

Contents

1	Setting	1
1.1	PS	1
2	Math	2
2.1	Basic Arithmetics	2
2.2	Convex Hull Trick	3
2.3	FFT	3
2.4	Chinese Remainder	3
3	Data Structure	4
3.1	Fenwick Tree	4
3.2	Merge Sort Tree	4
3.3	SegmentTree Lazy Propagation	4
3.4	Treap	5
4	Geometry	5
4.1	Basic Operations	5
4.2	Convex Hull	7
4.3	Poiont in Polygon	7
4.4	Polygon Cut	7
4.5	Rotating Calipers	8
4.6	Seperating Axis Theorem	8
4.7	Two Far Point	9
4.8	Two Nearest Point	9
5	Graph	9
5.1	Dijkstra	9
5.2	Bellman-Ford	10
5.3	Spfa	10
5.4	Topological Sort	11
5.5	Strongly Connected Component	11
5.6	2-SAT	11
5.7	Union Find	12
5.8	MST Kruskal	13
5.9	Lowest Common Ancestor	13
5.10	Maxflow dinic	14
5.11	Maxflow Edmonds-Karp	15
5.12	MCMF SPFA	15
5.13	MCMF	16
6	String	17
6.1	KMP	17
6.2	Manacher	17

6.3	Suffix Array	18
6.4	2nd Suffix Array	18
6.5	Trie	19
6.6	Aho-Corasick	19
6.7	Z Algorithm	19
7	Dynamic Programming	20
7.1	LIS	20
7.2	LIS only length	20
7.3	KnapSack	20
7.4	Coin Change	21
7.5	Bit Field DP	21
7.6	Knuth Optimization	21

1 Setting

1.1 PS

```
#include <bits/stdc++.h>

using namespace std;

#define for1(s, e) for(int i = s; i < e; i++)
#define for1j(s, e) for(int j = s; j < e; j++)
#define forEach(k) for(auto i : k)
#define forEachj(k) for(auto j : k)
#define sz(vct) vct.size()
#define all(vct) vct.begin(), vct.end()
#define sortv(vct) sort(vct.begin(), vct.end())
#define uniq(vct) sort(all(vct));vct.erase(unique(all(vct)), vct.end())
#define fi first
#define se second
#define INF (1ll << 60ll)

typedef unsigned long long ull;
typedef long long ll;
typedef ll llint;
typedef unsigned int uint;
typedef unsigned long long int ull;
typedef ull ullint;

typedef pair<int, int> pii;
typedef pair<ll, ll> pll;
typedef pair<double, double> pdd;
typedef pair<double, int> pdi;
typedef pair<string, string> pss;

typedef vector<int> iv1;
typedef vector<iv1> iv2;
typedef vector<ll> llv1;
```

```

typedef vector<llv1> llv2;

typedef vector<pii> piiv1;
typedef vector<piiv1> piiv2;
typedef vector<p11> p11v1;
typedef vector<p11v1> p11v2;
typedef vector<pdd> pddv1;
typedef vector<pddv1> pddv2;

const double EPS = 1e-8;
const double PI = acos(-1);

template<typename T>
T sq(T x) { return x * x; }

int sign(ll x) { return x < 0 ? -1 : x > 0 ? 1 : 0; }
int sign(int x) { return x < 0 ? -1 : x > 0 ? 1 : 0; }
int sign(double x) { return abs(x) < EPS ? 0 : x < 0 ? -1 : 1; }

void solve() {

}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(NULL); cout.tie(NULL);
    int tc = 1; // cin >> tc;
    while(tc--) solve();
}

```

## 2 Math

### 2.1 Basic Arithmetics

```

typedef long long ll;
typedef unsigned long long ull;

// calculate lg2(a)
inline int lg2(ll a) {
    return 63 - __builtin_clzll(a);
}

// calculate the number of 1-bits
inline int bitcount(ll a) {
    return __builtin_popcountll(a);
}

// calculate ceil(a/b)
// |a|, |b| <= (2^63)-1 (does not cover -2^63)
ll ceildiv(ll a, ll b) {
    if (b < 0) return ceildiv(-a, -b);
    if (a < 0) return (-a) / b;
    return ((ull)a + (ull)b - 1ull) / b;
}

```

```

// calculate floor(a/b)
// |a|, |b| <= (2^63)-1 (does not cover -2^63)
ll floordiv(ll a, ll b) {
    if (b < 0) return floordiv(-a, -b);
    if (a >= 0) return a / b;
    return -(ll)((ull)(-a) + b - 1) / b;
}

// calculate a*b % m
// x86-64 only
ll large_mod_mul(ll a, ll b, ll m) {
    return ll((__int128)a*(__int128)b%m);
}

// calculate a*b % m
// |m| < 2^62, x86 available
// O(Logb)
ll large_mod_mul(ll a, ll b, ll m) {
    a %= m; b %= m; ll r = 0, v = a;
    while (b) {
        if (b&1) r = (r + v) % m;
        b >>= 1;
        v = (v << 1) % m;
    }
    return r;
}

// calculate n^k % m
ll modpow(ll n, ll k, ll m) {
    ll ret = 1;
    n %= m;
    while (k) {
        if (k & 1) ret = large_mod_mul(ret, n, m);
        n = large_mod_mul(n, n, m);
        k /= 2;
    }
    return ret;
}

// calculate gcd(a, b)
ll gcd(ll a, ll b) {
    return b == 0 ? a : gcd(b, a % b);
}

// find a pair (c, d) s.t. ac + bd = gcd(a, b)
pair<ll, ll> extended_gcd(ll a, ll b) {
    if (b == 0) return { 1, 0 };
    auto t = extended_gcd(b, a % b);
    return { t.second, t.first - t.second * (a / b) };
}

// find x in [0, m) s.t. ax === gcd(a, m) (mod m)
ll modinverse(ll a, ll m) {
    return (extended_gcd(a, m).first % m + m) % m;
}

```

```

}

// calculate modular inverse for 1 ~ n
void calc_range_modinv(int n, int mod, int ret[]) {
    ret[1] = 1;
    for (int i = 2; i <= n; ++i)
        ret[i] = (11)(mod - mod/i) * ret[mod%i] % mod;
}

```

```

// p is prime
// calculate a^b % p
ll pow(ll a, ll b){
    if(b == 0) return 1;
    ll n = pow(a, b/2) % p;
    ll temp = (n * n) % p;

    if(b%2==0) return temp;
    return (a * temp) % p;
}

```

```

// p is prime
// calculate a/b % p
ll fermat(ll a, ll b){
    return a % p * pow(b, p-2) % p;
}

