL>
P[N]; LL rev[N], w[N | 1], a[N], b[N], inv_n, g; LL Pow(
LL b, LL p) { LL ret = 1; while (p) { if (p & 1) ret = (
ret * b) % MOD; b = (b * b) % MOD; p >>= 1; } return ret
; } LL primitive_root(LL p) { vector<LL> factor; LL phi
= p - 1, n = phi; for (LL i = 2; i * i <= n; i++) { if (
n % i) continue; factor.emplace_back(i); while (n % i ==
 0) n /= i; } if (n > 1) factor.emplace_back(n); for (LL
 res = 2; res <= p; res++) { bool ok = true; for (LL i =
 0; i < factor.size() && ok; i++) ok &= Pow(res, phi /
factor[i]) != 1; if (ok) return res; } return -1; } void
 prepare(LL n) { LL sz = abs(31 - __builtin_clz(n)); LL
r = Pow(g, (MOD - 1) / n); inv_n = Pow(n, MOD - 2); w[0]
 = w[n] = 1; for (LL i = 1; i < n; i++) w[i] = (w[i - 1]
 * r) % MOD; for (LL i = 1; i < n; i++) rev[i] = (rev[i
>> 1] >> 1) | ((i & 1) << (sz - 1)); } void NTT(LL *a,
LL n, LL dir = 0) { for (LL i = 1; i < n - 1; i++) if (i
 < rev[i]) swap(a[i], a[rev[i]]); for (LL m = 2; m <= n;
 m <<= 1) { for (LL i = 0; i < n; i += m) { for (LL j =
0; j < (m >> 1); j++) { LL &u = a[i + j], &v = a[i + j +
 (m >> 1)]; LL t = v * w[dir ? n - n / m * j : n / m * j
] % MOD; v = u - t < 0 ? u - t + MOD : u - t; u = u + t
>= MOD ? u + t - MOD : u + t; } } } if (dir) for (LL i =
 0; i < n; i++) a[i] = (inv_n * a[i]) % MOD; } vector<LL
> mul(vector<LL> p, vector<LL> q) { LL n = p.size(), m =
 q.size(); LL t = n + m - 1, sz = 1; while (sz < t) sz
<<= 1; prepare(sz); for (LL i = 0; i < n; i++) a[i] = p[
i]; for (LL i = 0; i < m; i++) b[i] = q[i]; for (LL i =
n; i < sz; i++) a[i] = 0; for (LL i = m; i < sz; i++) b[
i] = 0; NTT(a, sz); NTT(b, sz); for (LL i = 0; i < sz; i
++) a[i] = (a[i] * b[i]) % MOD; NTT(a, sz, 1); vector<LL
> c(a, a + sz); while (c.size() && c.back() == 0) c.
pop_back(); return c; }
```

3.21 primality test

```
using ULL = unsigned long long; /* // 5472940991761
worst carmichael number */ inline ULL mul(ULL a,ULL b,
ULL mod){ LL ans = a * b - mod * (ULL) (1.L / mod * a *
b); return ans + mod * (ans < 0) - mod * (ans >= (LL)
mod); } inline ULL bigmod(ULL num,ULL pow,ULL mod){ ULL
ans = 1; for( ; pow > 0; pow >>= 1, num = mul(num, num,
mod)) if(pow&1) ans = mul(ans,num,mod); return ans; }
inline bool is_prime(ULL n){ if(n < 2 or n % 6 % 4 != 1)
 return (n|1) == 3; ULL a[] = {2, 325, 9375, 28178,
450775, 9780504, 1795265022}; ULL s = __builtin_ctzll(n
-1), d = n >> s; for(ULL x: a){ ULL p = bigmod(x % n, d,
 n), i = s; for( ; p != 1 and p != n-1 and x % n and i
--; p = mul(p, p, n)); if(p != n-1 and i != s) return
false; } return true; } /* // handle bigmod overflow */
vector <int> primes; vector <bool> prime(200,true); bool
 check_primality(LL p){ if(p<2) return false; if(primes.
empty()){ for(int i=2;i<100;i++){ for(int j=i+i;j<200;j
+=i) prime[j] = false; } for(int i=2;i<200;i++){ if(
prime[i]) primes.push_back(i); } } for(auto a: primes){
if(p!=a and bigmod(a,p-1,p) != 1) return false; } return
 true; }
```

3.22 prime counting function

```
namespace PCF {
  const LL MAX = 1E13;
  const int N = 7E6; /// around MAX^(2/3)/15
  const int M = 7, PM = 2 * 3 * 5 * 7 * 11 * 13 * 17;

  bool isp[N];
  int prime[N], pi[N];
  int phi[M + 1][PM + 1], sz[M + 1];

  auto div = [](LL a, LL b) -> LL { return double(a) / b
  ; };
  auto rt2 = [](LL x) -> int { return sqrtl(x); };
  auto rt3 = [](LL x) -> int { return cbrtl(x); };

  void init() {
    int cnt = 0;
    for (int i = 2; i < N; i++) isp[i] = true;
    pi[0] = pi[1] = 0;
    for (int i = 2; i < N; i++) {
      if (isp[i]) prime[++cnt] = i;
      pi[i] = cnt;
      for (int j = 1; j <= cnt && i * prime[j] < N; j++)
      {
        isp[i * prime[j]] = false;
        if (i % prime[j] == 0) break;
      }
    }
    sz[0] = 1;
    for (int i = 0; i <= PM; ++i) phi[0][i] = i;
    for (int i = 1; i <= M; ++i) {
      sz[i] = prime[i] * sz[i - 1];
      for (int j = 1; j <= PM; ++j) phi[i][j] = phi[i -
      1][j] - phi[i - 1][div(j, prime[i])];
    }
  }
  LL getphi(LL x, int s) {
    if (s == 0) return x;
    if (s <= M) return phi[s][x % sz[s]] + (x / sz[s]) *
     phi[s][sz[s]];
    if (x <= 1LL * prime[s] * prime[s]) return pi[x] - s
     + 1;
    if (x <= 1LL * prime[s] * prime[s] * prime[s] && x <
     N) {
      int s2x = pi[rt2(x)];
      LL ans = pi[x] - (s2x + s - 2) * (s2x - s + 1) / 2;
      for (int i = s + 1; i <= s2x; ++i) ans += pi[div(x,
       prime[i])];
      return ans;
    }
    return getphi(x, s - 1) - getphi(div(x, prime[s]), s
    - 1);
  }
  LL getpi(LL x) {
    if (x < N) return pi[x];
    LL ans = getphi(x, pi[rt3(x)]) + pi[rt3(x)] - 1;
    for (int i = pi[rt3(x)] + 1, ed = pi[rt2(x)]; i <=
    ed; ++i) ans -= getpi(div(x, prime[i])) - i + 1;
    return ans;
  }
}  // namespace PCF
```

4 Graph

4.1 Bellman Ford

```
void bellmanford(int n, int m, vector<int> edge[], int
dist[], int src){ fill(dist, dist + n, INT_MAX); dist[
src] = 0; int i, j, k; vector<int> v; for (i = 0; i < n;
 i++){ for (j = 0; j < m; j++) { v = edge[j]; if (dist[v
[1]] > dist[v[0]] + v[2]) dist[v[1]] = dist[v[0]] + v
[2]; } } for (j = 0; j < m; j++){ /* // For checking
negative loop */ v = edge[j]; if (dist[v[1]] > dist[v
[0]] + v[2]){ fill(dist, dist + n, INT_MIN); /* //
Negative loop detected */ return; } } }
```

4.2 BridgeTree

```
vector<PLL> g[N]; vector<int> ng[N]; int disc[N], low[N
], mark[N], vis[N], timer = 1; void find_bridge(int u,
int p) { disc[u] = low[u] = timer++; bool fl = 1; for (
auto [v, id] : g[u]) { if (v == p && fl) { fl = 0;
continue; } if (disc[v]) { low[u] = min(low[u], disc[v])
; } else { find_bridge(v, u); low[u] = min(low[u], low[v
]); if (disc[u] < low[v]) { mark[id] = 1; } } } } void
colorComponents(int u, int color) { if (vis[u]) return;
vis[u] = color; for (auto [v, id] : g[u]) { if (mark[id
]) continue; colorComponents(v, color); } } void solve()
 { int n, m; cin >> n >> m; vector<PLL> edges; for (int
i = 0; i < m; i++) { int u, v; cin >> u >> v; edges.
push_back({u, v}); g[u].push_back({v, i}); g[v].
push_back({u, i}); } find_bridge(1, 0); int color = 1;
for (int i = 1; i <= n; i++) { if (!vis[i])
colorComponents(i, color++); } for (int i = 0; i < m; i
++) { if (mark[i]) { ng[vis[edges[i].first]].push_back(
vis[edges[i].second]); ng[vis[edges[i].second]].
push_back(vis[edges[i].first]); } } }
```

4.3 DSU, MST

```
class DSU { vector<int> parent, size; public: DSU(int n)
 : parent(n + 1), size(n + 1, 1) { iota(parent.begin(),
parent.end(), 0); } int root(int u) { if (parent[u] == u
) return u; return parent[u] = root(parent[u]); } bool
same(int u, int v) { return root(u) == root(v); } void
merge(int u, int v) { u = root(u), v = root(v); if (u ==
 v) return; if (size[u] < size[v]) swap(u, v); parent[v]
 = u, size[u] += size[v]; } }; int kruskal(vector<tuple<
int, int, int>> edges, int n) { sort(edges.begin(),
edges.end()); DSU mst(n); int cost = 0; for (auto &[w, u
, v] : edges) { if (mst.same(u, v)) continue; mst.merge(
u, v); cost += w; } return cost; } /* // PRIM'S SPANNING
 TREE (MST) */ DIJKSTRA code... start from a node, and
push nodes which are not marked popped edges weight are
taken
```

4.4 ETT, VT

```
struct euler_tour { int time = 0; tree &T; int n; vector
<int> start, finish, level, par; euler_tour(tree &T, int
 root = 0) : T(T), n(T.n), start(n), finish(n), level(n)
, par(n) { time = 0; call(root); } void call(int node,
int p = -1) { if (p != -1) level[node] = level[p] + 1;
start[node] = time++; for (int e : T[node]) if (e != p)
call(e, node); par[node] = p; finish[node] = time++; }
bool isAncestor(int node, int par) { return start[par]
<= start[node] and finish[par] >= finish[node]; } int
subtreeSize(int node) { return finish[node] - start[node
] + 1 >> 1; } }; tree virtual_tree(vector<int> &nodes,
lca_table &table, euler_tour &tour) { sort(nodes.begin()
, nodes.end(), [&](int x, int y) { return tour.start[x]
< tour.start[y]; }); int n = nodes.size(); for (int i =
0; i + 1 < n; i++) nodes.push_back(table.lca(nodes[i],
nodes[i + 1])); sort(nodes.begin(), nodes.end()); nodes.
erase(unique(nodes.begin(), nodes.end()), nodes.end());
sort(nodes.begin(), nodes.end(), [&](int x, int y) {
return tour.start[x] < tour.start[y]; }); n = nodes.size
(); stack<int> st; st.push(0); tree ans(n); for (int i =
 1; i < n; i++) { while (!tour.isAncestor(nodes[i],
nodes[st.top()])) st.pop(); ans.addEdge(st.top(), i); st
.push(i); } return ans; } set<int> getCenters(tree &T) {
 int n = T.n; vector<int> deg(n), q; set<int> s; for (
int i = 0; i < n; i++) { deg[i] = T[i].size(); if (deg[i
] == 1) q.push_back(i); s.insert(i); } for (vector<int>
t; s.size() > 2; q = t) { for (auto x : q) { for (auto e
 : T[x]) if (--deg[e] == 1) t.push_back(e); s.erase(x);
} } return s; }
```

4.5 HLD

```
class HLD { vector<int> parent, depth, heavy, head, pos,
 euler, start, end; int n, cur_pos; LazySegmentTree
segTree; int dfs(int v, const vector<vector<int>> &adj)
{ int size = 1, max_c_size = 0; for (int c : adj[v]) {
if (c != parent[v]) { parent[c] = v; depth[c] = depth[v]
 + 1; int c_size = dfs(c, adj); size += c_size; if (
c_size > max_c_size) { max_c_size = c_size; heavy[v] = c
; } } } return size; } void decompose(int v, int h,
const vector<vector<int>> &adj) { head[v] = h; pos[v] =
cur_pos++; euler.push_back(v); start[v] = euler.size() -
 1; if (heavy[v] != -1) decompose(heavy[v], h, adj); for
 (int c : adj[v]) { if (c != parent[v] && c != heavy[v])
 decompose(c, c, adj); } end[v] = euler.size() - 1; }
public: HLD(int n, const vector<vector<int>> &adj,
vector<LL> &v) : n(n), parent(n), depth(n), heavy(n, -1)
, head(n), pos(n), start(n), end(n), cur_pos(0), segTree
(v, 0) { parent[0] = -1; depth[0] = 0; dfs(0, adj);
decompose(0, 0, adj); for (int i = 0; i < n; i++)
segTree.update(pos[i], pos[i], v[i]); } void update_path
 (int a, int b, LL val) { while (head[a] != head[b]) {
if (depth[head[a]] < depth[head[b]]) swap(a, b); segTree
.update(pos[head[a]], pos[a], val); a = parent[head[a]];
 } if (depth[a] > depth[b]) swap(a, b); segTree.update(
pos[a], pos[b], val); } LL query_path (int a, int b) {
LL res = 0; while (head[a] != head[b]) { if (depth[head[
a]] < depth[head[b]]) swap(a, b); res = max (res,
segTree.query(pos[head[a]], pos[a])); a = parent[head[a
]]; } if (depth[a] > depth[b]) swap(a, b); res = max (
res, segTree.query(pos[a], pos[b])); return res; } void
update_subtree (int v, LL val) { segTree.update(start[v
], end[v], val); } LL query_subtree (int v) { return
segTree.query(start[v], end[v]); } };
```

4.6 Hungarian

```
/* Complexity: O(n^3) but optimized It finds minimum
cost maximum matching. For finding maximum cost maximum
matching add -cost and return -matching() 1-indexed */
struct Hungarian { long long c[N][N], fx[N], fy[N], d[N
]; int l[N], r[N], arg[N], trace[N]; queue<int> q; int
start, finish, n; const long long inf = 1e18; Hungarian
() {} Hungarian(int n1, int n2): n(max(n1, n2)) { for (
int i = 1; i <= n; ++i) { fy[i] = l[i] = r[i] = 0; for (
int j = 1; j <= n; ++j) c[i][j] = inf; /* // make it 0
for maximum cost matching (not necessarily with max
count of matching) */ } } void add_edge(int u, int v,
long long cost) { c[u][v] = min(c[u][v], cost); } inline
 long long getC(int u, int v) { return c[u][v] - fx[u] -
 fy[v]; } void initBFS() { while (!q.empty()) q.pop(); q
.push(start); for (int i = 0; i <= n; ++i) trace[i] = 0;
 for (int v = 1; v <= n; ++v) { d[v] = getC(start, v);
arg[v] = start; } finish = 0; } void findAugPath() {
while (!q.empty()) { int u = q.front(); q.pop(); for (
int v = 1; v <= n; ++v) if (!trace[v]) { long long w =
getC(u, v); if (!w) { trace[v] = u; if (!r[v]) { finish
= v; return; } q.push(r[v]); } if (d[v] > w) { d[v] = w;
 arg[v] = u; } } } } void subX_addY() { long long delta
= inf; for (int v = 1; v <= n; ++v) if (trace[v] == 0 &&
 d[v] < delta) { delta = d[v]; } /* Rotate */ fx[start]
+= delta; for (int v = 1; v <= n; ++v) if(trace[v]) {
int u = r[v]; fy[v] -= delta; fx[u] += delta; } else d[v
] -= delta; for (int v = 1; v <= n; ++v) if (!trace[v]
&& !d[v]) { trace[v] = arg[v]; if (!r[v]) { finish = v;
return; } q.push(r[v]); } } void Enlarge() { do { int u
= trace[finish]; int nxt = l[u]; l[u] = finish; r[finish
] = u; finish = nxt; } while (finish); } long long
maximum_matching() { for (int u = 1; u <= n; ++u) { fx[u
] = c[u][1]; for (int v = 1; v <= n; ++v) { fx[u] = min(
fx[u], c[u][v]); } } for (int v = 1; v <= n; ++v) { fy[v
] = c[1][v] - fx[1]; for (int u = 1; u <= n; ++u) { fy[v
] = min(fy[v], c[u][v] - fx[u]); } } for (int u = 1; u
<= n; ++u) { start = u; initBFS(); while (!finish) {
findAugPath(); if (!finish) subX_addY(); } Enlarge(); }
long long ans = 0; for (int i = 1; i <= n; ++i) { if (c[
i][l[i]] != inf) ans += c[i][l[i]]; else l[i] = 0; }
return ans; } }; int32_t main() { ios_base::
sync_with_stdio(0); cin.tie(0); int n1, n2, m; cin >> n1
 >> n2 >> m; Hungarian M(n1, n2); for (int i = 1; i <= m
; i++) { int u, v, w; cin >> u >> v >> w; M.add_edge(u,
v, -w); } cout << -M.maximum_matching() << '\n'; for (
int i = 1; i <= n1; i++) cout << M.l[i] << ' '; return
0; }
```

4.7 K th shortest path

```
void K_shortest(int n, int m) { int st, des, k, u, v; LL
 w; scanf("%d%d%d", &st, &des, &k); st--, des--; vector<
vector<pii> > edges(n); for (int i = 0; i < m; i++) {
scanf("%d%d%lld", &u, &v, &w); u--, v--; edges[u].
push_back({w, v}); } vector<vector<LL> > dis(n, vector<
LL>(k + 1, 1e8)); vector<int> vis(n); priority_queue<pii
, vector<pii>, greater<pii> > q; q.emplace(0LL, st);
while (!q.empty()) { v = q.top().second, w = q.top().
first; q.pop(); if (vis[v] >= k) continue; /* // for the
 varient, check if this path is greater than previous,
if not, continue // if(vis[v]>0 && w == dis[v][vis[v
]-1]) continue; */ dis[v][vis[v]] = w; vis[v]++; for (
auto nd : edges[v]) { q.emplace(w + nd.first, nd.second)
; } } LL ans = dis[des][k - 1]; if (ans == 1e8) ans =
-1; printf("%lld\n", ans); }
```

4.8 LCA, CD

```
struct Tree { vector<vector<int>> adj; Tree(int N) : adj
(N + 1) {} void addEdges(int u, int v) { adj[u].
push_back(v); adj[v].push_back(u); } }; class LCA { int
N, K; vector<vector<int>> &adj, anc; vector<int> level;
public: LCA(Tree &tree) : adj(tree.adj) { N = tree.adj.
size() - 1; K = 33 - __builtin_clz(N); anc.assign(N + 1,
 vector<int>(K)); level.assign(N + 1, 0); initLCA(1); }
```

```
void initLCA(int u, int p = 0) { anc[u][0] = p; level[u]
 = level[p] + 1; for (int i = 1; i < K; i++) { anc[u][i]
 = anc[anc[u][i - 1]][i - 1]; } for (auto v : adj[u]) if
 (v != p) { initLCA(v, u); } } int getAnc(int u, int k)
{ for (int i = K - 1; i >= 0; i--) if (k & (1 << i)) u =
 anc[u][i]; return u; } int lca(int u, int v) { if (
level[u] > level[v]) swap(u, v); v = getAnc(v, level[v]
- level[u]); if (u == v) return u; for (int i = K - 1; i
 >= 0; i--) { if (anc[u][i] != anc[v][i]) u = anc[u][i],
 v = anc[v][i]; } return anc[u][0]; } int dis(int u, int
 v) { return level[u] + level[v] - 2 * level[lca(u, v)];
 } }; class CD { vector<vector<int>> adj; vector<int>
sub; vector<bool> blocked; int N; public: CD(Tree &tree)
 : adj(tree.adj) { N = tree.adj.size() - 1; blocked.
assign(N + 1, 0); sub.assign(N + 1, 0); compute(); }
void compute(int u = 1, int p = 0) { sub[u] = 1; for (
auto v : adj[u]) if (v != p) { compute(v, u); sub[u] +=
sub[v]; } } int centroid(int u, int p = 0) { int tot =
sub[u]; for (auto v : adj[u]) { if (v == p || blocked[v
]) continue; if (2 * sub[v] > tot) { sub[u] = tot - sub[
v]; sub[v] = tot; return centroid(v, u); } } return u; }
 int count(int u, int p) { /* // calculate ans */ } void
 update(int u, int p) { /* // update */ } int decompose(
int u = 1) { u = centroid(u); blocked[u] = 1; int ans =
0; /* ///// Do something here //// count() update() */
for (auto v : adj[u]) if (!blocked[v]) { ans += count(v,
 u); update(v, u); } /* /// reset updates here */ for (
auto v : adj[u]) if (!blocked[v]) { decompose(v); }
return ans; } };
```

4.9 block cut tree

```
const int N = 200010; bitset <N> art, good; vector <int>
 g[N], tree[N], st, comp[N]; int n, m, ptr, cur, in[N],
low[N], id[N]; void dfs (int u, int from = -1) { in[u] =
 low[u] = ++ptr; st.emplace_back(u); for (int v : g[u])
if (v ^ from) { if (!in[v]) { dfs(v, u); low[u] = min(
low[u], low[v]); if (low[v] >= in[u]) { art[u] = in[u] >
 1 or in[v] > 2; comp[++cur].emplace_back(u); while (
comp[cur].back() ^ v) { comp[cur].emplace_back(st.back()
); st.pop_back(); } } } else { low[u] = min(low[u], in[v
]); } } } void buildTree() { ptr = 0; for (int i = 1; i
<= n; ++i) { if (art[i]) id[i] = ++ptr; } for (int i =
1; i <= cur; ++i) { int x = ++ptr; for (int u : comp[i])
 { if (art[u]) { tree[x].emplace_back(id[u]); tree[id[u
]].emplace_back(x); } else { id[u] = x; } } } } int main
() { cin >> n >> m; while (m--) { int u, v; scanf("%d %d
", &u, &v); g[u].emplace_back(v); g[v].emplace_back(u);
} for (int i = 1; i <= n; ++i) { if (!in[i]) dfs(i); }
buildTree(); for (int i = 1; i <= ptr; ++i) { cout << i
<< " --> "; for (int j : tree[i]) cout << j << " "; cout
 << '\n'; } return 0; }
```

4.10 maximum bipartite matching hopkroft

```
/* // do everything 1-based */ class BipartiteMatcher {
private: int n, m; /* // Number of vertices in left and
right sets */ vector<vector<int>> adj; /* // Adjacency
list for the bipartite graph */ vector<int> dist; /* //
Distance array for BFS */ vector<int> matchL, matchR; /*
 // matchL[u] is the right vertex matched with u; */ /*
// matchR[v] is the left vertex matched with v */ public
: BipartiteMatcher(int n, int m) : n(n), m(m) { adj.
resize(n + 1); matchL.resize(n + 1, 0); matchR.resize(m
+ 1, 0); dist.resize(n + 1); } void addEdge(int u, int v
) { adj[u].push_back(v); } bool bfs() { queue<int> q;
for (int u = 1; u <= n; u++) { if (matchL[u] == 0) {
dist[u] = 0; q.push(u); } else { dist[u] = INT_MAX; } }
dist[0] = INT_MAX; while (!q.empty()) { int u = q.front
(); q.pop(); if (dist[u] < dist[0]) { for (int v : adj[u
]) { if (dist[matchR[v]] == INT_MAX) { dist[matchR[v]] =
 dist[u] + 1; q.push(matchR[v]); } } } } return dist[0]
!= INT_MAX; } bool dfs(int u) { if (u != 0) { for (int v
 : adj[u]) { if (dist[matchR[v]] == dist[u] + 1) { if (
dfs(matchR[v])) { matchL[u] = v; matchR[v] = u; return
true; } } } dist[u] = INT_MAX; return false; } return
true; } int hopcroftKarp() { int matching = 0; while (
bfs()) { for (int u = 1; u <= n; u++) { if (matchL[u] ==
 0 && dfs(u)) { matching++; } } } return matching; } };
```

4.11 strongly connected component

```
bool vis[N]; vector<int> adj[N], adjr[N]; vector<int>
order, component; /* // tp = 0, finding topo order, //
tp = 1, reverse edge traversal */ void dfs(int u, int tp
 = 0) { vis[u] = true; if (tp) component.push_back(u);
auto& ad = (tp ? adjr : adj); for (int v : ad[u]) if (!
vis[v]) dfs(v, tp); if (!tp) order.push_back(u); } int
main() { for (int i = 1; i <= n; i++) { if (!vis[i]) dfs
(i); } memset(vis, 0, sizeof vis); reverse(order.begin()
, order.end()); for (int i : order) { if (!vis[i]) { /*
// one component is found */ dfs(i, 1), component.clear
(); } } }
```

5 String

5.1 KMP

```
int KMP(vector<int> &a, vector<int> &b) { /* // number
of occurance of a in b */ vector<int> pi(n); for (int i
= 1, j = 0; i < n; i++) { while (j && a[i] != a[j]) j =
pi[j - 1]; if (a[i] == a[j]) j++; pi[i] = j; } int ans =
 0; for (int i = 0, j = 0; i < m; i++) { while (j && b[i
] != a[j]) j = pi[j - 1]; if (a[j] == b[i]) j++; if (j
== n) ans++, j = pi[j - 1]; } return ans; }
```

5.2 Manacher

```
void Manacher() { vector<int> d1(n); /* // d[i] = number
 of palindromes taking s[i] as center */ for (int i = 0,
 l = 0, r = -1; i < n; i++) { int k = (i > r) ? 1 : min(
d1[l + r - i], r - i + 1); while (0 <= i - k && i + k <
n && s[i - k] == s[i + k]) k++; d1[i] = k--; if (i + k >
 r) l = i - k, r = i + k; } vector<int> d2(n); /* // d[i
] = number of palindromes taking s[i-1] and s[i] as
center */ for (int i = 0, l = 0, r = -1; i < n; i++) {
int k = (i > r) ? 0 : min(d2[l + r - i + 1], r - i + 1);
 while (0 <= i - k - 1 && i + k < n && s[i - k - 1] == s
[i + k]) k++; d2[i] = k--; if (i + k > r) l = i - k - 1,
 r = i + k; } }
```

5.3 SuffixArray

```
void inducedSort (const vector <int> &vec, int val_range
, vector <int> &SA, const vector <int> &sl, const vector
 <int> &lms_idx) { vector <int> l(val_range, 0), r(
val_range, 0); for (int c : vec) { ++r[c]; if (c + 1 <
val_range) ++l[c + 1]; } partial_sum(l.begin(), l.end(),
 l.begin()); partial_sum(r.begin(), r.end(), r.begin());
 fill(SA.begin(), SA.end(), -1); for (int i = lms_idx.
size() - 1; i >= 0; --i) SA[--r[vec[lms_idx[i]]]] =
lms_idx[i]; for (int i : SA) if (i > 0 and sl[i - 1]) SA
[l[vec[i - 1]]++] = i - 1; fill(r.begin(), r.end(), 0);
for (int c : vec) ++r[c]; partial_sum(r.begin(), r.end()
, r.begin()); for (int k = SA.size() - 1, i = SA[k]; k;
--k, i = SA[k]) { if (i and !sl[i - 1]) SA[--r[vec[i -
1]]] = i - 1; } } vector <int> suffixArray (const vector
 <int> &vec, int val_range) { const int n = vec.size();
vector <int> sl(n), SA(n), lms_idx; for (int i = n - 2;
i >= 0; --i) { sl[i] = vec[i] > vec[i + 1] or (vec[i] ==
 vec[i + 1] and sl[i + 1]); if (sl[i] and !sl[i + 1])
lms_idx.emplace_back(i + 1); } reverse(lms_idx.begin(),
lms_idx.end()); inducedSort(vec, val_range, SA, sl,
lms_idx); vector <int> new_lms_idx(lms_idx.size()),
```

```
lms_vec(lms_idx.size()); for (int i = 0, k = 0; i < n; ++i) { if (SA[i] > 0 and !sl[SA[i]] and sl[SA[i] - 1]) new_lms_idx[k++] = SA[i]; } int cur = 0; SA[n - 1] = 0; for (int k = 1; k < new_lms_idx.size(); ++k) { int i = new_lms_idx[k - 1], j = new_lms_idx[k]; if (vec[i] ^ vec[j]) { SA[j] = ++cur; continue; } bool flag = 0; for ( int a = i + 1, b = j + 1; ; ++a, ++b) { if (vec[a] ^ vec[b]) { flag = 1; break; } if ((!sl[a] and sl[a - 1]) or (!sl[b] and sl[b - 1])) { flag = !(!sl[a] and sl[a - 1] and !sl[b] and sl[b - 1]); break; } } SA[j] = flag ? ++cur : cur; } for (int i = 0; i < lms_idx.size(); ++i) lms_vec[i] = SA[lms_idx[i]]; if (cur + 1 < lms_idx.size()) { auto lms_SA = suffixArray(lms_vec, cur + 1); for ( int i = 0; i < lms_idx.size(); ++i) new_lms_idx[i] = lms_idx[lms_SA[i]]; } inducedSort(vec, val_range, SA, sl , new_lms_idx); return SA; } vector <int> getSuffixArray (const string &s, const int LIM = 128) { vector <int> vec(s.size() + 1); copy(begin(s), end(s), begin(vec)); vec.back() = '#'; auto ret = suffixArray(vec, LIM); ret.erase(ret.begin()); return ret; } /* // build RMQ on it to get LCP of any two suffix */ vector <int> getLCParray (const string &s, const vector <int> &SA) { int n = s.size(), k = 0; vector <int> lcp(n), rank(n); for (int i = 0; i < n; ++i) rank[SA[i]] = i; for (int i = 0; i < n; ++i, k ? --k : 0) { if (rank[i] == n - 1) { k = 0; continue; } int j = SA[rank[i] + 1]; while (i + k < n and j + k < n and s[i + k] == s[j + k]) ++k; lcp[rank[i]] = k; } lcp[n - 1] = 0; return lcp; }
```

5.4 Z

```
vector<int> z(string const& s) { int n = size(s); vector <int> z(n); int x = 0, y = 0; for (int i = 1; i < n; i ++) { z[i] = max(0, min(z[i - x], y - i + 1)); while (i + z[i] < n && s[z[i]] == s[i + z[i]]) { x = i, y = i + z [i], z[i]++; } } return z; }
```

5.5 double hashing

```
Some Primes: 1000000007, 1000000009, 1000000861, 1000099999 ( < 2^30 ) 1088888881, 1111211111, 1500000001, 1481481481 ( < 2^31 ) namespace Hashing { # define ff first #define ss second const PLL M = {1e9+7, 1e9+9}; const LL base = 1259; const int N = 1e6+7; PLL operator+ (const PLL& a, LL x) {return {a.ff + x, a.ss + x};} PLL operator- (const PLL& a, LL x) {return {a.ff - x, a.ss - x};} PLL operator* (const PLL& a, LL x) { return {a.ff * x, a.ss * x};} PLL operator+ (const PLL& a, PLL x) {return {a.ff + x.ff, a.ss + x.ss};} PLL operator- (const PLL& a, PLL x) {return {a.ff - x.ff, a. ss - x.ss};} PLL operator* (const PLL& a, PLL x) {return {a.ff * x.ff, a.ss * x.ss};} PLL operator% (const PLL& a, PLL m) {return {a.ff % m.ff, a.ss % m.ss};} ostream& operator<<(ostream& os, PLL hash) { return os<<"("<<hash .ff<<", "<<hash.ss<<")"; } PLL pb[N]; /* ///powers of base mod M ///Call pre before everything */ void hashPre () { pb[0] = {1,1}; for (int i=1; i<N; i++) pb[i] = (pb[ i-1] * base)%M; } /* ///Calculates hashes of all prefixes of s including empty prefix */ vector<PLL> hashList(string s) { int n = s.size(); vector<PLL> ans(n +1); ans[0] = {0,0}; for (int i=1; i<=n; i++) ans[i] = ( ans[i-1] * base + s[i-1])%M; return ans; } /* /// Calculates hash of substring s[l..r] (1 indexed) */ PLL substringHash(const vector<PLL> &hashlist, int l, int r) { return (hashlist[r]+(M-hashlist[l-1])*pb[r-l+1])%M; } /* ///Calculates Hash of a string */ PLL Hash (string s ) { PLL ans = {0,0}; for (int i=0; i<s.size(); i++) ans =(ans*base + s[i])%M; return ans; } /* ///appends c to string */ PLL append(PLL cur, char c) { return (cur*base + c)%M; } /* ///prepends c to string with size k */ PLL prepend(PLL cur, int k, char c) { return (pb[k]*c + cur )%M; } /* ///replaces the i-th (0-indexed) character from right from a to b; */ PLL replace(PLL cur, int i, char a, char b) { return (cur + pb[i] * (M+b-a))%M; } /* ///Erases c from front of the string with size len */ PLL pop_front(PLL hash, int len, char c) { return (hash + pb[len-1]*(M-c))%M; } /* ///concatenates two strings where length of the right is k */ PLL concat(PLL left, PLL right, int k) { return (left*pb[k] + right)%M; } PLL power (const PLL& a, LL p) { if (p==0) return {1,1}; PLL ans = power(a, p/2); ans = (ans * ans)%M; if (p%2) ans = (ans*a)%M; return ans; } PLL inverse(PLL a) { if ( M.ss == 1) return power(a, M.ff-2); return power(a, (M. ff-1)*(M.ss-1)-1); } /* ///Erases c from the back of the string */ PLL invb = inverse({base, base}); PLL pop_back(PLL hash, char c) { return ((hash-c+M)*invb)%M; } /* ///Calculates hash of string with size len repeated cnt times ///This is O(log n). For O(1), pre- calculate inverses */ PLL repeat(PLL hash, int len, LL cnt) { PLL mul = ((pb[len*cnt]-1+M) * inverse(pb[len]-1+ M))%M; PLL ans = (hash*mul); if (pb[len].ff == 1) ans.ff = hash.ff*cnt; if (pb[len].ss == 1) ans.ss = hash.ss* cnt; return ans%M; } } using namespace Hashing; vector< PLL> forwardHash, backwardHash; int n; bool check(int l, int r) { return substringHash(forwardHash, l, r) == substringHash(backwardHash, n+1-r, n+1-l); }
```

6 DP

6.1 CHT

```
struct Line { mutable LL m, c, p; bool operator<(const Line& o) const { return m < o.m; } bool operator<(LL x) const { return p < x; } }; /* // this calculates maximum value of m * x + c over all lines // to get minimum value use m = -m , c = -c , query(x) = -query(x) */ struct LineContainer : multiset<Line, less<>> { /* // ( for doubles, use inf = 1/.0, div(a,b) = a/b) */ static const LL inf = LLONG_MAX; LL div(LL a, LL b) { /* // floored division */ return a / b - ((a ^ b) < 0 && a % b ); } bool isect(iterator x, iterator y) { if (y == end() ) return x->p = inf, 0; if (x->m == y->m) x->p = x->c > y->c ? inf : -inf; else x->p = div(y->c - x->c, x->m - y ->m); return x->p >= y->p; } void add(LL m, LL c) { auto z = insert({m, c, 0}), y = z++, x = y; while (isect(y, z)) z = erase(z); if (x != begin() && isect(--x, y)) isect(x, y = erase(y)); while ((y = x) != begin() && (-- x)->p >= y->p) isect(x, erase(y)); } LL query(LL x) { assert(!empty()); auto l = *lower_bound(x); return l.m * x + l.c; } };
```

6.2 CatalanDp

```
const int nmax = 1e4 + 1; const int mod = 1000000007; int catalan[nmax + 1]; /* // comb formula: ((2n)Cn)-((2n )C(n-1)) = (1/(n+1))*((2n)Cn) */ void genCatalan(int n) { catalan[0] = catalan[1] = 1; for (int i = 2; i <= n; i ++) { catalan[i] = 0; for (int j = 0; j < i; j++) { catalan[i] += (catalan[j] * catalan[i - j - 1]) % mod; if (catalan[i] >= mod) { catalan[i] -= mod; } } } }
```

6.3 DearrangementDP

```
const int nmax = 2e5 + 1; int drng[nmax + 1]; void gen_drng(int n) { drng[2] = 1; for (int i = 3; i <= n; i ++) { drng[i] = ((i - 1ll) * ((drng[i - 2] + drng[i - 1]) % mod)) % mod; } }
```

6.4 Li Chao Tree

```
const ll inf = 2e18; struct Line { ll m, c; ll eval(ll x ) { return m * x + c; } }; struct node { Line line; node * left = nullptr; node* right = nullptr; node(Line line) : line(line) {} void add_segment(Line nw, int l, int r, int L, int R) { if (l > r || r < L || l > R) return;
```

```
int m = (l + 1 == r ? l : (l + r) / 2); if (l >= L and r <= R) { bool lef = nw.eval(l) < line.eval(l); bool mid = nw.eval(m) < line.eval(m); if (mid) swap(line, nw); if (l == r) return; if (lef != mid) { if (left == nullptr) left = new node(nw); else left -> add_segment(nw, l, m, L, R); } else { if (right == nullptr) right = new node(nw); else right -> add_segment(nw, m + 1, r, L, R); } return; } if (max(l, L) <= min(m, R)) { if (left == nullptr) left = new node({0, inf}); left -> add_segment(nw, l, m, L, R); } if (max(m + 1, L) <= min(r, R)) { if (right == nullptr) right = new node ({0, inf}); right -> add_segment(nw, m + 1, r, L, R); } } ll query_segment(ll x, int l, int r, int L, int R) { if (l > r || r < L || l > R) return inf; int m = (l + 1 == r ? l : (l + r) / 2); if (l >= L and r <= R) { ll ans = line.eval(x); if (l < r) { if (x <= m && left != nullptr) ans = min(ans, left -> query_segment(x, l, m, L, R)); if (x > m && right != nullptr) ans = min(ans, right -> query_segment(x, m + 1, r, L, R)); } return ans; } ll ans = inf; if (max(l, L) <= min(m, R)) { if (left == nullptr) left = new node({0, inf}); ans = min(ans, left -> query_segment(x, l, m, L, R)); } if (max(m + 1, L) <= min(r, R)) { if (right == nullptr) right = new node ({0, inf}); ans = min(ans, right -> query_segment(x, m + 1, r, L, R)); } return ans; } }; struct LiChaoTree { int L, R; node* root; LiChaoTree() : L(numeric_limits<int>::min() / 2), R(numeric_limits<int>::max() / 2), root(nullptr) {} LiChaoTree(int L, int R) : L(L), R(R) { root = new node ({0, inf}); } void add_line(Line line) { root -> add_segment(line, L, R, L, R); } /* y = mx + b: x in [l, r] */ void add_segment(Line line, int l, int r) { root -> add_segment(line, L, R, l, r); } ll query(ll x) { return root -> query_segment(x, L, R, L, R); } ll query_segment(ll x, int l, int r) { return root -> query_segment(x, l, r, L, R); } }; int32_t main() { ios_base::sync_with_stdio(0); cin.tie(0); LiChaoTree t = LiChaoTree((int)-1e9, (int) 1e9); int n, q; cin >> n >> q; for (int i = 0; i < n; i++) { ll l, r, a, b; cin >> l >> r >> a >> b; r--; t.add_segment({a, b}, l, r); } while (q--) { int ty; cin >> ty; if (ty == 0) { ll l, r, a, b; cin >> l >> r >> a >> b; r--; t.add_segment({a, b}, l, r); } else { ll x; cin >> x; ll ans = t.query(x); if (ans >= inf) cout << "INFINITY\n"; else cout << ans << '\n'; } }}
```

6.5 SOS