```

## 2.2 Convex Hull Trick

```

struct CHTLinear {
    struct Line {
        long long a, b;
        long long y(long long x) const { return a * x + b; }
    };
    vector<Line> stk;
    int qpt;
    CHTLinear() : qpt(0) { }
    // when you need maximum : (previous L).a < (now L).a
    // when you need minimum : (previous L).a > (now L).a
    void pushLine(const Line& l) {
        while (stk.size() > 1) {
            Line& l0 = stk[stk.size() - 1];
            Line& l1 = stk[stk.size() - 2];
            if ((l0.b - l1.b) * (l0.a - l1.a) > (l1.b - l0.b) * (l.a - l0.a))
                break;
            stk.pop_back();
        }
        stk.push_back(l);
    }
    // (previous x) <= (current x)
    // it calculates max/min at x
    long long query(long long x) {
        while (qpt + 1 < stk.size()) {
            Line& l0 = stk[qpt];
            Line& l1 = stk[qpt + 1];

```

```

            if (l1.a - l0.a > 0 && (l0.b - l1.b) > x * (l1.a - l0.a)) break;
            if (l1.a - l0.a < 0 && (l0.b - l1.b) < x * (l1.a - l0.a)) break;
            ++qpt;
        }
        return stk[qpt].y(x);
    }
};

```

## 2.3 FFT

```

void fft(int sign, int n, double *real, double *imag) {
    double theta = sign * 2 * pi / n;
    for (int m = n; m >= 2; m >>= 1, theta *= 2) {
        double wr = 1, wi = 0, c = cos(theta), s = sin(theta);
        for (int i = 0, mh = m >> 1; i < mh; ++i) {
            for (int j = i; j < n; j += m) {
                int k = j + mh;
                double xr = real[j] - real[k], xi = imag[j] - imag[k];
                real[j] += real[k], imag[j] += imag[k];
                real[k] = wr * xr - wi * xi, imag[k] = wr * xi + wi * xr;
            }
            double _wr = wr * c - wi * s, _wi = wr * s + wi * c;
            wr = _wr, wi = _wi;
        }
    }
    for (int i = 1, j = 0; i < n; ++i) {
        for (int k = n >> 1; k > (j ^= k); k >>= 1);
        if (j < i) swap(real[i], real[j]), swap(imag[i], imag[j]);
    }
}

// Compute Poly(a)*Poly(b), write to r; Indexed from 0
// O(n*Logn)
int mult(int *a, int n, int *b, int m, int *r) {
    const int maxn = 100;
    static double ra[maxn], rb[maxn], ia[maxn], ib[maxn];
    int fn = 1;
    while (fn < n + m) fn <= 1; // n + m: interested length
    for (int i = 0; i < n; ++i) ra[i] = a[i], ia[i] = 0;
    for (int i = n; i < fn; ++i) ra[i] = ia[i] = 0;
    for (int i = 0; i < m; ++i) rb[i] = b[i], ib[i] = 0;
    for (int i = m; i < fn; ++i) rb[i] = ib[i] = 0;
    fft(1, fn, ra, ia);
    fft(1, fn, rb, ib);
    for (int i = 0; i < fn; ++i) {
        double real = ra[i] * rb[i] - ia[i] * ib[i];
        double imag = ra[i] * ib[i] + rb[i] * ia[i];
        ra[i] = real, ia[i] = imag;
    }
    fft(-1, fn, ra, ia);
    for (int i = 0; i < fn; ++i) r[i] = (int)floor(ra[i] / fn + 0.5);
    return fn;
}

```

## 2.4 Chinese Remainder

```
// find x s.t. x === a[0] (mod n[0])
//             === a[1] (mod n[1])
//             ...
// assumption: gcd(n[i], n[j]) = 1
ll chinese_remainder(ll* a, ll* n, int size) {
    if (size == 1) return *a;
    ll tmp = modinverse(n[0], n[1]);
    ll tmp2 = (tmp * (a[1] - a[0])) % n[1] + n[1] % n[1];
    ll ora = a[1];
    ll tgcd = gcd(n[0], n[1]);
    a[1] = a[0] + n[0] / tgcd * tmp2;
    n[1] *= n[0] / tgcd;
    ll ret = chinese_remainder(a + 1, n + 1, size - 1);
    n[1] /= n[0] / tgcd;
    a[1] = ora;
    return ret;
}
```

## 3 Data Structure

### 3.1 Fenwick Tree

```
const int TSIZE = 100000;
int tree[TSIZE + 1];

// Returns the sum from index 1 to p, inclusive
int query(int p) {
    int ret = 0;
    for (; p > 0; p -= p & -p) ret += tree[p];
    return ret;
}

// Adds val to element with index pos
void add(int p, int val) {
    for (; p <= TSIZE; p += p & -p) tree[p] += val;
}
```

### 3.2 Merge Sort Tree

```
llv1 a;
llv1 mTree[Mx];
void makeTree(ll idx, ll ss, ll se) {
    if (ss == se) {
        mTree[idx].push_back(a[ss]);
        return;
    }
    ll mid = (ss + se) / 2;
    makeTree(2 * idx + 1, ss, mid);
    makeTree(2 * idx + 2, mid + 1, se);
    merge(mTree[2 * idx + 1].begin(), mTree[2 * idx + 1].end(), mTree[2 * idx + 2].begin(), mTree[2 * idx + 2].end(), back_inserter(mTree[idx]));
}
ll query(ll node, ll start, ll end, ll q_s, ll q_e, ll k) {
```

```
// i j k: Ai, Ai+1, ..., 로Aj 이루어진부분수열중에서보다 k 큰원소의개수를출력한
// 다 .
if (q_s > end || start > q_e) return 0;
if (q_s <= start && q_e >= end) {
    return mTree[node].size() - (upper_bound(mTree[node].begin(), mTree[node].end(), k) - mTree[node].begin());
}
ll mid = (start + end) / 2;
ll p1 = query(2 * node + 1, start, mid, q_s, q_e, k);
ll p2 = query(2 * node + 2, mid + 1, end, q_s, q_e, k);
return p1 + p2;
}
```

### 3.3 SegmentTree Lazy Propagation

```
// example implementation of sum tree
const int TSIZE = 131072; // always 2^k form && n <= TSIZE
int segtree[TSIZE * 2], prop[TSIZE * 2];
void seg_init(int nod, int l, int r) {
    if (l == r) segtree[nod] = dat[l];
    else {
        int m = (l + r) >> 1;
        seg_init(nod << 1, l, m);
        seg_init(nod << 1 | 1, m + 1, r);
        segtree[nod] = segtree[nod << 1] + segtree[nod << 1 | 1];
    }
}
void seg_relax(int nod, int l, int r) {
    if (prop[nod] == 0) return;
    if (l < r) {
        int m = (l + r) >> 1;
        segtree[nod << 1] += (m - l + 1) * prop[nod];
        prop[nod << 1] += prop[nod];
        segtree[nod << 1 | 1] += (r - m) * prop[nod];
        prop[nod << 1 | 1] += prop[nod];
    }
    prop[nod] = 0;
}
int seg_query(int nod, int l, int r, int s, int e) {
    if (r < s || e < l) return 0;
    if (s <= l && r <= e) return segtree[nod];
    seg_relax(nod, l, r);
    int m = (l + r) >> 1;
    return seg_query(nod << 1, l, m, s, e) + seg_query(nod << 1 | 1, m + 1, r, s, e);
}
void seg_update(int nod, int l, int r, int s, int e, int val) {
    if (r < s || e < l) return;
    if (s <= l && r <= e) {
        segtree[nod] += (r - l + 1) * val;
        prop[nod] += val;
        return;
    }
    seg_relax(nod, l, r);
    int m = (l + r) >> 1;
```

```

    seg_update(nod << 1, l, m, s, e, val);
    seg_update(nod << 1 | 1, m + 1, r, s, e, val);
    segtree[nod] = segtree[nod << 1] + segtree[nod << 1 | 1];
}
// usage:
// seg_update(1, 0, n - 1, qs, qe, val);
// seg_query(1, 0, n - 1, qs, qe);