```
/* Given a fixed array A of 2^N integers, we need to calculate for all x function F(x) = Sum of all A[i] such that x&i = i, i.e., i is a subset of x. */ /* // iterative version */ for(int mask = 0; mask < (1<<N); ++mask){ dp[mask][-1] = A[mask]; /* //handle base case separately (leaf states) */ for(int i = 0;i < N; ++i){ if(mask & (1<<i)) dp[mask][i] = dp[mask][i-1] + dp[mask ^(1<<i)][i-1]; else dp[mask][i] = dp[mask][i-1]; } F[mask] = dp[mask][N-1]; } /* //memory optimized, super easy to code. */ for(int i = 0; i<(1<<N); ++i) F[i] = A[i]; for(int i = 0;i < N; ++i) for(int mask = 0; mask < (1<<N); ++mask){ if(mask & (1<<i)) F[mask] += F[mask ^(1<<i)]; }
```

6.6 grundy

```
/* single pile game-> greedy or game dp multiple pile game and disjunctive(before playing, choose 1 pile) -> NIM game else-> Grundy(converts n any game piles to n NIM piles) grundy(x)->the smallest nonreachable grundy value there are n pile of games and k type of moves. if XOR(grundy(games)) == 0: losing state else winning state */ vector<int> moves, dp; int mex(vector<int> &a) { set<int> b(a.begin(), a.end()); for (int i = 0; ; ++i) if (!b.count(i)) return i; } int grundy(int x) { if (dp[x] != -1) return dp[x]; vector<int> reachable; for (auto m : moves) { if (x - m < 0) continue; int val = grundy(x - m); reachable.push_back(val); } return dp[x] = mex(reachable); }
```

7 Geometry

7.1 2D Shohag

```
const int N = 3e5 + 9; const double inf = 1e100; const double eps = 1e-9; const double PI = acos((double)-1.0); int sign(double x) { return (x > eps) - (x < -eps); } struct PT { double x, y; PT() { x = 0, y = 0; } PT(double x, double y) : x(x), y(y) {} PT(const PT &p) : x(p.x), y(p.y) {} PT operator + (const PT &a) const { return PT(x + a.x, y + a.y); } PT operator - (const PT &a) const { return PT(x - a.x, y - a.y); } PT operator * (const double a) const { return PT(x * a, y * a); } friend PT operator * (const double &a, const PT &b) { return PT(a * b.x, a * b.y); } PT operator / (const double a) const { return PT(x / a, y / a); } bool operator == (PT a) const { return sign(a.x - x) == 0 && sign(a.y - y) == 0; } bool operator != (PT a) const { return !(*this == a); } bool operator < (PT a) const { return sign(a.x - x) == 0 ? y < a.y : x < a.x; } bool operator > (PT a) const { return sign(a.x - x) == 0 ? y > a.y : x > a.x; } double norm() { return sqrt(x * x + y * y); } double norm2() { return x * x + y * y; } PT perp() { return PT(-y, x); } double arg() { return atan2(y, x); } PT truncate(double r) { /* returns a vector with norm r and having same direction */ double k = norm(); if (!sign(k)) return *this; r /= k; return PT(x * r, y * r); } }; istream &operator >> (istream &in, PT &p) { return in >> p.x >> p.y; } ostream &operator << (ostream &out, PT &p) { return out << "(" << p.x << "," << p.y << ")"; } inline double dot(PT a, PT b) { return a.x * b.x + a.y * b.y; } inline double dist2(PT a, PT b) { return dot(a - b, a - b); } inline double dist(PT a, PT b) { return sqrt(dot(a - b, a - b)); } inline double cross(PT a, PT b) { return a.x * b.y - a.y * b.x; } inline double cross2(PT a, PT b, PT c) { return cross(b - a, c - a); } inline int orientation(PT a, PT b, PT c) { return sign(cross(b - a, c - a)); } PT perp(PT a) { return PT(-a.y, a.x); } PT rotateccw90(PT a) { return PT(-a.y, a.x); } PT rotatecw90(PT a) { return PT(a.y, -a.x); } PT rotateccw(PT a, double t) { return PT(a.x * cos(t) - a.y * sin(t), a.x * sin(t) + a.y * cos(t)); } PT rotatecw(PT a, double t) { return PT(a.x * cos(t) + a.y * sin(t), -a.x * sin(t) + a.y * cos(t)); } double SQ(double x) { return x * x; } double rad_to_deg(double r) { return (r * 180.0 / PI); } double deg_to_rad(double d) { return (d * PI / 180.0); } double get_angle(PT a, PT b) { double costheta = dot(a, b) / a.norm() / b.norm(); return acos(max((double)-1.0, min((double)1.0, costheta))); } bool is_point_in_angle(PT b, PT a, PT c, PT p) { /* does point p lie in angle <bac */ assert(orientation(a, b, c) != 0); if (orientation(a, c, b) < 0) swap(b, c); return orientation(a, c, p) >= 0 && orientation(a, b, p) <= 0; } bool half(PT p) { return p.y > 0.0 || (p.y == 0.0 && p.x < 0.0); } void polar_sort(vector<PT> &v) { /* sort points in counterclockwise */ sort(v.begin(), v.end(), [](PT a,PT b) { return make_tuple(half(a), 0.0, a.norm2()) < make_tuple(half(b), cross(a, b), b.norm2()); }); } void polar_sort(vector<PT> &v, PT o) { /* sort points in counterclockwise with respect to point o */ sort(v.begin(), v.end(), [&](PT a,PT b) { return make_tuple(half(a - o), 0.0, (a - o).norm2()) < make_tuple(half(b - o), cross(a - o, b - o), (b - o).norm2()); }); }
```

```
struct line { PT a, b; /* goes through points a and b */
 PT v; double c; /* line form: direction vec [cross] (x,
 y) = c */ line() {} /* direction vector v and offset c
*/ line(PT v, double c) : v(v), c(c) { auto p =
get_points(); a = p.first; b = p.second; } /* equation
ax + by + c = 0 */ line(double _a, double _b, double _c)
 : v({_b, -_a}), c(-_c) { auto p = get_points(); a = p.
first; b = p.second; } /* goes through points p and q */
 line(PT p, PT q) : v(q - p), c(cross(v, p)), a(p), b(q)
 {} pair<PT, PT> get_points() { /* extract any two
points from this line */ PT p, q; double a = -v.y, b = v
.x; /* ax + by = c */ if (sign(a) == 0) { p = PT(0, c /
b); q = PT(1, c / b); } else if (sign(b) == 0) { p = PT(
c / a, 0); q = PT(c / a, 1); } else { p = PT(0, c / b);
q = PT(1, (c - a) / b); } return {p, q}; } /* ax + by +
c = 0 */ array<double, 3> get_abc() { double a = -v.y, b
 = v.x; return {a, b, -c}; } /* 1 if on the left, -1 if
on the right, 0 if on the line */ int side(PT p) {
return sign(cross(v, p) - c); } /* line that is
perpendicular to this and goes through point p */ line
perpendicular_through(PT p) { return {p, p + perp(v)}; }
 /* translate the line by vector t i.e. shifting it by
vector t */ line translate(PT t) { return {v, c + cross(
v, t)}; } /* compare two points by their orthogonal
projection on this line */ /* a projection point comes
before another if it comes first according to vector v
*/ bool cmp_by_projection(PT p, PT q) { return dot(v, p)
 < dot(v, q); } line shift_left(double d) { PT z = v.
perp().truncate(d); return line(a + z, b + z); } }; /*
find a point from a through b with distance d */ PT
point_along_line(PT a, PT b, double d) { assert(a != b);
 return a + (((b - a) / (b - a).norm()) * d); } /*
projection point c onto line through a and b assuming a
!= b */ PT project_from_point_to_line(PT a, PT b, PT c)
{ return a + (b - a) * dot(c - a, b - a) / (b - a).norm2
(); } /* reflection point c onto line through a and b
assuming a != b */ PT reflection_from_point_to_line(PT a
, PT b, PT c) { PT p = project_from_point_to_line(a,b,c)
; return p + p - c; } /* minimum distance from point c
to line through a and b */ double
dist_from_point_to_line(PT a, PT b, PT c) { return fabs(
cross(b - a, c - a) / (b - a).norm()); } /* returns true
 if point p is on line segment ab */ bool
is_point_on_seg(PT a, PT b, PT p) { if (fabs(cross(p - b
, a - b)) < eps) { if (p.x < min(a.x, b.x) - eps || p.x
> max(a.x, b.x) + eps) return false; if (p.y < min(a.y,
b.y) - eps || p.y > max(a.y, b.y) + eps) return false;
return true; } return false; } /* minimum distance point
 from point c to segment ab that lies on segment ab */
PT project_from_point_to_seg(PT a, PT b, PT c) { double
r = dist2(a, b); if (sign(r) == 0) return a; r = dot(c -
 a, b - a) / r; if (r < 0) return a; if (r > 1) return b
; return a + (b - a) * r; } /* minimum distance from
point c to segment ab */ double dist_from_point_to_seg(
PT a, PT b, PT c) { return dist(c,
project_from_point_to_seg(a, b, c)); } /* 0 if not
parallel, 1 if parallel, 2 if collinear */ int
is_parallel(PT a, PT b, PT c, PT d) { double k = fabs(
cross(b - a, d - c)); if (k < eps){ if (fabs(cross(a - b
, a - c)) < eps && fabs(cross(c - d, c - a)) < eps)
return 2; else return 1; } else return 0; } /* check if
two lines are same */ bool are_lines_same(PT a, PT b, PT
 c, PT d) { if (fabs(cross(a - c, c - d)) < eps && fabs(
cross(b - c, c - d)) < eps) return true; return false; }
 /* bisector vector of <abc */ PT angle_bisector(PT &a,
PT &b, PT &c){ PT p = a - b, q = c - b; return p + q *
sqrt(dot(p, p) / dot(q, q)); } /* 1 if point is ccw to
the line, 2 if point is cw to the line, 3 if point is on
 the line */ int point_line_relation(PT a, PT b, PT p) {
 int c = sign(cross(p - a, b - a)); if (c < 0) return 1;
 if (c > 0) return 2; return 3; } /* intersection point
between ab and cd assuming unique intersection exists */
 bool line_line_intersection(PT a, PT b, PT c, PT d, PT
&ans) { double a1 = a.y - b.y, b1 = b.x - a.x, c1 =
cross(a, b); double a2 = c.y - d.y, b2 = d.x - c.x, c2 =
 cross(c, d); double det = a1 * b2 - a2 * b1; if (det ==
 0) return 0; ans = PT((b1 * c2 - b2 * c1) / det, (c1 *
a2 - a1 * c2) / det); return 1; } /* intersection point
between segment ab and segment cd assuming unique
intersection exists */ bool seg_seg_intersection(PT a,
PT b, PT c, PT d, PT &ans) { double oa = cross2(c, d, a)
, ob = cross2(c, d, b); double oc = cross2(a, b, c), od
= cross2(a, b, d); if (oa * ob < 0 && oc * od < 0){ ans
= (a * ob - b * oa) / (ob - oa); return 1; } else return
 0; } /* intersection point between segment ab and
segment cd assuming unique intersection may not exists
*/ /* se.size()==0 means no intersection */ /* se.size()
==1 means one intersection */ /* se.size()==2 means
range intersection */ set<PT>
seg_seg_intersection_inside(PT a, PT b, PT c, PT d) { PT
 ans; if (seg_seg_intersection(a, b, c, d, ans)) return
{ans}; set<PT> se; if (is_point_on_seg(c, d, a)) se.
insert(a); if (is_point_on_seg(c, d, b)) se.insert(b);
if (is_point_on_seg(a, b, c)) se.insert(c); if (
is_point_on_seg(a, b, d)) se.insert(d); return se; } /*
intersection between segment ab and line cd */ /* 0 if
do not intersect, 1 if proper intersect, 2 if segment
intersect */ int seg_line_relation(PT a, PT b, PT c, PT
d) { double p = cross2(c, d, a); double q = cross2(c, d,
 b); if (sign(p) == 0 && sign(q) == 0) return 2; else if
 (p * q < 0) return 1; else return 0; } /* intersection
between segament ab and line cd assuming unique
intersection exists */ bool seg_line_intersection(PT a,
PT b, PT c, PT d, PT &ans) { bool k = seg_line_relation(
a, b, c, d); assert(k != 2); if (k)
line_line_intersection(a, b, c, d, ans); return k; } /*
minimum distance from segment ab to segment cd */ double
 dist_from_seg_to_seg(PT a, PT b, PT c, PT d) { PT dummy
; if (seg_seg_intersection(a, b, c, d, dummy)) return
0.0; else return min({dist_from_point_to_seg(a, b, c),
dist_from_point_to_seg(a, b, d), dist_from_point_to_seg(
c, d, a), dist_from_point_to_seg(c, d, b)}); } /*
minimum distance from point c to ray (starting point a
and direction vector b) */ double dist_from_point_to_ray
(PT a, PT b, PT c) { b = a + b; double r = dot(c - a, b
- a); if (r < 0.0) return dist(c, a); return
dist_from_point_to_line(a, b, c); } /* starting point as
 and direction vector ad */ bool ray_ray_intersection(PT
 as, PT ad, PT bs, PT bd) { double dx = bs.x - as.x, dy
= bs.y - as.y; double det = bd.x * ad.y - bd.y * ad.x;
if (fabs(det) < eps) return 0; double u = (dy * bd.x -
dx * bd.y) / det; double v = (dy * ad.x - dx * ad.y) /
det; if (sign(u) >= 0 && sign(v) >= 0) return 1; else
return 0; } double ray_ray_distance(PT as, PT ad, PT bs,
 PT bd) { if (ray_ray_intersection(as, ad, bs, bd))
return 0.0; double ans = dist_from_point_to_ray(as, ad,
bs); ans = min(ans, dist_from_point_to_ray(bs, bd, as));
 return ans; }
struct circle { PT p; double r; circle() {} circle(PT _p
, double _r): p(_p), r(_r) {}; /* center (x, y) and
radius r */ circle(double x, double y, double _r): p(PT(
x, y)), r(_r) {}; /* circumcircle of a triangle */ /*
the three points must be unique */ circle(PT a, PT b, PT
 c) { b = (a + b) * 0.5; c = (a + c) * 0.5;
line_line_intersection(b, b + rotatecw90(a - b), c, c +
rotatecw90(a - c), p); r = dist(a, p); } /* inscribed
circle of a triangle */ /* pass a bool just to
differentiate from circumcircle */ circle(PT a, PT b, PT
```

```
c, bool t) { line u, v; double m = atan2(b.y - a.y, b.x
- a.x), n = atan2(c.y - a.y, c.x - a.x); u.a = a; u.b =
u.a + (PT(cos((n + m)/2.0), sin((n + m)/2.0))); v.a = b
; m = atan2(a.y - b.y, a.x - b.x), n = atan2(c.y - b.y,
c.x - b.x); v.b = v.a + (PT(cos((n + m)/2.0), sin((n + m
)/2.0))); line_line_intersection(u.a, u.b, v.a, v.b, p);
r = dist_from_point_to_seg(a, b, p); } bool operator ==
(circle v) { return p == v.p && sign(r - v.r) == 0; }
double area() { return PI * r * r; } double
circumference() { return 2.0 * PI * r; } }; /* 0 if
outside, 1 if on circumference, 2 if inside circle */
int circle_point_relation(PT p, double r, PT b) { double
d = dist(p, b); if (sign(d - r) < 0) return 2; if (sign
(d - r) == 0) return 1; return 0; } /* 0 if outside, 1
if on circumference, 2 if inside circle */ int
circle_line_relation(PT p, double r, PT a, PT b) {
double d = dist_from_point_to_line(a, b, p); if (sign(d
- r) < 0) return 2; if (sign(d - r) == 0) return 1;
return 0; } /* compute intersection of line through
points a and b with */ /* circle centered at c with
radius r > 0 */ vector<PT> circle_line_intersection(PT c
, double r, PT a, PT b) { vector<PT> ret; b = b - a; a =
a - c; double A = dot(b, b), B = dot(a, b); double C =
dot(a, a) - r * r, D = B * B - A * C; if (D < -eps)
return ret; ret.push_back(c + a + b * (-B + sqrt(D + eps
)) / A); if (D > eps) ret.push_back(c + a + b * (-B -
sqrt(D)) / A); return ret; } /* 5 - outside and do not
intersect */ /* 4 - intersect outside in one point */ /*
3 - intersect in 2 points */ /* 2 - intersect inside in
one point */ /* 1 - inside and do not intersect */ int
circle_circle_relation(PT a, double r, PT b, double R) {
double d = dist(a, b); if (sign(d - r - R) > 0) return
5; if (sign(d - r - R) == 0) return 4; double l = fabs(r
- R); if (sign(d - r - R) < 0 && sign(d - l) > 0)
return 3; if (sign(d - l) == 0) return 2; if (sign(d - l
) < 0) return 1; assert(0); return -1; } vector<PT>
circle_circle_intersection(PT a, double r, PT b, double
R) { if (a == b && sign(r - R) == 0) return {PT(1e18, 1
e18)}; vector<PT> ret; double d = sqrt(dist2(a, b)); if
(d > r + R || d + min(r, R) < max(r, R)) return ret;
double x = (d * d - R * R + r * r) / (2 * d); double y =
sqrt(r * r - x * x); PT v = (b - a) / d; ret.push_back(
a + v * x + rotateccw90(v) * y); if (y > 0) ret.
push_back(a + v * x - rotateccw90(v) * y); return ret; }
/* returns two circle c1, c2 through points a, b and of
radius r */ /* 0 if there is no such circle, 1 if one
circle, 2 if two circle */ int get_circle(PT a, PT b,
double r, circle &c1, circle &c2) { vector<PT> v =
circle_circle_intersection(a, r, b, r); int t = v.size()
; if (!t) return 0; c1.p = v[0], c1.r = r; if (t == 2)
c2.p = v[1], c2.r = r; return t; } /* returns two circle
c1, c2 which is tangent to line u, goes through */ /*
point q and has radius r1; 0 for no circle, 1 if c1 = c2
, 2 if c1 != c2 */ int get_circle(line u, PT q, double
r1, circle &c1, circle &c2) { double d =
dist_from_point_to_line(u.a, u.b, q); if (sign(d - r1 *
2.0) > 0) return 0; if (sign(d) == 0) { cout << u.v.x <<
' ' << u.v.y << '\n'; c1.p = q + rotateccw90(u.v).
truncate(r1); c2.p = q + rotatecw90(u.v).truncate(r1);
c1.r = c2.r = r1; return 2; } line u1 = line(u.a +
rotateccw90(u.v).truncate(r1), u.b + rotateccw90(u.v).
truncate(r1)); line u2 = line(u.a + rotatecw90(u.v).
truncate(r1), u.b + rotatecw90(u.v).truncate(r1));
circle cc = circle(q, r1); PT p1, p2; vector<PT> v; v =
circle_line_intersection(q, r1, u1.a, u1.b); if (!v.size
()) v = circle_line_intersection(q, r1, u2.a, u2.b); v.
push_back(v[0]); p1 = v[0], p2 = v[1]; c1 = circle(p1,
r1); if (p1 == p2) { c2 = c1; return 1; } c2 = circle(p2
, r1); return 2; } /* returns the circle such that for
all points w on the circumference of the circle */ /*
dist(w, a) : dist(w, b) = rp : rq */ /* rp != rq */ /*
https:en.wikipedia.org/wiki/Circles_of_Apollonius */
circle get_apollonius_circle(PT p, PT q, double rp,
double rq ){ rq *= rq ; rp *= rp ; double a = rq - rp ;
assert(sign(a)); double g = rq * p.x - rp * q.x ; g /= a
; double h = rq * p.y - rp * q.y ; h /= a ; double c =
rq * p.x * p.x - rp * q.x * q.x + rq * p.y * p.y - rp *
q.y * q.y ; c /= a ; PT o(g, h); double r = g * g + h *
h - c ; r = sqrt(r); return circle(o,r); } /* returns
area of intersection between two circles */ double
circle_circle_area(PT a, double r1, PT b, double r2) {
double d = (a - b).norm(); if(r1 + r2 < d + eps) return
0; if(r1 + d < r2 + eps) return PI * r1 * r1; if(r2 + d
< r1 + eps) return PI * r2 * r2; double theta_1 = acos((
r1 * r1 + d * d - r2 * r2) / (2 * r1 * d)), theta_2 =
acos((r2 * r2 + d * d - r1 * r1)/(2 * r2 * d)); return
r1 * r1 * (theta_1 - sin(2 * theta_1)/2.) + r2 * r2 * (
theta_2 - sin(2 * theta_2)/2.); } /* tangent lines from
point q to the circle */ int tangent_lines_from_point(PT
p, double r, PT q, line &u, line &v) { int x = sign(
dist2(p, q) - r * r); if (x < 0) return 0; /* point in
cricle */ if (x == 0) { /* point on circle */ u = line(q
, q + rotateccw90(q - p)); v = u; return 1; } double d =
dist(p, q); double l = r * r / d; double h = sqrt(r * r
- l * l); u = line(q, p + ((q - p).truncate(l) + (
rotateccw90(q - p).truncate(h)))); v = line(q, p + ((q -
p).truncate(l) + (rotatecw90(q - p).truncate(h))));
return 2; } /* returns outer tangents line of two
circles */ /* if inner == 1 it returns inner tangent
lines */ int tangents_lines_from_circle(PT c1, double r1
, PT c2, double r2, bool inner, line &u, line &v) { if (
inner) r2 = -r2; PT d = c2 - c1; double dr = r1 - r2, d2
= d.norm2(), h2 = d2 - dr * dr; if (d2 == 0 || h2 < 0)
{ assert(h2 != 0); return 0; } vector<pair<PT, PT>>out;
for (int tmp: {- 1, 1}) { PT v = (d * dr + rotateccw90(d
) * sqrt(h2) * tmp) / d2; out.push_back({c1 + v * r1, c2
+ v * r2}); } u = line(out[0].first, out[0].second); if
(out.size() == 2) v = line(out[1].first, out[1].second)
; return 1 + (h2 > 0); } /* O(n^2 log n) */ /* https:
vjudge.net/problem/UVA-12056 */ struct CircleUnion { int
n; double x[2020], y[2020], r[2020]; int covered[2020];
vector<pair<double, double> > seg, cover; double arc,
pol; inline int sign(double x) {return x < -eps ? -1 : x
> eps;} inline int sign(double x, double y) {return
sign(x - y);} inline double SQ(const double x) {return x
* x;} inline double dist(double x1, double y1, double
x2, double y2) {return sqrt(SQ(x1 - x2) + SQ(y1 - y2));}
inline double angle(double A, double B, double C) {
double val = (SQ(A) + SQ(B) - SQ(C)) / (2 * A * B); if (
val < -1) val = -1; if (val > +1) val = +1; return acos(
val); } CircleUnion() { n = 0; seg.clear(), cover.clear
(); arc = pol = 0; } void init() { n = 0; seg.clear(),
cover.clear(); arc = pol = 0; } void add(double xx,
double yy, double rr) { x[n] = xx, y[n] = yy, r[n] = rr,
covered[n] = 0, n++; } void getarea(int i, double lef,
double rig) { arc += 0.5 * r[i] * r[i] * (rig - lef -
sin(rig - lef)); double x1 = x[i] + r[i] * cos(lef), y1
= y[i] + r[i] * sin(lef); double x2 = x[i] + r[i] * cos(
rig), y2 = y[i] + r[i] * sin(rig); pol += x1 * y2 - x2 *
y1; } double solve() { for (int i = 0; i < n; i++) {
for (int j = 0; j < i; j++) { if (!sign(x[i] - x[j]) &&
!sign(y[i] - y[j]) && !sign(r[i] - r[j])) { r[i] = 0.0;
break; } } } for (int i = 0; i < n; i++) { for (int j =
0; j < n; j++) { if (i != j && sign(r[j] - r[i]) >= 0 &&
sign(dist(x[i], y[i], x[j], y[j]) - (r[j] - r[i])) <=
0) { covered[i] = 1; break; } } } for (int i = 0; i < n;
i++) { if (sign(r[i]) && !covered[i]) { seg.clear();
for (int j = 0; j < n; j++) { if (i != j) { double d =
```