```

### 3.4 Treap

```

// Treap* root = NULL;
// root = insert(root, new Treap(3));
typedef int type;
struct Treap {
    Treap* left = NULL, * right = NULL;
    int size = 1, prio = rand();
    type key;
    Treap(type key) : key(key) { }
    void calcSize() {
        size = 1;
        if (left != NULL) size += left->size;
        if (right != NULL) size += right->size;
    }
    void setLeft(Treap* l) { left = l, calcSize(); }
    void setRight(Treap* r) { right = r, calcSize(); }
};
typedef pair<Treap*, Treap*> TPair;
TPair split(Treap* root, type key) {
    if (root == NULL) return TPair(NULL, NULL);
    if (root->key < key) {
        TPair rs = split(root->right, key);
        root->setRight(rs.first);
        return TPair(root, rs.second);
    }
    TPair ls = split(root->left, key);
    root->setLeft(ls.second);
    return TPair(ls.first, root);
}
Treap* insert(Treap* root, Treap* node) {
    if (root == NULL) return node;
    if (root->prio < node->prio) {
        TPair s = split(root, node->key);
        node->setLeft(s.first);
        node->setRight(s.second);
        return node;
    }
    else if (node->key < root->key)
        root->setLeft(insert(root->left, node));
    else
        root->setRight(insert(root->right, node));
    return root;
}
Treap* merge(Treap* a, Treap* b) {
    if (a == NULL) return b;
    if (b == NULL) return a;

```

```

    if (a->prio < b->prio) {
        b->setLeft(merge(a, b->left));
        return b;
    }
    a->setRight(merge(a->right, b));
    return a;
}
Treap* erase(Treap* root, type key) {
    if (root == NULL) return root;
    if (root->key == key) {
        Treap* ret = merge(root->left, root->right);
        delete root;
        return ret;
    }
    if (key < root->key)
        root->setLeft(erase(root->left, key));
    else
        root->setRight(erase(root->right, key));
    return root;
}
Treap* kth(Treap* root, int k) { // kth key
    int l_size = 0;
    if (root->left != NULL) l_size += root->left->size;
    if (k <= l_size) return kth(root->left, k);
    if (k == l_size + 1) return root;
    return kth(root->right, k - l_size - 1);
}
int countLess(Treap* root, type key) { // count less than key
    if (root == NULL) return 0;
    if (root->key >= key)
        return countLess(root->left, key);
    int ls = (root->left ? root->left->size : 0);
    return ls + 1 + countLess(root->right, key);
}

```

## 4 Geometry

### 4.1 Basic Operations

```

const double eps = 1e-9;

inline int diff(double lhs, double rhs) {
    if (lhs - eps < rhs && rhs < lhs + eps) return 0;
    return (lhs < rhs) ? -1 : 1;
}

inline bool is_between(double check, double a, double b) {
    if (a < b)
        return (a - eps < check && check < b + eps);
    else
        return (b - eps < check && check < a + eps);
}

struct Point {

```

```

double x, y;
bool operator==(const Point& rhs) const {
    return diff(x, rhs.x) == 0 && diff(y, rhs.y) == 0;
}
Point operator+(const Point& rhs) const {
    return Point{ x + rhs.x, y + rhs.y };
}
Point operator-(const Point& rhs) const {
    return Point{ x - rhs.x, y - rhs.y };
}
Point operator*(double t) const {
    return Point{ x * t, y * t };
}
};

struct Circle {
    Point center;
    double r;
};

struct Line {
    Point pos, dir;
};

inline double inner(const Point& a, const Point& b) {
    return a.x * b.x + a.y * b.y;
}

inline double outer(const Point& a, const Point& b) {
    return a.x * b.y - a.y * b.x;
}

inline int ccw_line(const Line& line, const Point& point) {
    return diff(outer(line.dir, point - line.pos), 0);
}

inline int ccw(const Point& a, const Point& b, const Point& c) {
    return diff(outer(b - a, c - a), 0);
}

inline double dist(const Point& a, const Point& b) {
    return sqrt(inner(a - b, a - b));
}

inline double dist2(const Point &a, const Point &b) {
    return inner(a - b, a - b);
}

inline double dist(const Line& line, const Point& point, bool segment = false) {
    double c1 = inner(point - line.pos, line.dir);
    if (segment && diff(c1, 0) <= 0) return dist(line.pos, point);
    double c2 = inner(line.dir, line.dir);
    if (segment && diff(c2, c1) <= 0) return dist(line.pos + line.dir, point);
    return dist(line.pos + line.dir * (c1 / c2), point);
}

```

```

bool get_cross(const Line& a, const Line& b, Point& ret) {
    double mdet = outer(b.dir, a.dir);
    if (diff(mdet, 0) == 0) return false;
    double t2 = outer(a.dir, b.pos - a.pos) / mdet;
    ret = b.pos + b.dir * t2;
    return true;
}

bool get_segment_cross(const Line& a, const Line& b, Point& ret) {
    double mdet = outer(b.dir, a.dir);
    if (diff(mdet, 0) == 0) return false;
    double t1 = -outer(b.pos - a.pos, b.dir) / mdet;
    double t2 = outer(a.dir, b.pos - a.pos) / mdet;
    if (!is_between(t1, 0, 1) || !is_between(t2, 0, 1)) return false;
    ret = b.pos + b.dir * t2;
    return true;
}

Point inner_center(const Point &a, const Point &b, const Point &c) {
    double wa = dist(b, c), wb = dist(c, a), wc = dist(a, b);
    double w = wa + wb + wc;
    return Point{ (wa * a.x + wb * b.x + wc * c.x) / w, (wa * a.y + wb * b.y +
        wc * c.y) / w };
}

Point outer_center(const Point &a, const Point &b, const Point &c) {
    Point d1 = b - a, d2 = c - a;
    double area = outer(d1, d2);
    double dx = d1.x * d1.x * d2.y - d2.x * d2.x * d1.y
        + d1.y * d2.y * (d1.y - d2.y);
    double dy = d1.y * d1.y * d2.x - d2.y * d2.y * d1.x
        + d1.x * d2.x * (d1.x - d2.x);
    return Point{ a.x + dx / area / 2.0, a.y - dy / area / 2.0 };
}

vector<Point> circle_line(const Circle& circle, const Line& line) {
    vector<Point> result;
    double a = 2 * inner(line.dir, line.dir);
    double b = 2 * (line.dir.x * (line.pos.x - circle.center.x)
        + line.dir.y * (line.pos.y - circle.center.y));
    double c = inner(line.pos - circle.center, line.pos - circle.center)
        - circle.r * circle.r;
    double det = b * b - 2 * a * c;
    int pred = diff(det, 0);
    if (pred == 0)
        result.push_back(line.pos + line.dir * (-b / a));
    else if (pred > 0) {
        det = sqrt(det);
        result.push_back(line.pos + line.dir * ((-b + det) / a));
        result.push_back(line.pos + line.dir * ((-b - det) / a));
    }
    return result;
}

```

```

vector<Point> circle_circle(const Circle& a, const Circle& b) {
    vector<Point> result;
    int pred = diff(dist(a.center, b.center), a.r + b.r);
    if (pred > 0) return result;
    if (pred == 0) {
        result.push_back((a.center * b.r + b.center * a.r) * (1 / (a.r + b.r)));
        return result;
    }
    double aa = a.center.x * a.center.x + a.center.y * a.center.y - a.r * a.r;
    double bb = b.center.x * b.center.x + b.center.y * b.center.y - b.r * b.r;
    double tmp = (bb - aa) / 2.0;
    Point cdiff = b.center - a.center;
    if (diff(cdiff.x, 0) == 0) {
        if (diff(cdiff.y, 0) == 0)
            return result; // if (diff(a.r, b.r) == 0): same circle
        return circle_line(a, Line{ Point{ 0, tmp / cdiff.y }, Point{ 1, 0 } });
    }
    return circle_line(a,
        Line{ Point{ tmp / cdiff.x, 0 }, Point{ -cdiff.y, cdiff.x } });
}

Circle circle_from_3pts(const Point& a, const Point& b, const Point& c) {
    Point ba = b - a, cb = c - b;
    Line p{ (a + b) * 0.5, Point{ ba.y, -ba.x } };
    Line q{ (b + c) * 0.5, Point{ cb.y, -cb.x } };
    Circle circle;
    if (!get_cross(p, q, circle.center))
        circle.r = -1;
    else
        circle.r = dist(circle.center, a);
    return circle;
}

Circle circle_from_2pts_rad(const Point& a, const Point& b, double r) {
    double det = r * r / dist2(a, b) - 0.25;
    Circle circle;
    if (det < 0)
        circle.r = -1;
    else {
        double h = sqrt(det);
        // center is to the left of a->b
        circle.center = (a + b) * 0.5 + Point{ a.y - b.y, b.x - a.x } * h;
        circle.r = r;
    }
    return circle;
}

```

## 4.2 Convex Hull

```

// find convex hull
// O(n*logn)
vector<Point> convex_hull(vector<Point>& dat) {
    if (dat.size() <= 3) return dat;
    vector<Point> upper, lower;
    sort(dat.begin(), dat.end(), [](const Point& a, const Point& b) {

```

```

        return (a.x == b.x) ? a.y < b.y : a.x < b.x;
    });
    for (const auto& p : dat) {
        while (upper.size() >= 2 && ccw(++upper.rbegin(), *upper.rbegin(), p)
            >= 0) upper.pop_back();
        while (lower.size() >= 2 && ccw(++lower.rbegin(), *lower.rbegin(), p)
            <= 0) lower.pop_back();
        upper.emplace_back(p);
        lower.emplace_back(p);
    }
    upper.insert(upper.end(), ++lower.rbegin(), --lower.rend());
    return upper;
}

```

## 4.3 Point in Polygon

```

typedef double coord_t;

inline coord_t is_left(Point p0, Point p1, Point p2) {
    return (p1.x - p0.x) * (p2.y - p0.y) - (p2.x - p0.x) * (p1.y - p0.y);
}

// point in polygon test
// http://geomalgorithms.com/a03-inclusion.html
bool is_in_polygon(Point p, vector<Point>& poly) {
    int wn = 0;
    for (int i = 0; i < poly.size(); ++i) {
        int ni = (i + 1 == poly.size()) ? 0 : i + 1;
        if (poly[i].y <= p.y) {
            if (poly[ni].y > p.y) {
                if (is_left(poly[i], poly[ni], p) > 0) {
                    ++wn;
                }
            }
        }
        else {
            if (poly[ni].y <= p.y) {
                if (is_left(poly[i], poly[ni], p) < 0) {
                    --wn;
                }
            }
        }
    }
    return wn != 0;
}

```

## 4.4 Polygon Cut

```

// Left side of a->b
vector<Point> cut_polygon(const vector<Point>& polygon, Line line) {
    if (!polygon.size()) return polygon;
    typedef vector<Point>::const_iterator piter;
    piter la, lan, fi, fip, i, j;
    la = lan = fi = fip = polygon.end();

```

```

i = polygon.end() - 1;
bool lastin = diff(ccw_line(line, polygon[polygon.size() - 1]), 0) > 0;
for (j = polygon.begin(); j != polygon.end(); j++) {
    bool thisin = diff(ccw_line(line, *j), 0) > 0;
    if (lastin && !thisin) {
        la = i;
        lan = j;
    }
    if (!lastin && thisin) {
        fi = j;
        fip = i;
    }
    i = j;
    lastin = thisin;
}
if (fi == polygon.end()) {
    if (!lastin) return vector<Point>();
    return polygon;
}
vector<Point> result;
for (i = fi; i != lan; i++) {
    if (i == polygon.end()) {
        i = polygon.begin();
        if (i == lan) break;
    }
    result.push_back(*i);
}
Point lc, fc;
get_cross(Line{ *la, *lan - *la }, line, lc);
get_cross(Line{ *fip, *fi - *fip }, line, fc);
result.push_back(lc);
if (diff(dist2(lc, fc), 0) != 0) result.push_back(fc);
return result;
}

```

## 4.5 Rotating Calipers

```

// get all antipodal pairs
// O(n)
void antipodal_pairs(vector<Point>& pt) {
    // calculate convex hull
    sort(pt.begin(), pt.end(), [](const Point& a, const Point& b) {
        return (a.x == b.x) ? a.y < b.y : a.x < b.x;
    });
    vector<Point> up, lo;
    for (const auto& p : pt) {
        while (up.size() >= 2 && ccw(++up.rbegin(), *up.rbegin(), p) >= 0) up.
            pop_back();
        while (lo.size() >= 2 && ccw(++lo.rbegin(), *lo.rbegin(), p) <= 0) lo.
            pop_back();
        up.emplace_back(p);
        lo.emplace_back(p);
    }

    for (int i = 0, j = (int)lo.size() - 1; i + 1 < up.size() || j > 0; ) {