```
dist(x[i], y[i], x[j], y[j]); if (sign(d - (r[j] + r[i])
) >= 0 || sign(d - abs(r[j] - r[i])) <= 0) { continue; }
 double alpha = atan2(y[j] - y[i], x[j] - x[i]); double
beta = angle(r[i], d, r[j]); pair<double, double> tmp(
alpha - beta, alpha + beta); if (sign(tmp.first) <= 0 &&
 sign(tmp.second) <= 0) { seg.push_back(pair<double,
double>(2 * PI + tmp.first, 2 * PI + tmp.second)); }
else if (sign(tmp.first) < 0) { seg.push_back(pair<
double, double>(2 * PI + tmp.first, 2 * PI)); seg.
push_back(pair<double, double>(0, tmp.second)); } else {
 seg.push_back(tmp); } } } sort(seg.begin(), seg.end());
 double rig = 0; for (vector<pair<double, double> >::
iterator iter = seg.begin(); iter != seg.end(); iter++)
{ if (sign(rig - iter->first) >= 0) { rig = max(rig,
iter->second); } else { getarea(i, rig, iter->first);
rig = iter->second; } } if (!sign(rig)) { arc += r[i] *
r[i] * PI; } else { getarea(i, rig, 2 * PI); } } }
return pol / 2.0 + arc; } } CU; double area_of_triangle(
PT a, PT b, PT c) { return fabs(cross(b - a, c - a) *
0.5); } /* -1 if strictly inside, 0 if on the polygon, 1
 if strictly outside */ int is_point_in_triangle(PT a,
PT b, PT c, PT p) { if (sign(cross(b - a,c - a)) < 0)
swap(b, c); int c1 = sign(cross(b - a,p - a)); int c2 =
sign(cross(c - b,p - b)); int c3 = sign(cross(a - c,p -
c)); if (c1<0 || c2<0 || c3 < 0) return 1; if (c1 + c2 +
 c3 != 3) return 0; return -1; } double perimeter(vector
<PT> &p) { double ans=0; int n = p.size(); for (int i =
0; i < n; i++) ans += dist(p[i], p[(i + 1) % n]); return
 ans; } double area(vector<PT> &p) { double ans = 0; int
 n = p.size(); for (int i = 0; i < n; i++) ans += cross(
p[i], p[(i + 1) % n]); return fabs(ans) * 0.5; } /*
centroid of a (possibly non-convex) polygon, */ /*
assuming that the coordinates are listed in a clockwise
or */ /* counterclockwise fashion. Note that the
centroid is often known as */ /* the "center of gravity"
 or "center of mass". */ PT centroid(vector<PT> &p) {
int n = p.size(); PT c(0, 0); double sum = 0; for (int i
 = 0; i < n; i++) sum += cross(p[i], p[(i + 1) % n]);
double scale = 3.0 * sum; for (int i = 0; i < n; i++) {
int j = (i + 1) % n; c = c + (p[i] + p[j]) * cross(p[i],
 p[j]); } return c / scale; } /* 0 if cw, 1 if ccw */
bool get_direction(vector<PT> &p) { double ans = 0; int
n = p.size(); for (int i = 0; i < n; i++) ans += cross(p
[i], p[(i + 1) % n]); if (sign(ans) > 0) return 1;
return 0; } /* it returns a point such that the sum of
distances */ /* from that point to all points in p is
minimum */ /* O(n log^2 MX) */ PT geometric_median(
vector<PT> p) { auto tot_dist = [&](PT z) { double res =
 0; for (int i = 0; i < p.size(); i++) res += dist(p[i],
 z); return res; }; auto findY = [&](double x) { double
yl = -1e5, yr = 1e5; for (int i = 0; i < 60; i++) {
double ym1 = yl + (yr - yl) / 3; double ym2 = yr - (yr -
 yl) / 3; double d1 = tot_dist(PT(x, ym1)); double d2 =
tot_dist(PT(x, ym2)); if (d1 < d2) yr = ym2; else yl =
ym1; } return pair<double, double> (yl, tot_dist(PT(x,
yl))); }; double xl = -1e5, xr = 1e5; for (int i = 0; i
< 60; i++) { double xm1 = xl + (xr - xl) / 3; double xm2
 = xr - (xr - xl) / 3; double y1, d1, y2, d2; auto z =
findY(xm1); y1 = z.first; d1 = z.second; z = findY(xm2);
 y2 = z.first; d2 = z.second; if (d1 < d2) xr = xm2;
else xl = xm1; } return {xl, findY(xl).first }; }
vector<PT> convex_hull(vector<PT> &p) { if (p.size() <=
1) return p; vector<PT> v = p; sort(v.begin(), v.end());
 vector<PT> up, dn; for (auto& p : v) { while (up.size()
 > 1 && orientation(up[up.size() - 2], up.back(), p) >=
0) { up.pop_back(); } while (dn.size() > 1 &&
orientation(dn[dn.size() - 2], dn.back(), p) <= 0) { dn.
pop_back(); } up.push_back(p); dn.push_back(p); } v = dn
; if (v.size() > 1) v.pop_back(); reverse(up.begin(), up
.end()); up.pop_back(); for (auto& p : up) { v.push_back
(p); } if (v.size() == 2 && v[0] == v[1]) v.pop_back();
return v; } /* checks if convex or not */ bool is_convex
(vector<PT> &p) { bool s[3]; s[0] = s[1] = s[2] = 0; int
 n = p.size(); for (int i = 0; i < n; i++) { int j = (i
+ 1) % n; int k = (j + 1) % n; s[sign(cross(p[j] - p[i],
 p[k] - p[i])) + 1] = 1; if (s[0] && s[2]) return 0; }
return 1; } /* -1 if strictly inside, 0 if on the
polygon, 1 if strictly outside */ /* it must be strictly
 convex, otherwise make it strictly convex first */ int
is_point_in_convex(vector<PT> &p, const PT& x) { /* O(
log n) */ int n = p.size(); assert(n >= 3); int a =
orientation(p[0], p[1], x), b = orientation(p[0], p[n -
1], x); if (a < 0 || b > 0) return 1; int l = 1, r = n -
 1; while (l + 1 < r) { int mid = l + r >> 1; if (
orientation(p[0], p[mid], x) >= 0) l = mid; else r = mid
; } int k = orientation(p[l], p[r], x); if (k <= 0)
return -k; if (l == 1 && a == 0) return 0; if (r == n -
1 && b == 0) return 0; return -1; } bool
is_point_on_polygon(vector<PT> &p, const PT& z) { int n
= p.size(); for (int i = 0; i < n; i++) { if (
is_point_on_seg(p[i], p[(i + 1) % n], z)) return 1; }
return 0; } /* returns 1e9 if the point is on the
polygon */ int winding_number(vector<PT> &p, const PT& z
) { /* O(n) */ if (is_point_on_polygon(p, z)) return 1e9
; int n = p.size(), ans = 0; for (int i = 0; i < n; ++i)
 { int j = (i + 1) % n; bool below = p[i].y < z.y; if (
below != (p[j].y < z.y)) { auto orient = orientation(z,
p[j], p[i]); if (orient == 0) return 0; if (below == (
orient > 0)) ans += below ? 1 : -1; } } return ans; } /*
 -1 if strictly inside, 0 if on the polygon, 1 if
strictly outside */ int is_point_in_polygon(vector<PT> &
p, const PT& z) { /* O(n) */ int k = winding_number(p, z
); return k == 1e9 ? 0 : k == 0 ? 1 : -1; } /* id of the
 vertex having maximum dot product with z */ /* polygon
must need to be convex */ /* top - upper right vertex */
 /* for minimum dot product negate z and return -dot(z,
p[id]) */ int extreme_vertex(vector<PT> &p, const PT &z,
 const int top) { /* O(log n) */ int n = p.size(); if (n
 == 1) return 0; double ans = dot(p[0], z); int id = 0;
if (dot(p[top], z) > ans) ans = dot(p[top], z), id = top
; int l = 1, r = top - 1; while (l < r) { int mid = l +
r >> 1; if (dot(p[mid + 1], z) >= dot(p[mid], z)) l =
mid + 1; else r = mid; } if (dot(p[l], z) > ans) ans =
dot(p[l], z), id = l; l = top + 1, r = n - 1; while (l <
 r) { int mid = l + r >> 1; if (dot(p[(mid + 1) % n], z)
 >= dot(p[mid], z)) l = mid + 1; else r = mid; } l %= n;
 if (dot(p[l], z) > ans) ans = dot(p[l], z), id = l;
return id; } /* maximum distance from any point on the
perimeter to another point on the perimeter */ double
diameter(vector<PT> &p) { int n = (int)p.size(); if (n
== 1) return 0; if (n == 2) return dist(p[0], p[1]);
double ans = 0; int i = 0, j = 1; while (i < n) { while
(cross(p[(i + 1) % n] - p[i], p[(j + 1) % n] - p[j]) >=
0) { ans = max(ans, dist2(p[i], p[j])); j = (j + 1) % n;
 } ans = max(ans, dist2(p[i], p[j])); i++; } return sqrt
(ans); } /* minimum distance between two parallel lines
(non necessarily axis parallel) */ /* such that the
polygon can be put between the lines */ double width(
vector<PT> &p) { int n = (int)p.size(); if (n <= 2)
return 0; double ans = inf; int i = 0, j = 1; while (i <
 n) { while (cross(p[(i + 1) % n] - p[i], p[(j + 1) % n]
 - p[j]) >= 0) j = (j + 1) % n; ans = min(ans,
dist_from_point_to_line(p[i], p[(i + 1) % n], p[j])); i
++; } return ans; } /* minimum perimeter */ double
minimum_enclosing_rectangle(vector<PT> &p) { int n = p.
size(); if (n <= 2) return perimeter(p); int mndot = 0;
double tmp = dot(p[1] - p[0], p[0]); for (int i = 1; i <
 n; i++) { if (dot(p[1] - p[0], p[i]) <= tmp) { tmp =
```

```
dot(p[1] - p[0], p[i]); mndot = i; } } double ans = inf;
 int i = 0, j = 1, mxdot = 1; while (i < n) { PT cur = p
[(i + 1) % n] - p[i]; while (cross(cur, p[(j + 1) % n] -
 p[j]) >= 0) j = (j + 1) % n; while (dot(p[(mxdot + 1) %
 n], cur) >= dot(p[mxdot], cur)) mxdot = (mxdot + 1) % n
; while (dot(p[(mndot + 1) % n], cur) <= dot(p[mndot],
cur)) mndot = (mndot + 1) % n; ans = min(ans, 2.0 * ((
dot(p[mxdot], cur) / cur.norm() - dot(p[mndot], cur) /
cur.norm()) + dist_from_point_to_line(p[i], p[(i + 1) %
n], p[j]))); i++; } return ans; } /* given n points,
find the minimum enclosing circle of the points */ /*
call convex_hull() before this for faster solution */ /*
 expected O(n) */ random_device rd; mt19937 g(rd());
circle minimum_enclosing_circle(vector<PT> &p) { shuffle
(p.begin(), p.end(), g); int n = p.size(); circle c(p
[0], 0); for (int i = 1; i < n; i++) { if (sign(dist(c.p
, p[i]) - c.r) > 0) { c = circle(p[i], 0); for (int j =
0; j < i; j++) { if (sign(dist(c.p, p[j]) - c.r) > 0) {
c = circle((p[i] + p[j]) / 2, dist(p[i], p[j]) / 2); for
 (int k = 0; k < j; k++) { if (sign(dist(c.p, p[k]) - c.
r) > 0) { c = circle(p[i], p[j], p[k]); } } } } } }
return c; } /* returns a vector with the vertices of a
polygon with everything */ /* to the left of the line
going from a to b cut away. */ vector<PT> cut(vector<PT>
 &p, PT a, PT b) { vector<PT> ans; int n = (int)p.size()
; for (int i = 0; i < n; i++) { double c1 = cross(b - a,
 p[i] - a); double c2 = cross(b - a, p[(i + 1) % n] - a)
; if (sign(c1) >= 0) ans.push_back(p[i]); if (sign(c1 *
c2) < 0) { if (!is_parallel(p[i], p[(i + 1) % n], a, b))
 { PT tmp; line_line_intersection(p[i], p[(i + 1) % n],
a, b, tmp); ans.push_back(tmp); } } } return ans; } /*
not necessarily convex, boundary is included in the
intersection */ /* returns total intersected length */
/* it returns the sum of the lengths of the portions of
the line that are inside the polygon */ double
polygon_line_intersection(vector<PT> p, PT a, PT b) {
int n = p.size(); p.push_back(p[0]); line l = line(a, b)
; double ans = 0.0; vector< pair<double, int> > vec; for
 (int i = 0; i < n; i++) { int s1 = orientation(a, b, p[
i]); int s2 = orientation(a, b, p[i + 1]); if (s1 == s2)
 continue; line t = line(p[i], p[i + 1]); PT inter = (t.
v * l.c - l.v * t.c) / cross(l.v, t.v); double tmp = dot
(inter, l.v); int f; if (s1 > s2) f = s1 && s2 ? 2 : 1;
else f = s1 && s2 ? -2 : -1; vec.push_back(make_pair((f
> 0 ? tmp - eps : tmp + eps), f)); /* keep eps very
small like 1e-12 */ } sort(vec.begin(), vec.end()); for
(int i = 0, j = 0; i + 1 < (int)vec.size(); i++){ j +=
vec[i].second; if (j) ans += vec[i + 1].first - vec[i].
first; /* if this portion is inside the polygon */ /*
else ans = 0; if we want the maximum intersected length
which is totally inside the polygon, uncomment this and
take the maximum of ans */ } ans = ans / sqrt(dot(l.v, l
.v)); p.pop_back(); return ans; } /* given a convex
polygon p, and a line ab and the top vertex of the
polygon */ /* returns the intersection of the line with
the polygon */ /* it returns the indices of the edges of
 the polygon that are intersected by the line */ /* so
if it returns i, then the line intersects the edge (p[i
], p[(i + 1) % n]) */ array<int, 2>
convex_line_intersection(vector<PT> &p, PT a, PT b, int
top) { int end_a = extreme_vertex(p, (a - b).perp(), top
); int end_b = extreme_vertex(p, (b - a).perp(), top);
auto cmp_l = [&](int i) { return orientation(a, p[i], b)
; }; if (cmp_l(end_a) < 0 || cmp_l(end_b) > 0) return
{-1, -1}; /* no intersection */ array<int, 2> res; for (
int i = 0; i < 2; i++) { int lo = end_b, hi = end_a, n =
 p.size(); while ((lo + 1) % n != hi) { int m = ((lo +
hi + (lo < hi ? 0 : n)) / 2) % n; (cmp_l(m) == cmp_l(
end_b) ? lo : hi) = m; } res[i] = (lo + !cmp_l(hi)) % n;
 swap(end_a, end_b); } if (res[0] == res[1]) return {res
[0], -1}; /* touches the vertex res[0] */ if (!cmp_l(res
[0]) && !cmp_l(res[1])) switch ((res[0] - res[1] + (int)
p.size() + 1) % p.size()) { case 0: return {res[0], res
[0]}; /* touches the edge (res[0], res[0] + 1) */ case
2: return {res[1], res[1]}; /* touches the edge (res[1],
 res[1] + 1) */ } return res; /* intersects the edges (
res[0], res[0] + 1) and (res[1], res[1] + 1) */ } pair<
PT, int> point_poly_tangent(vector<PT> &p, PT Q, int dir
, int l, int r) { while (r - l > 1) { int mid = (l + r)
>> 1; bool pvs = orientation(Q, p[mid], p[mid - 1]) != -
dir; bool nxt = orientation(Q, p[mid], p[mid + 1]) != -
dir; if (pvs && nxt) return {p[mid], mid}; if (!(pvs ||
nxt)) { auto p1 = point_poly_tangent(p, Q, dir, mid + 1,
 r); auto p2 = point_poly_tangent(p, Q, dir, l, mid - 1)
; return orientation(Q, p1.first, p2.first) == dir ? p1
: p2; } if (!pvs) { if (orientation(Q, p[mid], p[l]) ==
dir) r = mid - 1; else if (orientation(Q, p[l], p[r]) ==
 dir) r = mid - 1; else l = mid + 1; } if (!nxt) { if (
orientation(Q, p[mid], p[l]) == dir) l = mid + 1; else
if (orientation(Q, p[l], p[r]) == dir) r = mid - 1; else
 l = mid + 1; } } pair<PT, int> ret = {p[l], l}; for (
int i = l + 1; i <= r; i++) ret = orientation(Q, ret.
first, p[i]) != dir ? make_pair(p[i], i) : ret; return
ret; } /* (ccw, cw) tangents from a point that is
outside this convex polygon */ /* returns indexes of the
 points */ /* ccw means the tangent from Q to that point
 is in the same direction as the polygon ccw direction
*/ pair<int, int> tangents_from_point_to_polygon(vector<
PT> &p, PT Q){ int ccw = point_poly_tangent(p, Q, 1, 0,
(int)p.size() - 1).second; int cw = point_poly_tangent(p
, Q, -1, 0, (int)p.size() - 1).second; return make_pair(
ccw, cw); } /* minimum distance from a point to a convex
 polygon */ /* it assumes point lie strictly outside the
 polygon */ double dist_from_point_to_polygon(vector<PT>
 &p, PT z) { double ans = inf; int n = p.size(); if (n
<= 3) { for(int i = 0; i < n; i++) ans = min(ans,
dist_from_point_to_seg(p[i], p[(i + 1) % n], z)); return
 ans; } auto [r, l] = tangents_from_point_to_polygon(p,
z); if(l > r) r += n; while (l < r) { int mid = (l + r)
>> 1; double left = dist2(p[mid % n], z), right= dist2(p
[(mid + 1) % n], z); ans = min({ans, left, right}); if(
left < right) r = mid; else l = mid + 1; } ans = sqrt(
ans); ans = min(ans, dist_from_point_to_seg(p[l % n], p
[(l + 1) % n], z)); ans = min(ans,
dist_from_point_to_seg(p[l % n], p[(l - 1 + n) % n], z))
; return ans; } /* minimum distance from convex polygon
p to line ab */ /* returns 0 is it intersects with the
polygon */ /* top - upper right vertex */ double
dist_from_polygon_to_line(vector<PT> &p, PT a, PT b, int
 top) { /* O(log n) */ PT orth = (b - a).perp(); if (
orientation(a, b, p[0]) > 0) orth = (a - b).perp(); int
id = extreme_vertex(p, orth, top); if (dot(p[id] - a,
orth) > 0) return 0.0; /* if orth and a are in the same
half of the line, then poly and line intersects */
return dist_from_point_to_line(a, b, p[id]); /* does not
 intersect */ } /* minimum distance from a convex
polygon to another convex polygon */ /* the polygon
doesnot overlap or touch */ /* tested in https:toph.co/p
/the-wall */ double dist_from_polygon_to_polygon(vector<
PT> &p1, vector<PT> &p2) { /* O(n log n) */ double ans =
 inf; for (int i = 0; i < p1.size(); i++) { ans = min(
ans, dist_from_point_to_polygon(p2, p1[i])); } for (int
i = 0; i < p2.size(); i++) { ans = min(ans,
dist_from_point_to_polygon(p1, p2[i])); } return ans; }
/* maximum distance from a convex polygon to another
convex polygon */ double
maximum_dist_from_polygon_to_polygon(vector<PT> &u,
vector<PT> &v){ /* O(n) */ int n = (int)u.size(), m = (
```