```

```

        get_pair(up[i], lo[j]); // DO WHAT YOU WANT
        if (i + 1 == up.size()) {
            --j;
        }
        else if (j == 0) {
            ++i;
        }
        else if ((long long)(up[i + 1].y - up[i].y) * (lo[j].x - lo[j - 1].x)
            > (long long)(up[i + 1].x - up[i].x) * (lo[j].y - lo[j - 1].y)) {
            ++i;
        }
        else {
            --j;
        }
    }
}

```

## 4.6 Separating Axis Theorem

```

pair<double, double> get_projection(vector<Vector2> &points, Vector2 &axis) {
    double min_val = axis.dot(points[0]);
    double max_val = min_val;
    for (int i = 1; i < points.size(); i++) {
        double projected = axis.dot(points[i]);
        min_val = min(min_val, projected);
        max_val = max(max_val, projected);
    }
    return {min_val, max_val};
}

vector<Vector2> get_normals(vector<Vector2> &points) {
    vector<Vector2> ret;
    if (points.size() == 1)
        return ret;
    for (int i = 0; i < points.size(); i++) {
        Vector2 &a = points[i];
        Vector2 &b = points[(i + 1) % points.size()];
        ret.push_back(Vector2((b - a).y, -(b - a).x));
    }
    return ret;
}

bool can_separate(vector<Vector2> &A, vector<Vector2> &B) {
    if (A.size() == 1 && B.size() == 1)
        return true;
    auto c_a = get_convex_hull(A);
    auto c_b = get_convex_hull(B);
    auto n_a = get_normals(c_a);
    auto n_b = get_normals(c_b);

    n_a.insert(n_a.end(), n_b.begin(), n_b.end());
    if (c_a.size() > 1) n_a.push_back(Vector2(c_a[1] - c_a[0]));
    if (c_b.size() > 1) n_a.push_back(Vector2(c_b[1] - c_b[0]));

    for (Vector2 &axis : n_a) {

```



```

    auto p_a = get_projection(c_a, axis);
    auto p_b = get_projection(c_b, axis);
    if (!((p_a.second >= p_b.first) && (p_b.second >= p_a.first))) return true;
}
return false;
}

```

## 4.7 Two Far Point

```

pair<Vector2, Vector2> get_max_points(vector<Vector2> &points) {
    int left = 0, right = max_element(points.begin(), points.end()) - points.begin();
    int ret1 = left, ret2 = right;
    double max_len = (points[right] - points[left]).norm();
    int end = right;
    Vector2 left_dir = Vector2(0, -1.0);
    vector<Vector2> edges;
    for (int i = 0; i < points.size(); i++)
        edges.push_back((points[(i + 1) % points.size()] - points[i]).normalized());
    while (right != 0 || left != end) {
        double next1 = left_dir.dot(edges[left]);
        double next2 = -left_dir.dot(edges[right]);
        if (left != end && (right == 0 || next1 > next2)) {
            left_dir = edges[left];
            left = (left + 1) % points.size();
        } else {
            left_dir = -edges[right];
            right = (right + 1) % points.size();
        }
        double len = (points[right] - points[left]).norm();
        if (len > max_len) {
            ret1 = left;
            ret2 = right;
            max_len = len;
        }
    }
    return {points[ret1], points[ret2]};
}

```

## 4.8 Two Nearest Point

```

int dist(Point &p, Point &q) {
    return (p.x - q.x) * (p.x - q.x) + (p.y - q.y) * (p.y - q.y);
}
struct Comp {
    bool comp_in_x;
    Comp(bool b) : comp_in_x(b) {}
    bool operator()(Point &p, Point &q) {
        return (this->comp_in_x ? p.x < q.x : p.y < q.y);
    }
};
int nearest(vector<Point>::iterator it, int n) {
    if (n == 2) return dist(it[0], it[1]);
    if (n == 3) return min({dist(it[0], it[1]), dist(it[1], it[2]), dist(it[2], it[0])});
}

```

```

int line = (it[n / 2 - 1].x + it[n / 2].x) / 2;
int d = min(nearest(it, n / 2), nearest(it + n / 2, n - n / 2));
vector<Point> mid;
for (int i = 0; i < n; i++) {
    int t = line - it[i].x;
    if (t * t < d) mid.push_back(it[i]);
}
sort(mid.begin(), mid.end(), Comp(false));
int mid_sz = mid.size();
for (int i = 0; i < mid_sz - 1; i++)
    for (int j = i + 1; j < mid_sz && (mid[j].y - mid[i].y) * (mid[j].y - mid[i].y) < d; j++)
        d = min(d, dist(mid[i], mid[j]));
return d;
}

```

## 5 Graph

### 5.1 Dijkstra

```

template<typename T> struct Dijkstra {
    /*
     * T: 간선가중치타입
     */
    struct Edge {
        ll node;
        T cost;
        bool operator<(const Edge &to) const {
            return cost > to.cost;
        }
    };

    ll n;
    vector<vector<Edge>> adj;
    vector<ll> prev;

    Dijkstra(ll n) : n{n}, adj(n+1) {}

    void addEdge(ll s, ll e, T cost) {
        adj[s].push_back(Edge(e, cost));
    }

    void addUndirectedEdge(ll s, ll e, T cost) {
        addEdge(s, e, cost);
        addEdge(e, s, cost);
    }

    vector<ll> dijkstra(ll s) {
        vector<ll> dist(n+1, INF);
        prev.resize(n+1, -1);
        priority_queue<edge> pq;
        pq.push({s, 0ll});
        dist[s] = 0;
        while (!pq.empty()) {

```

```

    edge cur = pq.top();
    pq.pop();
    if (cur.cost > dist[cur.node]) continue;
    for (auto &nxt : adj[cur.node])
        if (dist[cur.node] + nxt.cost < dist[nxt.node]) {
            prev[nxt.node] = cur.node;
            dist[nxt.node] = dist[cur.node] + nxt.cost;
            pq.push({ nxt.node, dist[nxt.node] });
        }
    }
    return dist;
}

vector<ll> getPath(ll s, ll e) {
    vector<ll> ret;
    ll current = e;
    while(current != -1) {
        ret.push_back(current);
        current = prev[current];
    }
    reverse(ret.begin(), ret.end());
    return ret;
}
};

```

## 5.2 Bellman-Ford

```

struct BellmanFord {
    struct BellmanEdge {
        ll to, cost;

        BellmanEdge(ll to, ll cost) : to(to), cost(cost) {}
    };

    ll N;
    vector<vector< BellmanEdge > > adj;
    llv1 D;
    vector<ll> prev;

    BellmanFord(ll N) : N(N) {
        adj.resize(N + 1);
    }

    void addEdge(ll s, ll e, ll cost) {
        adj[s].push_back(BellmanEdge(e, cost));
    }

    bool run(ll start_point) {
        // 음수간선 cycle 유무를반환합니다 .
        // 거리정보는 D 벡터에저장됩니다 .
        // O(V * E)

        D.resize(N + 1, INF);
        prev.resize(N + 1, -1);
        D[start_point] = 0;
    }
};