```
int)v.size(); double ans = 0; if (n < 3 || m < 3) { for
(int i = 0; i < n; i++) { for (int j = 0; j < m; j++)
ans = max(ans, dist2(u[i], v[j])); } return sqrt(ans); }
 if (u[0].x > v[0].x) swap(n, m), swap(u, v); int i = 0,
 j = 0, step = n + m + 10; while (j + 1 < m && v[j].x <
v[j + 1].x) j++ ; while (step--) { if (cross(u[(i + 1)%n
] - u[i], v[(j + 1)%m] - v[j]) >= 0) j = (j + 1) % m;
else i = (i + 1) % n; ans = max(ans, dist2(u[i], v[j]));
 } return sqrt(ans); } /* calculates the area of the
union of n polygons (not necessarily convex). */ /* the
points within each polygon must be given in CCW order.
*/ /* complexity: O(N^2), where N is the total number of
 points */ double rat(PT a, PT b, PT p) { return !sign(a
.x - b.x) ? (p.y - a.y) / (b.y - a.y) : (p.x - a.x) / (b
.x - a.x); }; double polygon_union(vector<vector<PT>> &p
) { int n = p.size(); double ans=0; for(int i = 0; i < n
; ++i) { for (int v = 0; v < (int)p[i].size(); ++v) { PT
 a = p[i][v], b = p[i][(v + 1) % p[i].size()]; vector<
pair<double, int>> segs; segs.emplace_back(0, 0), segs.
emplace_back(1, 0); for(int j = 0; j < n; ++j) { if(i !=
 j) { for(size_t u = 0; u < p[j].size(); ++u) { PT c = p
[j][u], d = p[j][(u + 1) % p[j].size()]; int sc = sign(
cross(b - a, c - a)), sd = sign(cross(b - a, d - a)); if
(!sc && !sd) { if(sign(dot(b - a, d - c)) > 0 && i > j)
{ segs.emplace_back(rat(a, b, c), 1), segs.emplace_back(
rat(a, b, d), -1); } } else { double sa = cross(d - c, a
 - c), sb = cross(d - c, b - c); if(sc >= 0 && sd < 0)
segs.emplace_back(sa / (sa - sb), 1); else if(sc < 0 &&
sd >= 0) segs.emplace_back(sa / (sa - sb), -1); } } } }
sort(segs.begin(), segs.end()); double pre = min(max(
segs[0].first, 0.0), 1.0), now, sum = 0; int cnt = segs
[0].second; for(int j = 1; j < segs.size(); ++j) { now =
 min(max(segs[j].first, 0.0), 1.0); if (!cnt) sum += now
 - pre; cnt += segs[j].second; pre = now; } ans += cross
(a, b) * sum; } } return ans * 0.5; }
/* contains all points p such that: cross(b - a, p - a)
 >= 0 */
struct HP { PT a, b; HP() {} HP(PT a, PT b) : a(a), b(b)
 {} HP(const HP& rhs) : a(rhs.a), b(rhs.b) {} int
operator < (const HP& rhs) const { PT p = b - a; PT q =
rhs.b - rhs.a; int fp = (p.y < 0 || (p.y == 0 && p.x <
0)); int fq = (q.y < 0 || (q.y == 0 && q.x < 0)); if (fp
 != fq) return fp == 0; if (cross(p, q)) return cross(p,
 q) > 0; return cross(p, rhs.b - a) < 0; } PT
line_line_intersection(PT a, PT b, PT c, PT d) { b = b -
 a; d = c - d; c = c - a; return a + b * cross(c, d) /
cross(b, d); } PT intersection(const HP &v) { return
line_line_intersection(a, b, v.a, v.b); } }; int check(
HP a, HP b, HP c) { return cross(a.b - a.a, b.
intersection(c) - a.a) > -eps; /* -eps to include
polygons of zero area (straight lines, points) */ } /*
consider half-plane of counter-clockwise side of each
line */ /* if lines are not bounded add infinity
rectangle */ /* returns a convex polygon, a point can
occur multiple times though */ /* complexity: O(n log(n)
) */ vector<PT> half_plane_intersection(vector<HP> h) {
sort(h.begin(), h.end()); vector<HP> tmp; for (int i =
0; i < h.size(); i++) { if (!i || cross(h[i].b - h[i].a,
 h[i - 1].b - h[i - 1].a)) { tmp.push_back(h[i]); } } h
= tmp; vector<HP> q(h.size() + 10); int qh = 0, qe = 0;
for (int i = 0; i < h.size(); i++) { while (qe - qh > 1
&& !check(h[i], q[qe - 2], q[qe - 1])) qe--; while (qe -
 qh > 1 && !check(h[i], q[qh], q[qh + 1])) qh++; q[qe++]
 = h[i]; } while (qe - qh > 2 && !check(q[qh], q[qe -
2], q[qe - 1])) qe--; while (qe - qh > 2 && !check(q[qe
- 1], q[qh], q[qh + 1])) qh++; vector<HP> res; for (int
i = qh; i < qe; i++) res.push_back(q[i]); vector<PT>
hull; if (res.size() > 2) { for (int i = 0; i < res.size
(); i++) { hull.push_back(res[i].intersection(res[(i +
1) % ((int)res.size())])); } } return hull; } /* rotate
the polygon such that the (bottom, left)-most point is
at the first position */ void reorder_polygon(vector<PT>
 &p) { int pos = 0; for (int i = 1; i < p.size(); i++) {
 if (p[i].y < p[pos].y || (sign(p[i].y - p[pos].y) == 0
&& p[i].x < p[pos].x)) pos = i; } rotate(p.begin(), p.
begin() + pos, p.end()); } /* a and b are convex
polygons */ /* returns a convex hull of their minkowski
sum */ /* min(a.size(), b.size()) >= 2 */ /* https:cp-
algorithms.com/geometry/minkowski.html */ vector<PT>
minkowski_sum(vector<PT> a, vector<PT> b) {
reorder_polygon(a); reorder_polygon(b); int n = a.size()
, m = b.size(); int i = 0, j = 0; a.push_back(a[0]); a.
push_back(a[1]); b.push_back(b[0]); b.push_back(b[1]);
vector<PT> c; while (i < n || j < m) { c.push_back(a[i]
+ b[j]); double p = cross(a[i + 1] - a[i], b[j + 1] - b[
j]); if (sign(p) >= 0) ++i; if (sign(p) <= 0) ++j; }
return c; } /* returns the area of the intersection of
the circle with center c and radius r */ /* and the
triangle formed by the points c, a, b */ double
_triangle_circle_intersection(PT c, double r, PT a, PT b
) { double sd1 = dist2(c, a), sd2 = dist2(c, b); if(sd1
> sd2) swap(a, b), swap(sd1, sd2); double sd = dist2(a,
b); double d1 = sqrtl(sd1), d2 = sqrtl(sd2), d = sqrt(sd
); double x = abs(sd2 - sd - sd1) / (2 * d); double h =
sqrtl(sd1 - x * x); if(r >= d2) return h * d / 2; double
 area = 0; if(sd + sd1 < sd2) { if(r < d1) area = r * r
* (acos(h / d2) - acos(h / d1)) / 2; else { area = r * r
 * ( acos(h / d2) - acos(h / r)) / 2; double y = sqrtl(r
 * r - h * h); area += h * (y - x) / 2; } } else { if(r
< h) area = r * r * (acos(h / d2) + acos(h / d1)) / 2;
else { area += r * r * (acos(h / d2) - acos(h / r)) / 2;
 double y = sqrtl(r * r - h * h); area += h * y / 2; if(
r < d1) { area += r * r * (acos(h / d1) - acos(h / r)) /
 2; area += h * y / 2; } else area += h * x / 2; } }
return area; } /* intersection between a simple polygon
and a circle */ double polygon_circle_intersection(
vector<PT> &v, PT p, double r) { int n = v.size();
double ans = 0.00; PT org = {0, 0}; for(int i = 0; i < n
; i++) { int x = orientation(p, v[i], v[(i + 1) % n]);
if(x == 0) continue; double area =
_triangle_circle_intersection(org, r, v[i] - p, v[(i +
1) % n] - p); if (x < 0) ans -= area; else ans += area;
} return abs(ans); } /* find a circle of radius r that
contains as many points as possible */ /* O(n^2 log n);
*/ double maximum_circle_cover(vector<PT> p, double r,
circle &c) { int n = p.size(); int ans = 0; int id = 0;
double th = 0; for (int i = 0; i < n; ++i) { /* maximum
circle cover when the circle goes through this point */
vector<pair<double, int>> events = {{-PI, +1}, {PI,
-1}}; for (int j = 0; j < n; ++j) { if (j == i) continue
; double d = dist(p[i], p[j]); if (d > r * 2) continue;
double dir = (p[j] - p[i]).arg(); double ang = acos(d /
2 / r); double st = dir - ang, ed = dir + ang; if (st >
PI) st -= PI * 2; if (st <= -PI) st += PI * 2; if (ed >
PI) ed -= PI * 2; if (ed <= -PI) ed += PI * 2; events.
push_back({st - eps, +1}); /* take care of precisions!
*/ events.push_back({ed, -1}); if (st > ed) { events.
push_back({-PI, +1}); events.push_back({+PI, -1}); } }
sort(events.begin(), events.end()); int cnt = 0; for (
auto &&e: events) { cnt += e.second; if (cnt > ans) {
ans = cnt; id = i; th = e.first; } } } PT w = PT(p[id].x
 + r * cos(th), p[id].y + r * sin(th)); c = circle(w, r)
; /* best_circle */ return ans; } /* radius of the
maximum inscribed circle in a convex polygon */ double
maximum_inscribed_circle(vector<PT> p) { int n = p.size
(); if (n <= 2) return 0; double l = 0, r = 20000; while
 (r - l > eps) { double mid = (l + r) * 0.5; vector<HP>
h; const int L = 1e9; h.push_back(HP(PT(-L, -L), PT(L, -
```

```
L))); h.push_back(HP(PT(L, -L), PT(L, L))); h.push_back(
HP(PT(L, L), PT(-L, L))); h.push_back(HP(PT(-L, L), PT(-
L, -L))); for (int i = 0; i < n; i++) { PT z = (p[(i +
1) % n] - p[i]).perp(); z = z.truncate(mid); PT y = p[i]
 + z, q = p[(i + 1) % n] + z; h.push_back(HP(p[i] + z, p
[(i + 1) % n] + z)); } vector<PT> nw =
half_plane_intersection(h); if (!nw.empty()) l = mid;
else r = mid; } return l; } /* ear decomposition, O(n^3)
 but faster */ vector<vector<PT>> triangulate(vector<PT>
 p) { vector<vector<PT>> v; while (p.size() >= 3) { for
(int i = 0, n = p.size(); i < n; i++) { int pre = i == 0
 ? n - 1 : i - 1;; int nxt = i == n - 1 ? 0 : i + 1;;
int ori = orientation(p[i], p[pre], p[nxt]); if (ori <
0) { int ok = 1; for (int j = 0; j < n; j++) { if (j ==
i || j == pre || j == nxt)continue; if (
is_point_in_triangle(p[i], p[pre], p[nxt] , p[j]) < 1) {
 ok = 0; break; } } if (ok) { v.push_back({p[pre], p[i],
 p[nxt]}); p.erase(p.begin() + i); break; } } } } return
 v; }

struct star { int n; /* number of sides of the star */
double r; /* radius of the circumcircle */ star(int _n,
double _r) { n = _n; r = _r; } double area() { double
theta = PI / n; double s = 2 * r * sin(theta); double R
= 0.5 * s / tan(theta); double a = 0.5 * n * s * R;
double a2 = 0.25 * s * s / tan(1.5 * theta); return a -
n * a2; } }; /* given a list of lengths of the sides of
a polygon in counterclockwise order */ /* returns the
maximum area of a non-degenerate polygon that can be
formed using those lengths */ double
get_maximum_polygon_area_for_given_lengths(vector<double
> v) { if (v.size() < 3) { return 0; } int m = 0; double
 sum = 0; for (int i = 0; i < v.size(); i++) { if (v[i]
> v[m]) { m = i; } sum += v[i]; } if (sign(v[m] - (sum -
 v[m])) >= 0) { return 0; /* no non-degenerate polygon
is possible */ } /* the polygon should be a circular
polygon */ /* that is all points are on the
circumference of a circle */ double l = v[m] / 2, r = 1
e6; /* fix it correctly */ int it = 60; auto ang = [](
double x, double r) { /* x = length of the chord, r =
radius of the circle */ return 2 * asin((x / 2) / r); };
 auto calc = [=](double r) { double sum = 0; for (auto x
: v) { sum += ang(x, r); } return sum; }; /* compute the
 radius of the circle */ while (it--) { double mid = (l
+ r) / 2; if (calc(mid) <= 2 * PI) { r = mid; } else { l
 = mid; } } if (calc(r) <= 2 * PI - eps) { /* the center
 of the circle is outside the polygon */ auto calc2 =
[&](double r) { double sum = 0; for (int i = 0; i < v.
size(); i++) { double x = v[i]; double th = ang(x, r);
if (i != m) { sum += th; } else { sum += 2 * PI - th; }
} return sum; }; l = v[m] / 2; r = 1e6; it = 60; while (
it--) { double mid = (l + r) / 2; if (calc2(mid) > 2 *
PI) { r = mid; } else { l = mid; } } auto get_area =
[=](double r) { double ans = 0; for (int i = 0; i < v.
size(); i++) { double x = v[i]; double area = r * r *
sin(ang(x, r)) / 2; if (i != m) { ans += area; } else {
ans -= area; } } return ans; }; return get_area(r); }
else { /* the center of the circle is inside the polygon
 */ auto get_area = [=](double r) { double ans = 0; for
(auto x: v) { ans += r * r * sin(ang(x, r)) / 2; }
return ans; }; return get_area(r); } }
```

7.2 3D Shohag

```
const double inf = 1e100; const double eps = 1e-9; const
 double PI = acos((double)-1.0); int sign(double x) {
return (x > eps) - (x < -eps); } struct PT { double x, y
; PT() { x = 0, y = 0; } PT(double x, double y) : x(x),
y(y) {} PT(const PT &p) : x(p.x), y(p.y) {} void scan()
{ cin >> x >> y; } PT operator + (const PT &a) const {
return PT(x + a.x, y + a.y); } PT operator - (const PT &
a) const { return PT(x - a.x, y - a.y); } PT operator *
(const double a) const { return PT(x * a, y * a); }
friend PT operator * (const double &a, const PT &b) {
return PT(a * b.x, a * b.y); } PT operator / (const
double a) const { return PT(x / a, y / a); } bool
operator == (PT a) const { return sign(a.x - x) == 0 &&
sign(a.y - y) == 0; } bool operator != (PT a) const {
return !(*this == a); } bool operator < (PT a) const {
return sign(a.x - x) == 0 ? y < a.y : x < a.x; } bool
operator > (PT a) const { return sign(a.x - x) == 0 ? y
> a.y : x > a.x; } double norm() { return sqrt(x * x + y
 * y); } double norm2() { return x * x + y * y; } PT
perp() { return PT(-y, x); } double arg() { return atan2
(y, x); } PT truncate(double r) { /* // returns a vector
 with norm r and having same direction */ double k =
norm(); if (!sign(k)) return *this; r /= k; return PT(x
* r, y * r); } }; inline double dot(PT a, PT b) { return
 a.x * b.x + a.y * b.y; } inline double dist2(PT a, PT b
) { return dot(a - b, a - b); } inline double dist(PT a,
 PT b) { return sqrt(dot(a - b, a - b)); } inline double
 cross(PT a, PT b) { return a.x * b.y - a.y * b.x; }
inline int orientation(PT a, PT b, PT c) { return sign(
cross(b - a, c - a)); } PT perp(PT a) { return PT(-a.y,
a.x); } PT rotateccw90(PT a) { return PT(-a.y, a.x); }
PT rotatecw90(PT a) { return PT(a.y, -a.x); } PT
rotateccw(PT a, double t) { return PT(a.x * cos(t) - a.y
 * sin(t), a.x * sin(t) + a.y * cos(t)); } PT rotatecw(
PT a, double t) { return PT(a.x * cos(t) + a.y * sin(t),
 -a.x * sin(t) + a.y * cos(t)); } double SQ(double x) {
return x * x; } double rad_to_deg(double r) { return (r
* 180.0 / PI); } double deg_to_rad(double d) { return (d
 * PI / 180.0); } double get_angle(PT a, PT b) { double
costheta = dot(a, b) / a.norm() / b.norm(); return acos(
max((double)-1.0, min((double)1.0, costheta))); } struct
 p3 { double x, y, z; p3() { x = 0, y = 0; z = 0; } p3(
double x, double y, double z) : x(x), y(y), z(z) {} p3(
const p3 &p) : x(p.x), y(p.y), z(p.z) {} void scan() {
cin >> x >> y >> z; } p3 operator + (const p3 &a) const
{ return p3(x + a.x, y + a.y, z + a.z); } p3 operator -
(const p3 &a) const { return p3(x - a.x, y - a.y, z - a.
z); } p3 operator * (const double a) const { return p3(x
 * a, y * a, z * a); } friend p3 operator * (const
double &a, const p3 &b) { return p3(a * b.x, a * b.y, a
* b.z); } p3 operator / (const double a) const { return
p3(x / a, y / a, z / a); } bool operator == (p3 a) const
 { return sign(a.x - x) == 0 && sign(a.y - y) == 0 &&
sign(a.z - z) == 0; } bool operator != (p3 a) const {
return !(*this == a); } double abs() { return sqrt(x * x
 + y * y + z * z); } double sq() { return x * x + y * y
+ z * z; } p3 unit() { return *this / abs(); } }zero(0,
0, 0); double operator | (p3 v, p3 w) { /* //dot product
 */ return v.x * w.x + v.y * w.y + v.z * w.z; } p3
operator * (p3 v, p3 w) { /* //cross product */ return {
v.y * w.z - v.z * w.y, v.z * w.x - v.x * w.z, v.x * w.y
- v.y * w.x}; } double sq(p3 v) { return v | v; } double
 abs(p3 v) { return sqrt(sq(v)); } p3 unit(p3 v) {
return v / abs(v); } inline double dot(p3 a, p3 b) {
return a.x * b.x + a.y * b.y + a.z * b.z; } inline
double dist2(p3 a, p3 b) { return dot(a - b, a - b); }
inline double dist(p3 a, p3 b) { return sqrt(dot(a - b,
a - b)); } inline p3 cross(p3 a, p3 b) { return p3(a.y *
 b.z - a.z * b.y, a.z * b.x - a.x * b.z, a.x * b.y - a.y
 * b.x); } /* // if s is on the same side of the plane
pqr as the vector pq * pr then it will be positive //
otherwise negative or 0 if on the plane */ double orient
(p3 p, p3 q, p3 r, p3 s) { return (q - p) * (r - p) | (s
 - p); } /* // returns orientation of p to q to r on the
 plane perpendicular to n // assuming p, q, r are on the
```