```

```

bool isCycle = false;

for(1, N + 1) {
    for(1, N + 1) {
        for(int k=0; k < sz(adj[j]); k++) {
            BellmanEdge p = adj[j][k];
            int end = p.to;
            ll dist = D[j] + p.cost;

            if (D[j] != INF && D[end] > dist) {
                D[end] = dist;
                if (i == N) isCycle = true;
            }
        }
    }
}

return isCycle;
}

llv1 getPath(ll s, ll e) {
    vector<ll> ret;
    ll current = e;
    while(current != -1) {
        ret.push_back(current);
        current = prev[current];
    }
    reverse(ret.begin(), ret.end());
    return ret;
}
};

```

## 5.3 Spfa

*// shortest path faster algorithm  
// average for random graph : O(E) , worst : O(VE)*

```

const int MAXN = 20001;
const int INF = 100000000;
int n, m;
vector<pii> graph[MAXN];

bool inqueue[MAXN];
int dist[MAXN];

void spfa(int start) {
    for (int i = 0; i < n; ++i) dist[i] = INF;
    dist[start] = 0;

    queue<int> q;
    q.push(start);
    inqueue[start] = true;

    while (!q.empty()) {
        int here = q.front();
    }
}

```

```

q.pop();

inqueue[here] = false;
for (auto& nxt : graph[here]) {
    if (dist[here] + nxt.second < dist[nxt.first]) {
        dist[nxt.first] = dist[here] + nxt.second;
        if (!inqueue[nxt.first]) {
            q.push(nxt.first);
            inqueue[nxt.first] = true;
        }
    }
}
}
}
}
};

```

## 5.4 Topological Sort

```

struct TopologicalSort {
    // 1-index

    int n;
    iv1 in_degree;
    iv2 graph;
    iv1 result;

    TopologicalSort(int n) : n(n) {
        in_degree.resize(n + 1, 0);
        graph.resize(n + 1);
    }

    void addEdge(int s, int e) {
        graph[s].push_back(e);
        in_degree[e]++;
    }

    void run() {
        queue<int> q;

        for(1, n+1) {
            if(in_degree[i] == 0) q.push(i);
        }
        while(!q.empty()) {
            int here = q.front(); q.pop();
            result.push_back(here);

            for(0, sz(graph[here])) {
                int there = graph[here][i];

                if(--in_degree[there]==0) q.push(there);
            }
        }
    }
};

```

## 5.5 Strongly Connected Component

```

struct SCC {
    // 1-index
    // run() 후에 components 결과가담김 .

    ll V;
    llv2 edges, reversed_edges, components;
    vector<bool> visited;
    stack<ll> visit_log;

    SCC(ll V): V(V) {
        edges.resize(V + 1);
        reversed_edges.resize(V + 1);
    }

    void addEdge(int s, int e) {
        edges[s].push_back(e);
        reversed_edges[e].push_back(s);
    }

    void dfs(int node) {
        visited[node] = true;

        for (int next : edges[node])
            if (!visited[next]) dfs(next);
        visit_log.push(node);
    }

    void dfs2(int node) {
        visited[node] = true;
        for (int next:reversed_edges[node])
            if (!visited[next]) dfs2(next);
        components.back().push_back(node);
    }

    void run() {
        visited = vector<bool>(V + 1, false);
        for (int node = 1; node <= V; node++)
            if (!visited[node]) dfs(node);

        visited = vector<bool>(V + 1, false);
        while (!visit_log.empty()) {
            ll node = visit_log.top(); visit_log.pop();
            if (!visited[node]) {
                components.push_back(llv1());
                dfs2(node);
            }
        }
    }
};

```

## 5.6 2-SAT

```

struct Graph {
    int V;

```

```

vector<bool> visited;
stack<int> visit_stack;
vector<int> component_number, source_components;
vector<vector<int>> edges, reversed_edges, components, components_edges;

Graph(int V) : V(V) {
    edges.resize(V);
    reversed_edges.resize(V);
}

void append(int u, int v) {
    edges[u].push_back(v);
    reversed_edges[v].push_back(u);
}

void dfs(int node) {
    visited[node] = true;
    for (int next : edges[node])
        if (!visited[next])
            dfs(next);
    visit_stack.push(node);
}

void scc(int node) {
    visited[node] = true;
    for (int next : reversed_edges[node])
        if (!visited[next])
            scc(next);
    components.back().push_back(node);
}

void build_scc() {
    visited = vector<bool>(V, false);
    for (int node = 0; node < V; node++)
        if (!visited[node]) dfs(node);
    visited = vector<bool>(V, false);
    while (!visit_stack.empty()) {
        int node = visit_stack.top();
        visit_stack.pop();
        if (!visited[node]) {
            components.push_back(vector<int>());
            scc(node);
        }
    }
    component_number.resize(V);
    for (int i = 0; i < components.size(); i++)
        for (int node : components[i]) component_number[node] = i;
    vector<bool> is_source = vector<bool>(components.size(), true);
    components_edges.resize(components.size());
    for (int u = 0; u < V; u++)
        for (int v : edges[u]) {
            int cu = component_number[u];
            int cv = component_number[v];
            if (cu == cv) continue;
            components_edges[cu].push_back(cv);
            is_source[cv] = false;
        }
    for (int component = 0; component < components.size(); component++) {
        if (is_source[component]) source_components.push_back(component);
    }
}

```

```

    }
}

int main(void) {
    int V, E;
    cin >> V >> E;
    Graph graph(2 * V + 1);
    for (int i = 0; i < E; i++) {
        int u, v;
        cin >> u >> v;
        graph.append(-u + V, v + V);
        graph.append(-v + V, u + V);
    }
    graph.build_scc();
    vector<int> last_component(2 * V + 1, -1);
    bool is_answer = true;
    for (int i = 0; i < graph.components.size(); i++) {
        for (int node : graph.components[i]) {
            int negation = -(node - V) + V;
            if (last_component[negation] == i) is_answer = false;
            last_component[node] = i;
        }
    }
    if (is_answer) {
        vector<int> result(V);
        for (int i = 1; i <= V; i++) {
            int val = i + V;
            int negation = -i + V;
            result[i - 1] = graph.component_number[val] > graph.component_number[
                negation];
        }
        for (int val : result) cout << val << " ";
        cout << "\n";
    }
}

```

## 5.7 Union Find

```

struct UnionFind {
    int n;
    vector<int> u;

    UnionFind(int n) : n(n) {
        u.resize(n + 1);
        for (int i = 1; i <= n; i++) {
            u[i] = i;
        }
    }

    int find(int k) {
        if (u[u[k]] == u[k]) return u[k];
        else return u[k] = find(u[k]);
    }
}

```

```
void uni(int a, int b) {
    a = find(a);
    b = find(b);
    if(a < b) u[b] = a;
    else u[a] = b;
}
};
```

## 5.8 MST Kruskal

```
template <class T> struct MinimumSpanningTree {
    /*
        T: 가중치의타입

        n: 노드개수
        m: 간선개수
        result : MST 결과가중치 ( 합)
    */
    struct Edge {
        int u, v;
        T weight;

        Edge(int u, int v, T weight) : u(u), v(v), weight(weight) {}
        bool operator< (Edge other) const { return weight < other.weight; }
    };

    int n, m;
    vector<int> uf;
    vector<Edge> edges;
    vector<Edge> chosen_edges;

    T result; // 의MST 가중치합
    int cnt; // 뽑은간선수

    MinimumSpanningTree(int n, int m) : n(n), m(m) {
        uf.resize(n + 1);

        for(0, n + 1) {
            uf[i] = i;
        }
        result = 0;
        cnt = 0;
    }

    int find(int a) {
        /*
            Union-Find: Find 연산
        */
        if(uf[a] == a) return a;
        return uf[a] = find(uf[a]);
    }

    int merge(int a, int b) {
        /*
            Union-Find: Union합쳐진경우
        */
    }
};
```

```
        true 반환
    */

    a = find(a);
    b = find(b);

    if(a == b) return false;

    uf[b] = a;
    return true;
}

void add_edge(int u, int v, T cost) {
    edges.push_back(Edge(u, v, cost));
}

void run() {
    sort(edges.begin(), edges.end());

    for(int i = 0; i < edges.size(); i++) {
        if(merge(edges[i].u, edges[i].v)) {
            result += edges[i].weight;

            // chosen_edges.push_back(edges[i]);
            if(++cnt >= n - 1) break;
        }
    }
}
};
```

## 5.9 Lowest Common Ancestor

```
#define MAX_DEGREE 20

struct LCA {
    // root: 트리의루트설정 , n: 트리의노드개수
    // addEdge -> init -> query(O(Log(n)))

    ll root, n;
    llv1 depth;
    llv2 adj;
    llv2 parent; // n X MAX_DEGREE

    LCA(ll root, ll n) : root(root), n(n) {
        depth.resize(n + 1);
        adj.resize(n + 1);
        parent.resize(n + 1, llv1(MAX_DEGREE, 0));
    }

    void addEdge(ll a, ll b) {
        adj[a].push_back(b);
        adj[b].push_back(a);
    }

    void init() {
    }
};
```

```

dfs(root, 0, 1);

for(int i = 1; i < MAX_DEGREE; i++) {
    for(int j = 1; j <= n; j++) {
        parent[j][i] = parent[parent[j][i-1]][i-1];
    }
}

void dfs(int here, int par, int d) {
    depth[here] = d;
    parent[here][0] = par;

    for(int there : adj[here]) {
        if(depth[there] > 0) continue;

        dfs(there, here, d + 1);
    }
}

int query(int a, int b) {
    if(depth[a] > depth[b]) {
        swap(a, b);
    }

    for(int i = MAX_DEGREE - 1; i >= 0; i--) {
        if (depth[b] - depth[a] >= (1 << i)) {
            b = parent[b][i];
        }
    }

    if(a == b) {
        return a;
    }

    for(int i = MAX_DEGREE - 1; i >= 0; i--) {
        if(parent[a][i] != parent[b][i]) {
            a = parent[a][i];
            b = parent[b][i];
        }
    }

    return parent[a][0];
}
};

```

## 5.10 Maxflow dinic

```

// usage:
// MaxFlowDinic::init(n);
// MaxFlowDinic::add_edge(0, 1, 100, 100); // for bidirectional edge
// MaxFlowDinic::add_edge(1, 2, 100); // directional edge
// result = MaxFlowDinic::solve(0, 2); // source -> sink
// graph[i][edgeIndex].res -> residual
//

```

```

// in order to find out the minimum cut, use `l'.
// if l[i] == 0, i is unreachable.
//
// O(V*V*E)
// with unit capacities, O(min(V^(2/3), E^(1/2)) * E)
struct MaxFlowDinic {
    typedef int flow_t;
    struct Edge {
        int next;
        size_t inv; /* inverse edge index */
        flow_t res; /* residual */
    };
    int n;
    vector<vector<Edge>> graph;
    vector<int> q, l, start;

    void init(int _n) {
        n = _n;
        graph.resize(n);
        for (int i = 0; i < n; i++) graph[i].clear();
    }
    void add_edge(int s, int e, flow_t cap, flow_t caprev = 0) {
        Edge forward{ e, graph[e].size(), cap };
        Edge reverse{ s, graph[s].size(), caprev };
        graph[s].push_back(forward);
        graph[e].push_back(reverse);
    }
    bool assign_level(int source, int sink) {
        int t = 0;
        memset(&l[0], 0, sizeof(l[0]) * l.size());
        l[source] = 1;
        q[t++] = source;
        for (int h = 0; h < t && !l[sink]; h++) {
            int cur = q[h];
            for (const auto& e : graph[cur]) {
                if (l[e.next] || e.res == 0) continue;
                l[e.next] = l[cur] + 1;
                q[t++] = e.next;
            }
        }
        return l[sink] != 0;
    }
    flow_t block_flow(int cur, int sink, flow_t current) {
        if (cur == sink) return current;
        for (int& i = start[cur]; i < graph[cur].size(); i++) {
            auto& e = graph[cur][i];
            if (e.res == 0 || l[e.next] != l[cur] + 1) continue;
            if (flow_t res = block_flow(e.next, sink, min(e.res, current))) {
                e.res -= res;
                graph[e.next][e.inv].res += res;
                return res;
            }
        }
        return 0;
    }
};

```

```

flow_t solve(int source, int sink) {
    q.resize(n);
    l.resize(n);
    start.resize(n);
    flow_t ans = 0;
    while (assign_level(source, sink)) {
        memset(&start[0], 0, sizeof(start[0]) * n);
        while (flow_t flow = block_flow(source, sink, numeric_limits<flow_t>::max()))
            ans += flow;
    }
    return ans;
}
};

```

## 5.11 Maxflow Edmonds-Karp

```

struct MaxFlowEdgeDemands
{
    MaxFlowDinic mf;
    using flow_t = MaxFlowDinic::flow_t;

    vector<flow_t> ind, outd;
    flow_t D; int n;

    void init(int _n) {
        n = _n; D = 0; mf.init(n + 2);
        ind.clear(); outd.clear();
        ind.resize(n, 0); outd.resize(n, 0);
    }

    void add_edge(int s, int e, flow_t cap, flow_t demands = 0) {
        mf.add_edge(s, e, cap - demands);
        D += demands; ind[e] += demands; outd[s] += demands;
    }

    // returns { false, 0 } if infeasible
    // { true, maxflow } if feasible
    pair<bool, flow_t> solve(int source, int sink) {
        mf.add_edge(sink, source, numeric_limits<flow_t>::max());

        for (int i = 0; i < n; i++) {
            if (ind[i]) mf.add_edge(n, i, ind[i]);
            if (outd[i]) mf.add_edge(i, n + 1, outd[i]);
        }

        if (mf.solve(n, n + 1) != D) return{ false, 0 };

        for (int i = 0; i < n; i++) {
            if (ind[i]) mf.graph[i].pop_back();
            if (outd[i]) mf.graph[i].pop_back();
        }

        return{ true, mf.solve(source, sink) };
    }
}