```
 plane */ double orient_by_normal(p3 p, p3 q, p3 r, p3 n
) { return (q - p) * (r - p) | n; } double get_angle(p3
a, p3 b) { double costheta = dot(a, b) / a.abs() / b.abs
(); return acos(max((double)-1.0, min((double)1.0,
costheta))); } double small_angle(p3 v, p3 w) { return
acos(min(fabs(v | w) / abs(v) / abs(w), (double)1.0)); }
 struct plane { /* // n is the perpendicular normal
vector to the plane */ p3 n; double d; /* // (n | p) = d
 */ /* // From normal n and offset d */ plane(p3 n,
double d) : n(n), d(d) {} /* // From normal n and point
P */ plane(p3 n, p3 p) : n(n), d(n | p) {} /* // From
three non-collinear points P,Q,R */ plane(p3 p, p3 q, p3
 r) : plane((q - p) * (r - p), p) {} /* // positive if
on the same side as the normal, negative if on the
opposite side, 0 if on the plane */ double side(p3 p) {
return (n | p) - d; } /* // distance from point p to
plane */ double dist(p3 p) { return fabs(side(p)) / abs(
n); } /* // translate the plane by vector t */ plane
translate(p3 t) { return {n, d + (n | t)}; } /* // shift
 the plane perpendicular to n by distance dist */ plane
shiftUp(double dist) { return {n, d + dist * abs(n)}; }
/* // orthogonal projection of point p onto plane */ p3
proj(p3 p) { return p - n * side(p) / sq(n); } /* //
orthogonal reflection of point p onto plane */ p3 refl(
p3 p) { return p - n * 2 * side(p) / sq(n); } pair<p3,
p3> get_two_points_on_plane() { assert(sign(n.x) != 0 ||
 sign(n.y) != 0 || sign(n.z) != 0); if (sign(n.x) == 0
&& sign(n.y) == 0) return {p3(1, 0, d/n.z), p3(0, 1, d/n
.z)}; if (sign(n.y) == 0 && sign(n.z) == 0) return {p3(d
/n.x, 1, 0), p3(d/n.x, 0, 1)}; if (sign(n.z) == 0 &&
sign(n.x) == 0) return {p3(1, d/n.y, 0), p3(0, d/n.y, 1)
}; if (sign(n.x) == 0) return {p3(1, d/n.y, 0), p3(0, 0,
 d/n.z)}; if (sign(n.y) == 0) return {p3(0, 1, d/n.z),
p3(d/n.x, 0, 0)}; if (sign(n.z) == 0) return {p3(d/n.x,
0, 1), p3(0, d/n.y, 0)}; if (sign(d)!=0) return {p3(d/n.
x, 0, 0), p3(0, d/n.y, 0)}; return {p3(n.y, -n.x, 0), p3
(-n.y, n.x, 0)}; } }; struct coords { /* // coordinate
system for coplanar points // o is the origin, dx, dy,
dz are unit vectors similar to normal 3D system // but
dx and dy are on the plane */ p3 o, dx, dy, dz; /* //
From three points P, Q, R on the plane */ coords(p3 p,
p3 q, p3 r) : o(p) { dx = unit(q - p); dz = unit(dx * (r
 - p)); dy = dz * dx; } /* // From four points P,Q,R,S:
take directions PQ, PR, PS as is // it allows us to keep
 using integer coordinates but has some pitfalls // e.g.
 distances and angles are not preserved but relative
positions are (convex hull works) */ coords(p3 p, p3 q,
p3 r, p3 s) : o(p), dx(q - p), dy(r - p), dz(s - p) {}
/* // 2D position vector of point p in this coordinate
system centered at o // p must be on the plane */ PT
pos2d(p3 p) { return {(p - o) | dx, (p - o) | dy}; } /*
// returns the 3D position vector of point p in this new
 coordinate system // p can be outside the plane */ p3
pos3d(p3 p) { return {(p - o) | dx, (p - o) | dy, (p - o
) | dz}; } /* // given 2D position vector p centered at
o, return the original 3D position vector */ p3 pos3d(PT
 p){ return o + dx * p.x + dy * p.y; } }; struct line3d
{ /* // d is the direction vector of the line */ p3 d, o
; /* // p = o + k * d (k is a real parameter) */ line3d
() {} /* // From two points P, Q */ line3d(p3 p, p3 q) :
 d(q - p), o(p) {} /* // From two planes p1, p2 //
assuming they are not parallel */ line3d(plane p1, plane
 p2) { d = p1.n * p2.n; o = (p2.n * p1.d - p1.n * p2.d)
* d / sq(d); /* // o is actually the closest point on
the line to the origin */ } double dist2(p3 p) { return
sq(d * (p - o)) / sq(d); } double dist(p3 p) { return
sqrt(dist2(p)); } /* // compare points by their
projection on the line // so you can sort points on the
line using this */ bool cmp_proj(p3 p, p3 q) { return (d
 | p) < (d | q); } /* // orthogonal projection of point
p onto line */ p3 proj(p3 p) { return o + d * (d|(p - o)
) / sq(d); } /* // orthogonal reflection of point p onto
 line */ p3 refl(p3 p) { return proj(p) * 2 - p; } /* //
 returns the intersection point of the line with plane p
 // assuming plane and line are not parallel */ p3 inter
(plane p) { /* // assert((d | p.n) != 0); // no
intersection if parallel */ return o - d * p.side(o) / (
d | p.n); } }; /* // smallest distance between two lines
 */ double dist(line3d l1, line3d l2) { p3 n = l1.d * l2
.d; if (n == zero) return l1.dist(l2.o); /* // parallel
*/ return fabs((l2.o - l1.o) | n) / abs(n); } /* //
closest point from line l2 to line l1 */ p3
closest_on_l1(line3d l1, line3d l2) { p3 n2 = l2.d * (l1
.d * l2.d); return l1.o + l1.d * ((l2.o - l1.o) | n2) /
(l1.d | n2); } /* // small angle between direction
vectors of two lines */ double get_angle(line3d l1,
line3d l2) { return small_angle(l1.d, l2.d); } bool
is_parallel(line3d l1, line3d l2) { return l1.d * l2.d
== zero; } bool is_perpendicular(line3d l1, line3d l2) {
 return sign((l1.d | l2.d)) == 0; } /* // small angle
between normal vectors of two planes */ double get_angle
(plane p1, plane p2) { return small_angle(p1.n, p2.n); }
 bool is_parallel(plane p1, plane p2) { return p1.n * p2
.n == zero; } bool is_perpendicular(plane p1, plane p2)
{ return sign((p1.n | p2.n)) == 0; } double get_angle(
plane p, line3d l) { return PI / 2 - small_angle(p.n, l.
d); } bool is_parallel(plane p, line3d l) { return sign
((p.n | l.d)) == 0; } bool is_perpendicular(plane p,
line3d l) { return p.n * l.d == zero; } /* // returns
the line perpendicular to plane p and passing through
point o */ line3d perp_through(plane p, p3 o) {return
line3d(o, o + p.n);} /* // returns the plane
perpendicular to line l and passing through point o */
plane perp_through(line3d l, p3 o) {return plane(l.d, o)
;} /* // returns two points on intesection line of two
planes formed by points // a1, b1, c1 and a2, b2, c2
respectively */ pair<p3, p3> plane_plane_intersection(p3
 a1, p3 b1, p3 c1, p3 a2, p3 b2, p3 c2) { p3 n1 = (b1 -
a1) * (c1 - a1); p3 n2 = (b2 - a2) * (c2 - a2); double
d1 = n1 | a1, d2 = n2 | a2; p3 d = n1 * n2; if (d ==
zero) return make_pair(zero, zero); p3 o = (n2 * d1 - n1
 * d2) * d / (d | d); return make_pair(o, o + d); } /*
// returns center of circle passing through three // non
-colinear and co-planer points a, b and c */ p3
circle_center(p3 a, p3 b, p3 c) { p3 v1 = b - a, v2 = c
- a; double v1v1 = v1 | v1, v2v2 = v2 | v2, v1v2 = v1 |
v2; double base = 0.5 / (v1v1 * v2v2 - v1v2 * v1v2);
double k1 = base * v2v2 * (v1v1 - v1v2); double k2 =
base * v1v1 * (v2v2 - v1v2); return a + v1 * k1 + v2 *
k2; } /* // segment ab to point c */ double
distance_from_segment_to_point(p3 a, p3 b, p3 c) { if (
sign(dot(b - a, c - a)) < 0) return dist(a, c); if (sign
(dot(a - b, c - b)) < 0) return dist(b, c); return fabs(
cross((b - a).unit(), c - a).abs()); } double
distance_from_triangle_to_point(p3 a, p3 b, p3 c, p3 d)
{ plane P(a, b, c); p3 proj = P.proj(d); double dis =
min(distance_from_segment_to_point(a, b, d), min(
distance_from_segment_to_point(b, c, d),
distance_from_segment_to_point(c, a, d))); int o = sign(
orient_by_normal(a, b, proj, P.n)); int inside = o ==
sign(orient_by_normal(b, c, proj, P.n)); inside &= o ==
sign(orient_by_normal(c, a, proj, P.n)); if (inside)
return (d - proj).abs(); return dis; } double
distance_from_triangle_to_segment(p3 a, p3 b, p3 c, p3 d
, p3 e) { double l = 0.0, r = 1.0; int cnt = 100; double
 ret = inf; while (cnt--) { double mid1 = l + (r - l) /
3.0, mid2 = r - (r - l) / 3.0; double x =
distance_from_triangle_to_point(a, b, c, d + (e - d) *
```

```
mid1); double y = distance_from_triangle_to_point(a, b,
c, d + (e - d) * mid2); if (x < y) { r = mid2; ret = x;
} else { ret = y; l = mid1; } } return ret; } /* //
triangles are solid */ double
distance_from_triangle_to_triangle(p3 a, p3 b, p3 c, p3
d, p3 e, p3 f) { double ret = inf; ret = min(ret,
distance_from_triangle_to_segment(a, b, c, d, e)); ret =
 min(ret, distance_from_triangle_to_segment(a, b, c, e,
f)); ret = min(ret, distance_from_triangle_to_segment(a,
 b, c, f, d)); ret = min(ret,
distance_from_triangle_to_segment(d, e, f, a, b)); ret =
 min(ret, distance_from_triangle_to_segment(d, e, f, b,
c)); ret = min(ret, distance_from_triangle_to_segment(d,
 e, f, c, a)); return ret; } bool operator < (p3 p, p3 q
) { return tie(p.x, p.y, p.z) < tie(q.x, q.y, q.z); }
struct edge { int v; bool same; /* // is the common edge
 between two faces in the same order? */ }; /* // Given
a series of faces (lists of points) of a polyhedron,
reverse some of them // so that their orientations are
consistent (all area vectors of the faces either
pointing outwards or inwards) // just compute the area
vector of one face to see if its pointing outwards or
inwards */ vector<vector<p3>> reorient(vector<vector<p3
>> fs) { int n = fs.size(); /* // Find the common edges
and create the resulting graph */ vector<vector<edge>> g
(n); map<pair<p3,p3>, int> es; for (int u = 0; u < n; u
++) { for (int i = 0, m = fs[u].size(); i < m; i++) { p3
 a = fs[u][i], b = fs[u][(i + 1) % m]; /* // Lets look
at edge a-b */ if (es.count({a, b})) { /* // seen in
same order */ int v = es[{a, b}]; g[u].push_back({v,
true}); g[v].push_back({u, true}); } else if (es.count({
b, a})) { /* // seen in different order */ int v = es[ {
b, a}]; g[u].push_back({v,false}); g[v].push_back({u,
false}); } else { /* // not seen yet */ es[{a,b}] = u; }
 } } /* // Perform BFS to find which faces should be
flipped */ vector<bool> vis(n,false), flip(n); flip[0] =
 false; queue<int> q; q.push(0); while (!q.empty()) {
int u = q.front(); q.pop(); for (edge e : g[u]) { if (!
vis[e.v]) { vis[e.v] = true; /* // If the edge was in
the same order, // exactly one of the two should be
flipped */ flip[e.v] = (flip[u] ^ e.same); q.push(e.v);
} } } /* Actually perform the flips */ for (int u = 0; u
 < n; u++) if (flip[u]) { reverse(fs[u].begin(), fs[u].
end()); } return fs; } /* // O(n^2), O(n) faces in the
hull */ struct CH3D { struct face { int a, b, c;/* //
the number of three points on one face of the convex
hull */ bool ok; /* // whether the face belongs to the
face on the final convex hull */ }; int n; /* // initial
 vertex number */ vector<p3> P; int num; /* // convex
hull surface triangle number */ vector<face> F; /* //
convex surface triangles */ vector<vector<int>> g; void
init(vector<p3> p) { P = p; n = p.size(); F.resize(8 * n
 + 1); g.resize(n + 1, vector<int> (n + 1)); } double
len(p3 a) { return sqrt(a | a); } p3 cross(const p3 &a,
const p3 &b, const p3 &c) { return (b - a) * (c - a); }
double area(p3 a, p3 b, p3 c) { return len((b - a) * (c
- a)); } double volume(p3 a, p3 b, p3 c, p3 d) { return
(b - a) * (c - a) | (d - a); } /* // positive: p3 in the
 same direction */ double dblcmp(p3 &p, face &f) { p3 m
= P[f.b] - P[f.a]; p3 n = P[f.c] - P[f.a]; p3 t = p - P[
f.a]; return (m * n) | t; } void deal(int p, int a, int
b) { int f = g[a][b]; /* // search for another plane
adjacent to the edge */ face add; if (F[f].ok) { if (
dblcmp(P[p], F[f]) > eps) dfs(p, f); else { add.a = b;
add.b = a; add.c = p; /* // pay attention to the order
here, to be right-handed */ add.ok = true; g[p][b] = g[a
][p] = g[b][a] = num; F[num++] = add; } } } /* //
recursively search all faces that should be removed from
 the convex hull */ void dfs(int p, int now) { F[now].ok
 = 0; deal(p, F[now].b, F[now].a); deal(p, F[now].c, F[
now].b); deal(p, F[now].a, F[now].c); } bool same(int s,
 int t) { p3 &a = P[F[s].a]; p3 &b = P[F[s].b]; p3 &c =
P[F[s].c]; return fabs(volume(a, b, c, P[F[t].a])) < eps
 && fabs(volume(a, b, c, P[F[t].b])) < eps && fabs(
volume(a, b, c, P[F[t].c])) < eps; } /* // building a 3D
 convex hull */ void create_hull() { int i, j, tmp; face
 add; num = 0; if (n < 4)return; /* // ensure that the
first four points are not coplanar */ bool flag = true;
for (i = 1; i < n; i++) { if (len(P[0] - P[i]) > eps) {
swap(P[1], P[i]); flag = false; break; } } if (flag)
return; flag = true; /* // make the first three points
not collinear */ for (i = 2; i < n; i++) { if (len((P[0]
 - P[1]) * (P[1] - P[i])) > eps) { swap(P[2], P[i]);
flag = false; break; } } if (flag) return; flag = true;
/* // make the first four points not coplanar */ for (
int i = 3; i < n; i++) { if (fabs((P[0] - P[1]) * (P[1]
- P[2]) | (P[0] - P[i])) > eps) { swap(P[3], P[i]); flag
 = false; break; } } if (flag) return; for (i = 0; i <
4; i++) { add.a = (i + 1) % 4; add.b = (i + 2) % 4; add.
c = (i + 3) % 4; add.ok = true; if (dblcmp(P[i], add) >
0)swap(add.b, add.c); g[add.a][add.b] = g[add.b][add.c]
= g[add.c][add.a] = num; F[num++] = add; } for (i = 4; i
 < n; i++) { for (j = 0; j < num; j++) { if (F[j].ok &&
dblcmp(P[i], F[j]) > eps) { dfs(i, j); break; } } } tmp
= num; for (i = num = 0; i < tmp; i++) if (F[i].ok) F[
num++] = F[i]; } double surface_area() { double res = 0;
 if (n == 3) { p3 p = cross(P[0], P[1], P[2]); res = len
(p) / 2.0; return res; } for(int i = 0; i < num; i++) {
res += area(P[F[i].a], P[F[i].b], P[F[i].c]); } return
res / 2.0; } double volume() { double res = 0; p3 tmp(0,
 0, 0); for(int i = 0; i < num; i++) { res += volume(tmp
, P[F[i].a], P[F[i].b], P[F[i].c]); } return fabs(res /
6.0); } int number_of_triangles() { /* // number of
surface triangles */ return num; } int
number_of_polygons() { /* // number of surface polygons
*/ int i, j, res, flag; for (i = res = 0; i < num; i++)
{ flag = 1; for (j = 0; j < i; j++) { if (same(i, j)) {
flag = 0; break; } } res += flag; } return res; } p3
centroid() { /* // center of gravity */ p3 ans(0, 0, 0),
 o(0, 0, 0); double all = 0; for (int i = 0; i < num; i
++) { double vol = volume(o, P[F[i].a], P[F[i].b], P[F[i
].c]); ans = ans + (o + P[F[i].a] + P[F[i].b] + P[F[i].c
]) / 4.0 * vol; all += vol; } ans = ans / all; return
ans; } double point_to_face_distance(p3 p, int i) {
return fabs(volume(P[F[i].a], P[F[i].b], P[F[i].c], p) /
 len((P[F[i].b] - P[F[i].a]) * (P[F[i].c] - P[F[i].a])))
; } }; /* // given the radius of the sphere, latitude
and longitude of a point in degrees // return the 3D
coordinates of the point on the sphere assuming the
sphere is centered at the origin */ p3 get_sphere(double
 r, double lat, double lon) { lat *= PI / 180, lon *= PI
 / 180; return {r * cos(lat) * cos(lon), r * cos(lat) *
sin(lon), r * sin(lat)}; } int sphere_line_intersection(
p3 o, double r, line3d l, pair<p3,p3> &out) { double h2
= r * r - l.dist2(o); if (h2 < 0) return 0; /* // the
line doesnt touch the sphere */ p3 p = l.proj(o); p3 h =
 l.d * sqrt(h2)/abs(l.d); /* // vector parallel to l, of
 length h */ out = {p - h, p + h}; return 1 + (h2 > 0);
} /* // The shortest distance between two points A and B
 on a sphere (O, r) is // given by travelling along
plane OAB and on the surface of the sphere. It is called
 the great-circle distance // if a and b are outside the
 sphere, then it will give the distance between their
projections on the sphere */ double great_circle_dist(p3
 o, double r, p3 a, p3 b) { /* // s = r * theta */
return r * get_angle(a - o, b - o); } /* // Assume that
the sphere is centered at the origin // We will call a
segment [AB] valid if A and B are not // opposite each
```

```
other on the sphere */ bool validSegment(p3 a, p3 b) {
return a * b != zero || (a | b) > 0; } bool
proper_intersection(p3 a, p3 b, p3 c, p3 d, p3 &out) {
p3 ab = a * b, cd = c * d; /* // normals of planes OAB
and OCD */ int oa = sign(cd | a), ob = sign(cd | b), oc
= sign(ab | c), od = sign(ab | d); out = ab * cd * od;
/* // four multiplications => careful with overflow! */
return (oa != ob && oc != od && oa != oc); } /* //
Assume that the sphere is centered at the origin */ bool
 point_on_sphere_segment(p3 a, p3 b, p3 p) { p3 n = a*b;
 if (n == zero) return a * p == zero && (a | p) > 0;
return (n | p) == 0 && (n | a * p) >= 0 && (n | b * p)
<= 0; } struct DirectionSet : vector<p3> { using vector
::vector;/* // import constructors */ void insert(p3 p)
{ for (p3 q : *this) if (p*q == zero) return; push_back(
p); } }; /* // Assume that the sphere is centered at the
 origin // it returns the direction vectors of the
intersection points // to get the actual points, scale
the direction vectors to the radius of the sphere */
DirectionSet segment_segment_intersection_on_sphere(p3 a
, p3 b, p3 c, p3 d) { assert(validSegment(a, b) &&
validSegment(c, d)); p3 out; if (proper_intersection(a,
b, c, d, out)) return {out}; DirectionSet s; if (
point_on_sphere_segment(c, d, a)) s.insert(a); if (
point_on_sphere_segment(c, d, b)) s.insert(b); if (
point_on_sphere_segment(a, b, c)) s.insert(c); if (
point_on_sphere_segment(a, b, d)) s.insert(d); return s;
 } /* // small angle between spherical segments ab and
ac // assume that the sphere is centered at the origin
// all points a, b, c are on the sphere */ double
angle_on_sphere(p3 a, p3 b, p3 c) { return get_angle(a *
 b, a * c); } /* // oriented angle between spherical
segments ab and ac // that is how much we rotate
counterclockwise to get from ab to ac // assume that the
 sphere is centered at the origin // all points a, b, c
are on the sphere */ double oriented_angle_on_sphere(p3
a, p3 b, p3 c) { if ((a * b | c) >= 0) return
angle_on_sphere(a, b, c); else return 2 * PI -
angle_on_sphere(a, b, c); } /* // Assume that the sphere
 is centered at the origin // the polygon is simple and
given in counterclockwise order // for each consecutive
pair of points, the counterclockwise left // part of the
 segment is considered to be inside the surface area
that the polygon encloses // if the polygon is outside
the sphere, the projection of the polygon on the sphere
will be considered */ double
area_on_the_surface_of_the_sphere(double r, vector<p3> p
) { int n = p.size(); double sum = -(n - 2) * PI; for (
int i = 0; i < n; i++) { sum += oriented_angle_on_sphere
(p[(i + 1) % n], p[(i + 2) % n], p[i]); } return r * r *
 sum; } /* // Assume that O is the origin // it returns
0 if O is outside the polyhedron // 1 if O is inside the
 polyhedron, and the vector areas of the faces are
oriented towards the outside; // 1 if O is inside the
polyhedron, and the vector areas of the faces are
oriented towards the inside. */ int winding_number_3D(
vector<vector<p3>> fs) { double sum = 0; for (vector<p3>
 f : fs) { sum += remainder(
area_on_the_surface_of_the_sphere(1, f), 4 * PI); }
return round(sum / (4 * PI)); } struct sphere { p3 c;
double r; sphere() {} sphere(p3 c, double r) : c(c), r(r
) {} }; /* // spherical cap is a portion of a sphere cut
 off by a plane */ struct spherical_cap { p3 c; double r
; spherical_cap() {} spherical_cap(p3 c, double r) : c(c
), r(r) {} /* // angle th is the polar angle between the
 rays from the center of the sphere to one edge of the
cap // and orthogonal line from the center of the sphere
 to the plane of the cap // height of the cap (just like
 real world cap) */ double height(double th) { return r
* (1 - cos(th)); } /* // radius of the base of the cap
*/ double base_radius(double th) { return r * sin(th); }
 /* // volume of the cap */ double volume(double th) {
double h = height(th); return PI * h * h * (3 * r - h) /
 3.0; } /* // surface area of the cap */ double
surface_area(double th) { double h = height(th); return
2 * PI * r * h; } }; /* // returns the sphere passing
through four points */ sphere circumscribed_sphere(p3 a,
 p3 b, p3 c, p3 d) { assert( sign(plane(a, b, c).side(d)
) != 0); plane u = plane(a - b, (a + b) / 2); plane v =
plane(b - c, (b + c) / 2); plane w = plane(c - d, (c + d
) / 2); assert(!is_parallel(u, v)); assert(!is_parallel(
v, w)); line3d l1(u, v), l2(v, w); assert( sign(dist(l1,
 l2)) == 0); p3 C = closest_on_l1(l1, l2); return sphere
(C, abs(C - a)); } /* // it won't work if one sphere is
totally inside the other sphere // handle that case
separately // returns the surface area and volume of the
 intersection */ pair<double, double>
sphere_sphere_intersection(sphere s1, sphere s2) {
double d = abs(s1.c - s2.c); if(sign(d - s1.r - s2.r) >=
 0) return {0, 0}; /* // not intersecting */ /* // only
the distance matters, so we will now consider the
centers // of the big sphere to be (0, 0, 0) and (d, 0,
0) for the small sphere // we can transform the results
back to w.r.t the real centers if we want */ double R =
max(s1.r, s2.r); double r = min(s1.r, s2.r); double y =
R + r - d; double x = (R * R - r * r + d * d) / (2 * d);
 /* // the intersecting plane is parallel to the yz
plane // with the above x value as its x coordinate */
double w = d * d - r * r + R * R; double a = sqrt(4 * d
* d * R * R - w * w) / (2.0 * d); /* // a is the radius
of the intersecting circle on the intersecting plane //
with center (x, 0) */ double h1 = R - x; double h2 = y -
 h1; /* // h1 is the height of the intersecting
spherical cap of the big sphere // h2 is for the small
sphere // total volume of the whole intersection = sum
of the volumes of the spherical caps */ double volume =
PI * h1 * h1 * (3 * R - h1) / 3.0 + PI * h2 * h2 * (3 *
r - h2) / 3.0; /* // total surface area of the
intersecting spherical caps */ double surface_area = 2 *
 PI * R * h1 + 2 * PI * r * h2; return make_pair(
surface_area, volume); } sphere
smallest_enclosing_sphere(vector<p3> p) { int n = p.size
(); p3 c(0, 0, 0); for(int i = 0; i < n; i++) c = c + p[
i]; c = c / n; double ratio = 0.1; int pos = 0; int it =
 100000; while (it--) { pos = 0; for (int i = 1; i < n;
i++) { if(sq(c - p[i]) > sq(c - p[pos])) pos = i; } c =
c + (p[pos] - c) * ratio; ratio *= 0.998; } return
sphere(c, abs(c - p[pos])); } /* // it returns the angle
 of the spherical cap that is formed by the intersection
 of all tangents */ double tangent_from_point_to_sphere(
p3 p, sphere s) { double d = abs(p - s.c); if (sign(d -
s.r) < 0) return -1; /* // inside the sphere, so no
tangent */ if (sign(d - s.r) == 0) return -2; /* // on
the sphere, handle separately */ double tangent_length =
 sqrt(d * d - s.r * s.r); double th = acos(s.r / d);
return th; }
```

7.3 3D

```
template <typename DT> class Point { public: DT x, y, z;
 Point(){}; Point(DT x, DT y, DT z) : x(x), y(y), z(z)
{} template <typename X> Point(Point<X> p) : x(p.x), y(p
.y), z(p.z) {} Point operator + (const Point &rhs) const
 { return Point(x + rhs.x, y + rhs.y, z + rhs.z); }
Point operator - (const Point &rhs) const { return Point
(x - rhs.x, y - rhs.y, z - rhs.z); } Point operator * (
DT M) const { return Point(M * x, M * y, M * z); } Point
 operator / (DT M) const { return Point(x / M, y / M, z
/ M); } /* // cross product */ Point operator & (const
```

```
Point &rhs) const { return Point(y * rhs.z - z * rhs.y,z
 * rhs.x - x * rhs.z,x * rhs.y - y * rhs.x); } /* // dot
 product */ DT operator ^ (const Point &rhs) const {
return x * rhs.x + y * rhs.y + z * rhs.z; } bool
operator == (const Point &rhs) const { return x == rhs.x
 && y == rhs.y && z == rhs.z; } bool operator != (const
Point &rhs) const { return !(*this == rhs); } friend std
::istream& operator >> (std::istream &is, Point &p) {
return is >> p.x >> p.y >> p.z; } friend std::ostream&
operator << (std::ostream &os, const Point &p) { return
os << p.x << " " << p.y << " " << p.z; } friend DT DisSq
(const Point &a, const Point &b) { return (a.x - b.x)*(a
.x - b.x) + (a.y - b.y)*(a.y - b.y) + (a.z - b.z)*(a.z -
 b.z); } };

optional < Point <double> > ray_intersects_triangle(
const Point<double> &origin,const Point<double> &
ray_vector,const array <Point<double>, 3> &triangle) {
constexpr double epsilon = std::numeric_limits<double>::
epsilon(); auto [A, B, C] = triangle; Point<double>
edge1 = B - A; Point<double> edge2 = C - A; Point<double
> ray_cross_e2 = ray_vector & edge2; double det = edge1
^ ray_cross_e2; if (det > -epsilon && det < epsilon)
return {}; /* // Ray is parallel to this triangle. */
double inv_det = 1.0 / det; Point<double> s = ray_origin
 - A; double u = inv_det * (s ^ ray_cross_e2); if (u < 0
 || u > 1) return {}; Point<double> s_cross_e1 = s &
edge1; double v = inv_det * (ray_vector ^ s_cross_e1);
if (v < 0 || u + v > 1) return {}; /* // Compute t to
find the intersection Point */ double t = inv_det * (
edge2 ^ s_cross_e1); if (t > epsilon) return ray_origin
+ ray_vector * t; /* // ray intersection */ else return
{}; /* // Line intersection but not ray intersection */
} /* // HOW TO IMPLEMENT // auto tmp =
ray_intersects_triangle (origin, ray, v[i]); // if (tmp.
has_value ()) Point <double> intersection_point = tmp.
value (); */
```

7.4 3d hull

```
#define LD long double const LD EPS = 1e-9; const LD PI
= acos(-1); LD Sq(LD x) {return x * x;} LD Acos(LD x){
return acos(min(1.0L,max(-1.0L,x)));} LD Asin(LD x){
return asin(min(1.0L,max(-1.0L,x)));} LD Sqrt(LD x) {
return sqrt(max(0.0L, x));} int dcmp(LD x) { if(fabs(x)
< EPS) return 0; return (x > 0.0 ? +1 : -1); } struct
Point3 { LD x, y, z; Point3() {} Point3(LD a, LD b, LD c
) : x(a), y(b), z(c){} void operator=(const Point3& a) {
 x=a.x,y=a.y,z=a.z; } Point3 operator+(Point3 a) {
Point3 p{x + a.x, y + a.y, z + a.z}; return p; } Point3
operator-(Point3 a) { Point3 p{x - a.x, y - a.y, z - a.z
}; return p; } Point3 operator*(LD a) { return Point3(x*
a,y*a,z*a); } Point3 operator/(LD a) { assert(a > EPS);
Point3 p{x/a, y/a, z/a}; return p; } LD& operator[](int
a) { if (a == 0) return x; if (a == 1) return y; if (a
== 2) return z; assert(false); } bool IsZero() { return
abs(x)<EPS&& abs(y)<EPS && abs(z) < EPS; } bool operator
==(Point3 a) { return (*this - a).IsZero(); } LD dot(
Point3 a) { return x * a.x + y * a.y + z * a.z; } LD
Norm() { return Sqrt(x * x + y * y + z * z); } LD SqNorm
() { return x * x + y * y + z * z; } void NormalizeSelf
() { *this = *this/Norm(); } Point3 Normalize() { Point3
 res(*this); res.NormalizeSelf(); return res; } LD Dis(
Point3 a) { return (*this - a).Norm(); } pair<LD, LD>
SphericalAngles() { return {atan2(z,Sqrt(x*x+y*y)),atan2
(y,x)}; } LD Area(Point3 p) { return Norm() * p.Norm() *
 sin(Angle(p)) / 2; } /* LD Angle(Point3 p) { LD a =
Norm(), b = p.Norm(), c = Dis(p); return Acos((a*a+b*b-c
*c)/(2*a*b)); } */ LD Angle(Point3 b) { Point3 a(*this);
 return Acos(abs(a.dot(b))/a.Norm()/b.Norm()); } LD
Angle(Point3 p, Point3 q){return p.Angle(q);} Point3
cross(Point3 p) { Point3 q(*this); return {q[1]*p[2] - q
[2]*p[1], q[2]*p[0] - q[0] * p[2], q[0] * p[1] - q[1] *
p[0]}; } bool LexCmp(Point3& a, const Point3& b) { if (
abs(a.x - b.x) > EPS) { return a.x < b.x;} if (abs(a.y -
 b.y) > EPS) { return a.y < b.y;} return a.z < b.z; } };
 typedef vector<Point3> face; typedef vector<Point3>
edge; typedef vector<face> hull; #define INSIDE (-1) #
define ON (0) #define OUTSIDE (1) int side(Point3 a,
Point3 b, Point3 c, Point3 p){ Point3 norm = (b-a).cross
(c-a); Point3 me = p-a; return dcmp(me.dot(norm)); }
hull find_hull(vector<Point3> P) { random_shuffle(P.
begin(), P.end()); int n = P.size(); for(int j = 2; j <
n; j++) { Point3 n = (P[1]-P[0]).cross(P[j]-P[0]); if(n.
Norm() > EPS) {swap(P[j], P[2]);break;} } for(int j = 3;
 j < n; j++) { if(side(P[0],P[1],P[2],P[j])) { swap(P[j
], P[3]); break; } } if(side(P[0],P[1],P[2],P[3]) ==
OUTSIDE) swap(P[0], P[1]); hull H{ {P[0],P[1],P[2]}, {P
[0],P[3],P[1]}, {P[0],P[2],P[3]},{P[3],P[2],P[1]}}; auto
 make_degrees = [&](const hull& H) { map<edge,int> ans;
for(const auto & f : H) { for(int i = 0; i < 3; i++){
Point3 a = f[i], b = f[(i+1)%3]; ans[{a,b}]++; } }
return ans; }; for(int j = 4; j < n; j++) { hull H2; H2.
reserve((int)H.size()); vector<face> plane; for(const
auto & f : H) { int s = side(f[0],f[1],f[2],P[j]); if (s
 == INSIDE || s == ON) H2.push_back(f); } /* //For any
edge that now only has 1 incident //face (it's other
face deleted) add a new //face with this vertex and that
 edge. */ map<edge, int> D = make_degrees(H2); const
auto tmp = H2; for (const auto & f : tmp) { for(int i =
0; i < 3; i++) { Point3 a = f[i], b = f[(i+1)%3]; int d
= D[{a,b}] + D[{b,a}]; if (d==1) H2.push_back({a, P[j],
b}); } } H = H2; } return H; }
```

7.5 ClosestPairOfPoints

```
/* ///Gives squared Distance /// O(n log n) */ long long
 ClosestPair(vector<pair<int, int>> pts) { int n = pts.
size(); sort(pts.begin(), pts.end()); set<pair<int, int
>> s; long long best_dist = 1e18; int j = 0; for (int i
= 0; i < n; ++i) { int d = ceil(sqrt(best_dist)); while
(pts[i].first - pts[j].first >= best_dist) { s.erase({
pts[j].second, pts[j].first}); j += 1; } auto it1 = s.
lower_bound({pts[i].second - d, pts[i].first}); auto it2
 = s.upper_bound({pts[i].second + d, pts[i].first}); for
 (auto it = it1; it != it2; ++it) { int dx = pts[i].
first - it->second; int dy = pts[i].second - it->first;
best_dist = min(best_dist, 1LL * dx * dx + 1LL * dy * dy
); } s.insert({pts[i].second, pts[i].first}); } return
best_dist; }
```

7.6 MinDisSquares

```
typedef long double ld; const ld eps = 1e-12; int cmp(ld
 x, ld y = 0, ld tol = eps) { return ( x <= y + tol) ? (
x + tol < y) ? -1 : 0 : 1; } struct point{ ld x, y;
point(ld a, ld b) : x(a), y(b) {} point() {} }; struct
square{ ld x1, x2, y1, y2, a, b, c; point edges[4];
square(ld _a, ld _b, ld _c) { a = _a, b = _b, c = _c; x1
 = a - c * 0.5; x2 = a + c * 0.5; y1 = b - c * 0.5; y2 =
 b + c * 0.5; edges[0] = point(x1, y1); edges[1] = point
(x2, y1); edges[2] = point(x2, y2); edges[3] = point(x1,
 y2); } }; ld min_dist(point &a, point &b) { ld x = a.x
- b.x, y = a.y - b.y; return sqrt(x * x + y * y); } bool
 point_in_box(square s1, point p) { if (cmp(s1.x1, p.x)
!= 1 && cmp(s1.x2, p.x) != -1 && cmp(s1.y1, p.y) != 1 &&
 cmp(s1.y2, p.y) != -1) return true; return false; }
bool inside(square &s1, square &s2) { for(int i = 0; i <
 4; ++i) if(point_in_box(s2, s1.edges[i])) return true;
return false; } bool inside_vert(square &s1, square &s2)
 { if((cmp(s1.y1, s2.y1) != -1 && cmp(s1.y1, s2.y2) !=
```

```
1) || (cmp(s1.y2, s2.y1) != -1 && cmp(s1.y2, s2.y2) !=
1)) return true; return false; } bool inside_hori(square
 &s1, square &s2) { if ((cmp(s1.x1, s2.x1) != -1 && cmp(
s1.x1, s2.x2) != 1) || (cmp(s1.x2, s2.x1) != -1 && cmp(
s1.x2, s2.x2) != 1)) return true; return false; } ld
min_dist(square &s1, square &s2) { if (inside(s1, s2) ||
 inside(s2, s1)) return 0; ld ans = 1e100; for (int i =
0; i < 4; ++i) for (int j = 0; j < 4; ++j) ans = min(ans
, min_dist(s1.edges[i], s2.edges[j])); if (inside_hori(
s1, s2) || inside_hori(s2, s1)) { if (cmp(s1.y1, s2.y2)
!= -1) ans = min(ans, s1.y1 - s2.y2); else if (cmp(s2.y1
, s1.y2) != -1) ans = min(ans, s2.y1 - s1.y2); } if (
inside_vert(s1, s2) || inside_vert(s2, s1)) { if (cmp(s1
.x1, s2.x2) != -1) ans = min(ans, s1.x1 - s2.x2); else
if(cmp(s2.x1, s1.x2) != -1) ans = min(ans, s2.x1 - s1.x2
); } return ans; }
```

7.7 Pick$_s$*Theorem*

```
struct Point{ long long x, y; Point(){} Point(long long
x, long long y) : x(x), y(y){} }; /* /// twice the area
of polygon */ long long double_area(Point poly[], int n)
{ long long res = 0; for (int i = 0, j = n - 1; i < n; j
 = i++){ res += ((poly[j].x + poly[i].x) * (poly[j].y -
poly[i].y)); } return abs(res); } /* /// number of
lattice points strictly on polygon border */ long long
on_border(Point poly[], int n){ long long res = 0; for (
int i = 0, j = n - 1; i < n; j = i++){ res += __gcd(abs(
poly[i].x - poly[j].x), abs(poly[i].y - poly[j].y)); }
return res; } /* /// number of lattice points strictly
inside polygon */ long long interior(Point poly[], int n
){ long long res = 2 + double_area(poly, n) - on_border(
poly, n); return res / 2; }
```

7.8 convex

```
/* /// minkowski sum of two polygons in O(n) */ Polygon
minkowskiSum(Polygon A, Polygon B) { int n = A.size(), m
 = B.size(); rotate(A.begin(), min_element(A.begin(), A.
end()), A.end()); rotate(B.begin(), min_element(B.begin
(), B.end()), B.end()); A.push_back(A[0]); B.push_back(B
[0]); for (int i = 0; i < n; i++) A[i] = A[i + 1] - A[i
]; for (int i = 0; i < m; i++) B[i] = B[i + 1] - B[i];
Polygon C(n + m + 1); C[0] = A.back() + B.back(); merge(
A.begin(), A.end() - 1, B.begin(), B.end() - 1, C.begin
() + 1, polarComp(Point(0, 0), Point(0, -1))); for (int
i = 1; i < C.size(); i++) C[i] = C[i] + C[i - 1]; C.
pop_back(); return C; } /* // finds the rectangle with
minimum area enclosing a convex polygon and // the
rectangle with minimum perimeter enclosing a convex
polygon // Tf Ti Same */
pair<Tf, Tf> rotatingCalipersBoundingBox(const Polygon &
p) { using Linear::distancePointLine; int n = p.size();
int l = 1, r = 1, j = 1; Tf area = 1e100; Tf perimeter =
 1e100; for (int i = 0; i < n; i++) { Point v = (p[(i +
1) % n] - p[i]) / length(p[(i + 1) % n] - p[i]); while (
dcmp(dot(v, p[r % n] - p[i]) - dot(v, p[(r + 1) % n] - p
[i])) < 0) r++; while (j < r || dcmp(cross(v, p[j % n] -
 p[i]) - cross(v, p[(j + 1) % n] - p[i])) < 0) j++;
while (l < j || dcmp(dot(v, p[l % n] - p[i]) - dot(v, p
[(l + 1) % n] - p[i])) > 0) l++; Tf w = dot(v, p[r % n]
- p[i]) - dot(v, p[l % n] - p[i]); Tf h =
distancePointLine(p[j % n], Line(p[i], p[(i + 1) % n]));
 area = min(area, w * h); perimeter = min(perimeter, 2 *
 w + 2 * h); } return make_pair(area, perimeter); } /*
// returns the left side of polygon u after cutting it
by ray a->b */
Polygon cutPolygon(Polygon u, Point a, Point b) { using
Linear::lineLineIntersection; using Linear::onSegment;
Polygon ret; int n = u.size(); for (int i = 0; i < n; i
++) { Point c = u[i], d = u[(i + 1) % n]; if (dcmp(cross
(b - a, c - a)) >= 0) ret.push_back(c); if (dcmp(cross(b
 - a, d - c)) != 0) { Point t; lineLineIntersection(a, b
 - a, c, d - c, t); if (onSegment(t, Segment(c, d))) ret
.push_back(t); } } return ret; } /* // returns true if
point p is in or on triangle abc */
bool pointInTriangle(Point a, Point b, Point c, Point p)
 { return dcmp(cross(b - a, p - a)) >= 0 && dcmp(cross(c
 - b, p - b)) >= 0 && dcmp(cross(a - c, p - c)) >= 0; }
/* // pt must be in ccw order with no three collinear
points // returns inside = -1, on = 0, outside = 1 */
int pointInConvexPolygon(const Polygon &pt, Point p) {
int n = pt.size(); assert(n >= 3); int lo = 1, hi = n -
1; while (hi - lo > 1) { int mid = (lo + hi) / 2; if (
dcmp(cross(pt[mid] - pt[0], p - pt[0])) > 0) lo = mid;
else hi = mid; } bool in = pointInTriangle(pt[0], pt[lo
], pt[hi], p); if (!in) return 1; if (dcmp(cross(pt[lo]
- pt[lo - 1], p - pt[lo - 1])) == 0) return 0; if (dcmp(
cross(pt[hi] - pt[lo], p - pt[lo])) == 0) return 0; if (
dcmp(cross(pt[hi] - pt[(hi + 1) % n], p - pt[(hi + 1) %
n])) == 0) return 0; return -1; } /* // Extreme Point
for a direction is the farthest point in that direction
// u is the direction for extremeness */
int extremePoint(const Polygon &poly, Point u) { int n =
 (int)poly.size(); int a = 0, b = n; while (b - a > 1) {
 int c = (a + b) / 2; if (dcmp(dot(poly[c] - poly[(c +
1) % n], u)) >= 0 && dcmp(dot(poly[c] - poly[(c - 1 + n)
 % n], u)) >= 0) { return c; } bool a_up = dcmp(dot(poly
[(a + 1) % n] - poly[a], u)) >= 0; bool c_up = dcmp(dot(
poly[(c + 1) % n] - poly[c], u)) >= 0; bool a_above_c =
dcmp(dot(poly[a] - poly[c], u)) > 0; if (a_up && !c_up)
b = c; else if (!a_up && c_up) a = c; else if (a_up &&
c_up) { if (a_above_c) b = c; else a = c; } else { if (!
a_above_c) b = c; else a = c; } } if (dcmp(dot(poly[a] -
 poly[(a + 1) % n], u)) > 0 && dcmp(dot(poly[a] - poly[(
a - 1 + n) % n], u)) > 0) return a; return b % n; } /*
// For a convex polygon p and a line l, returns a list
of segments // of p that touch or intersect line l. //
the i'th segment is considered (p[i], p[(i + 1) modulo |
p|]) // #1 If a segment is collinear with the line, only
 that is returned // #2 Else if l goes through i'th
point, the i'th segment is added // Complexity: O(lg |p
|) */
vector<int> lineConvexPolyIntersection(const Polygon &p,
 Line l) { assert((int)p.size() >= 3); assert(l.a != l.b
); int n = p.size(); vector<int> ret; Point v = l.b - l.
a; int lf = extremePoint(p, rotate90(v)); int rt =
extremePoint(p, rotate90(v) * Ti(-1)); int olf = orient(
l.a, l.b, p[lf]); int ort = orient(l.a, l.b, p[rt]); if
(!olf || !ort) { int idx = (!olf ? lf : rt); if (orient(
l.a, l.b, p[(idx - 1 + n) % n]) == 0) ret.push_back((idx
 - 1 + n) % n); else ret.push_back(idx); return ret; }
if (olf == ort) return ret; for (int i = 0; i < 2; ++i)
{ int lo = i ? rt : lf; int hi = i ? lf : rt; int olo =
i ? ort : olf; while (true) { int gap = (hi - lo + n) %
n; if (gap < 2) break; int mid = (lo + gap / 2) % n; int
 omid = orient(l.a, l.b, p[mid]); if (!omid) { lo = mid;
 break; } if (omid == olo) lo = mid; else hi = mid; }
ret.push_back(lo); } return ret; } /* // Calculate [ACW,
 CW] tangent pair from an external point */
constexpr int CW = -1, ACW = 1; bool isGood(Point u,
Point v, Point Q, int dir) { return orient(Q, u, v) != -
dir; }
Point better(Point u, Point v, Point Q, int dir) {
return orient(Q, u, v) == dir ? u : v; }
Point pointPolyTangent(const Polygon &pt, Point Q, int
dir, int lo, int hi) { while (hi - lo > 1) { int mid = (
lo + hi) / 2; bool pvs = isGood(pt[mid], pt[mid - 1], Q,
 dir); bool nxt = isGood(pt[mid], pt[mid + 1], Q, dir);
```

```
if (pvs && nxt) return pt[mid]; if (!(pvs || nxt)) {
Point p1 = pointPolyTangent(pt, Q, dir, mid + 1, hi);
Point p2 = pointPolyTangent(pt, Q, dir, lo, mid - 1);
return better(p1, p2, Q, dir); } if (!pvs) { if (orient(
Q, pt[mid], pt[lo]) == dir) hi = mid - 1; else if (
better(pt[lo], pt[hi], Q, dir) == pt[lo]) hi = mid - 1;
else lo = mid + 1; } if (!nxt) { if (orient(Q, pt[mid],
pt[lo]) == dir) lo = mid + 1; else if (better(pt[lo], pt
[hi], Q, dir) == pt[lo]) hi = mid - 1; else lo = mid +
1; } } Point ret = pt[lo]; for (int i = lo + 1; i <= hi;
 i++) ret = better(ret, pt[i], Q, dir); return ret; } /*
 // [ACW, CW] Tangent */
pair<Point, Point> pointPolyTangents(const Polygon &pt,
Point Q) { int n = pt.size(); Point acw_tan =
pointPolyTangent(pt, Q, ACW, 0, n - 1); Point cw_tan =
pointPolyTangent(pt, Q, CW, 0, n - 1); return make_pair(
acw_tan, cw_tan); }
```

8 Misc

8.1 All Macros

```
//#pragma GCC optimize("Ofast")
//#pragma GCC optimization ("O3")
//#pragma comment(linker, "/stack:200000000")
//#pragma GCC optimize("unroll-loops")
//#pragma GCC target("sse,sse2,sse3,ssse3,sse4,popcnt,
abm,mmx,avx,tune=native")
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
    //find_by_order(k) --> returns iterator to the kth
    largest element counting from 0
    //order_of_key(val) --> returns the number of items
    in a set that are strictly smaller than our item
os.erase (os.find_by_order (os.order_of_key(v[i])))
  ==> to erase i-th element from ordered multiset
template <typename DT>
using ordered_set = tree <DT, null_type, less<DT>,
rb_tree_tag,tree_order_statistics_node_update>;
mod = {1500000007, 1500000013, 1500000023, 1500000057,
1500000077};
struct custom_hash {
  static uint64_t splitmix64 (uint64_t x) {
    x += 0x9e3779b97f4a7c15;
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    return x ^ (x >> 31);
  }
  size_t operator () (uint64_t x) const {
    static const uint64_t FIXED_RANDOM = chrono::
    steady_clock :: now ().time_since_epoch ().count ();
    return splitmix64 (x + FIXED_RANDOM);
  }
} Rng;
/*  for StressTesting
mt19937 rng(random_device{}()); int randomInt(int low,
int high) { uniform_int_distribution<int> dist(low, high
); return dist(rng); } vector<int> permutation(int n){
vector<int> p(n); iota(p.begin(), p.end(), 1); shuffle(p
.begin(), p.end(), rng); return p; }
*/
typedef gp_hash_table<int, int, custom_hash> gp;
int leap_years(int y) { return y / 4 - y / 100 + y /
400; }
bool is_leap(int y) { return y % 400 == 0 || (y % 4 == 0
 && y % 100 != 0); }
bool __builtin_mul_overflow (type1 a, type2 b, type3 &
res)
cin.tie(0)->ios_base::sync_with_stdio(0);
int getWeekday (int day, int month, int year) {
  if (month <= 2) {
    month += 12;
    year -= 1;
  }
  int f = (day + (13 * (month + 1)) / 5 + year + year /
  4 - year / 100 + year / 400) % 7;
  return f;
}
```

8.2 StressTest

```
#!/bin/bash
# Call as sh stress.sh ITERATIONS
g++ candidate.cpp -o candidate # candidate solution
g++ bruteforce.cpp -o bruteforce # bruteforce solution
g++ generator.cpp -o generator # test case generator
> all.txt
for i in $(seq 1 "$1") ; do
    echo "Attempt $i/$1"
    ./generator > in.txt
    echo "Attempt $i/$1" >> all.txt
    cat < in.txt >> all.txt
    ./bruteforce < in.txt > out1.txt
    ./candidate < in.txt > out2.txt
    diff -y out1.txt out2.txt > diff.txt
    if [ $? -ne 0 ] ; then
        echo -e "\nTest case:"
        cat in.txt
        echo -e "\nOutputs:"
        cat diff.txt
        break
    fi
done
files=("in.txt" "out1.txt" "out2.txt" "diff.txt" "
candidate" "bruteforce" "generator")
for file in "${files[@]}"; do
    rm "$file"
done
```

9 Equations and Formulas

9.1 Catalan Numbers, convolution and super

$C_n = \dfrac{1}{n+1}\dbinom{2n}{n}$ $C_0 = 1, C_1 = 1$ and $C_n = \displaystyle\sum_{k=0}^{n-1} C_k C_{n-1-k}$

The number of ways to completely parenthesize n+1 factors.
The number of triangulations of a convex polygon with n+2 sides (i.e. the number of partitions of polygon into disjoint triangles by using the diagonals).
The number of ways to connect the $2n$ points on a circle to form n disjoint i.e. non-intersecting chords.
Number of permutations of $1, \ldots, n$ that avoid the pattern 123 (or any of the other patterns of length 3); that is, the number of permutations with no three-term increasing sub-sequence. For $n = 3$, these permutations are 132, 213, 231, 312 and 321. $C_n^{(k)} = \dfrac{k+1}{n+k+1}\dbinom{2n+k}{n}$ $S(m,n) = \dfrac{(2m)!(2n)!}{m!n!(m+n)!}$

$$C_n^{(k)} = \frac{(2n+k-1)(2n+k)}{n(n+k+1)} C_{n-1}^{(k)}$$

9.2 Stirling Numbers First Kind

The Stirling numbers of the first kind count permutations according to their number of cycles (counting fixed points as cycles of length one).
$S(n,k)$ counts the number of permutations of n elements with k disjoint cycles.
$S(n,k) = (n-1)\cdot S(n-1,k) + S(n-1,k-1)$, *where*, $S(0,0) = 1, S(n,0) = S(0,n) = 0$ $\displaystyle\sum_{k=0}^{n} S(n,k) = n!$
The unsigned Stirling numbers may also be defined algebraically, as the coefficient of the rising factorial:

$$x^{\bar{n}} = x(x+1)...(x+n-1) = \sum_{k=0}^{n} S(n,k)x^k$$

Lets $[n,k]$ be the stirling number of the first kind, then

$$\left[{n \atop n-k}\right] = \sum_{0 \le i_1 < i_2 < i_k < n} i_1 i_2 .... i_k.$$

9.3 Stirling Numbers Second Kind

Stirling number of the second kind is the number of ways to partition a set of n objects into k non-empty subsets.
$S(n,k) = k \cdot S(n-1,k) + S(n-1,k-1)$, *where* $S(0,0) = 1, S(n,0) = S(0,n) = 0$ $S(n,2) = 2^{n-1} - 1$ $S(n,k)\cdot k!$ = number of ways to color n nodes using colors from 1 to k such that each color is used at least once.
An r-associated Stirling number of the second kind is the number of ways to partition a set of n objects into k subsets, with each subset containing at least r elements. It is denoted by $S_r(n,k)$ and obeys the recurrence relation. $S_r(n+1,k) = kS_r(n,k) + \dbinom{n}{r-1} S_r(n-r+1,k-1)$
Denote the n objects to partition by the integers $1, 2, \ldots, n$. Define the reduced Stirling numbers of the second kind, denoted $S^d(n,k)$, to be the number of ways to partition the integers $1, 2, \ldots, n$ into k nonempty subsets such that all elements in each subset have pairwise distance at least d. That is, for any integers i and j in a given subset, it is required that $|i-j| \ge d$. It has been shown that these numbers satisfy, $S^d(n,k) = S(n-d+1,k-d+1), n \ge k \ge d$

9.4 Other Combinatorial Identities

$\dbinom{n}{k} = \dfrac{n}{k}\dbinom{n-1}{k-1}$

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

$n, r \in N, n > r, \displaystyle\sum_{i=r}^{n}\binom{i}{r} = \binom{n+1}{r+1}$

If $P(n) = \displaystyle\sum_{k=0}^{n}\binom{n}{k}\cdot Q(k)$, then,

$$Q(n) = \sum_{k=0}^{n}(-1)^{n-k}\binom{n}{k}\cdot P(k)$$

If $P(n) = \displaystyle\sum_{k=0}^{n}(-1)^k\binom{n}{k}\cdot Q(k)$, then,

$$Q(n) = \sum_{k=0}^{n}(-1)^k\binom{n}{k}\cdot P(k)$$

9.5 Different Math Formulas

Picks Theorem : $A = i + b/2 - 1$
Deragements : $d(i) = (i-1)\times(d(i-1) + d(i-2))$

9.6 GCD and LCM

if m is any integer, then $\gcd(a + m{\cdot}b, b) = \gcd(a,b)$
The gcd is a multiplicative function in the following sense: if a_1 and a_2 are relatively prime, then $\gcd(a_1 \cdot a_2, b) = \gcd(a_1,b)\cdot\gcd(a_2,b)$.
$\gcd(a, \operatorname{lcm}(b,c)) = \operatorname{lcm}(\gcd(a,b), \gcd(a,c))$.
$\operatorname{lcm}(a, \gcd(b,c)) = \gcd(\operatorname{lcm}(a,b), \operatorname{lcm}(a,c))$.
For non-negative integers a and b, where a and b are not both zero, $\gcd(n^a - 1, n^b - 1) = n^{\gcd(a,b)} - 1$
$\gcd(a,b) = \displaystyle\sum_{k|a \text{ and } k|b} \phi(k)$

$\displaystyle\sum_{i=1}^{n}[\gcd(i,n) = k] = \phi\Big(\frac{n}{k}\Big)$

$\displaystyle\sum_{k=1}^{n}\gcd(k,n) = \sum_{d|n} d\cdot\phi\Big(\frac{n}{d}\Big)$

$\displaystyle\sum_{k=1}^{n}x^{\gcd(k,n)} = \sum_{d|n} x^d\cdot\phi\Big(\frac{n}{d}\Big)$

$\displaystyle\sum_{k=1}^{n}\frac{1}{\gcd(k,n)} = \sum_{d|n}\frac{1}{d}\cdot\phi\Big(\frac{n}{d}\Big) = \frac{1}{n}\sum_{d|n} d\cdot\phi(d)$

$\displaystyle\sum_{k=1}^{n}\frac{k}{\gcd(k,n)} = \frac{n}{2}\cdot\sum_{d|n}\frac{1}{d}\cdot\phi\Big(\frac{n}{d}\Big) = \frac{n}{2}\cdot\frac{1}{n}\cdot\sum_{d|n} d\cdot\phi(d)$

$\displaystyle\sum_{k=1}^{n}\frac{n}{\gcd(k,n)} = 2 * \sum_{k=1}^{n}\frac{k}{\gcd(k,n)} - 1$, for $n > 1$

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

$\displaystyle\sum_{i=1}^{n}\sum_{j=1}^{n}\gcd(i,j) = \sum_{d=1}^{n}\phi(d)\lfloor\frac{n}{d}\rfloor^2 = GPS(n-1) + 2\cdot\sum_{d|n} phi(d)\cdot\frac{n}{d} - n$

$\displaystyle\sum_{i=1}^{n}\sum_{j=1}^{n} i\cdot j[\gcd(i,j) = 1] = \sum_{i=1}^{n}\phi(i)i^2$

$F(n) = \displaystyle\sum_{i=1}^{n}\sum_{j=1}^{n}\operatorname{lcm}(i,j) = \sum_{l=1}^{n}\left(\frac{\left(1 + \lfloor\frac{n}{l}\rfloor\right)\left(\lfloor\frac{n}{l}\rfloor\right)}{2}\right)^2\sum_{d|l}\mu(d)ld$