```

```
};
```

## 5.12 MCMF SPFA

```

struct MCMF {
    struct Edge {
        ll to;
        ll capacity;
        ll cost;

        Edge* rev;
        Edge(ll to, ll capacity, ll cost) : to(to), capacity(capacity), cost(cost) {}
    };

    ll n;
    ll source, sink;
    vector<vector<Edge*>> graph;
    vector<bool> check;
    vector<ll> distance;
    vector<pair<ll, ll>> from;

    MCMF(ll n, ll source, ll sink): n(n), source(source), sink(sink) {

        // source: 시작점
        // sink: 도착점
        // n: 모델링한그래프의정점개수
        graph.resize(n + 1);
        check.resize(n + 1);
        from.resize(n + 1, make_pair(-1, -1));
        distance.resize(n + 1);
    };

    void addEdge(ll u, ll v, ll cap, ll cost) {
        Edge *ori = new Edge(v, cap, cost);
        Edge *rev = new Edge(u, 0, -cost);

        ori->rev = rev;
        rev->rev = ori;

        graph[u].push_back(ori);
        graph[v].push_back(rev);
    }

    void addEdgeFromSrc(ll v, ll cap, ll cost) {
        // 출발지점에서출발하는간선추가
        addEdge(source, v, cap, cost);
    }

    void addEdgeToSink(ll u, ll cap, ll cost) {
        // 도착지점으로가는간선추가
        addEdge(u, sink, cap, cost);
    }

    bool spfa(ll &total_flow, ll &total_cost) {

```

```
// spfa 기반의 MCMF

fill(check.begin(), check.end(), false);
fill(distance.begin(), distance.end(), INF);
fill(from.begin(), from.end(), make_pair(-1, -1));

distance[source] = 0;
queue<ll> q;
q.push(source);

while(!q.empty()) {
    ll x = q.front(); q.pop();

    check[x] = false;

    for(ll i = 0; i < graph[x].size(); i++) {
        Edge* e = graph[x][i];
        ll y = e->to;

        if(e->capacity > 0 && distance[x] + e->cost < distance[y]) {
            distance[y] = distance[x] + e->cost;
            from[y] = make_pair(x, i);

            if(!check[y]) {
                check[y] = true;
                q.push(y);
            }
        }
    }
}

if(distance[sink] == INF) return false;

// 간선들에서부터 sink 역추적하여경로를만든다 .
ll x = sink;
ll c = graph[from[x].first][from[x].second]->capacity;

while(from[x].first != -1) {
    if(c > graph[from[x].first][from[x].second]->capacity) {
        c = graph[from[x].first][from[x].second]->capacity;
    }
    x = from[x].first;
}

// 만든경로를따라유량을흘린다 .
x = sink;
while(from[x].first != -1) {
    Edge* e = graph[from[x].first][from[x].second];
    e->capacity -= c;
    e->rev->capacity += c;
    x = from[x].first;
}

total_flow += c;
```

```
total_cost += c * distance[sink];

return true;
}

pair<ll, ll> flow() {
    ll total_flow = 0;
    ll total_cost = 0;

    while(spfa(total_flow, total_cost));

    return make_pair(total_flow, total_cost);
}
};
```

## 5.13 MCMF

```
// precondition: there is no negative cycle.
// usage:
// MinCostFlow mcf(n);
// for(each edges) mcf.addEdge(from, to, cost, capacity);
// mcf.solve(source, sink); // min cost max flow
// mcf.solve(source, sink, 0); // min cost flow
// mcf.solve(source, sink, goal_flow); // min cost flow with total_flow >=
// goal_flow if possible
struct MinCostFlow {
    typedef int cap_t;
    typedef int cost_t;

    bool iszerocap(cap_t cap) { return cap == 0; }

    struct edge {
        int target;
        cost_t cost;
        cap_t residual_capacity;
        cap_t orig_capacity;
        size_t revid;
    };

    int n;
    vector<vector<edge>> graph;

    MinCostFlow(int n) : graph(n), n(n) {}

    void addEdge(int s, int e, cost_t cost, cap_t cap) {
        if (s == e) return;
        edge forward{ e, cost, cap, cap, graph[e].size() };
        edge backward{ s, -cost, 0, 0, graph[s].size() };
        graph[s].emplace_back(forward);
        graph[e].emplace_back(backward);
    }

    pair<cost_t, cap_t> augmentShortest(int s, int e, cap_t flow_limit) {
        auto infinite_cost = numeric_limits<cost_t>::max();
        auto infinite_flow = numeric_limits<cap_t>::max();
```



```

vector<pair<cost_t, cap_t>> dist(n, make_pair(infinite_cost, 0));
vector<int> from(n, -1), v(n);

dist[s] = pair<cost_t, cap_t>(0, infinite_flow);
queue<int> q;
v[s] = 1; q.push(s);
while(!q.empty()) {
    int cur = q.front();
    v[cur] = 0; q.pop();
    for (const auto& e : graph[cur]) {
        if (iszerocap(e.residual_capacity)) continue;
        auto next = e.target;
        auto ncost = dist[cur].first + e.cost;
        auto nflow = min(dist[cur].second, e.residual_capacity);
        if (dist[next].first > ncost) {
            dist[next] = make_pair(ncost, nflow);
            from[next] = e.revid;
            if (v[next]) continue;
            v[next] = 1; q.push(next);
        }
    }
}

auto p = e;
auto pathcost = dist[p].first;
auto flow = dist[p].second;
if (iszerocap(flow) || (flow_limit <= 0 && pathcost >= 0)) return pair<
    cost_t, cap_t>(0, 0);
if (flow_limit > 0) flow = min(flow, flow_limit);

while (from[p] != -1) {
    auto nedge = from[p];
    auto np = graph[p][nedge].target;
    auto fedge = graph[p][nedge].revid;
    graph[p][nedge].residual_capacity += flow;
    graph[np][fedge].residual_capacity -= flow;
    p = np;
}
return make_pair(pathcost * flow, flow);
}

pair<cost_t, cap_t> solve(int s, int e, cap_t flow_minimum = numeric_limits<
    cap_t>::max()) {
    cost_t total_cost = 0;
    cap_t total_flow = 0;
    for(;;) {
        auto res = augmentShortest(s, e, flow_minimum - total_flow);
        if (res.second <= 0) break;
        total_cost += res.first;
        total_flow += res.second;
    }
    return make_pair(total_cost, total_flow);
}
};

```

## 6 String

### 6.1 KMP

```

struct KMP {
    /*
     * s 문자열에서 문자열을 o 찾습니다. 매칭이 시작되는 인덱스 목록을 반환합니다
     *
     * Time: O(n + m)
     */
    vector<int> result;
    int MX;
    string s, o;
    int n, m; // n : s.Length(), m : o.Length();
    vector<int> fail;

    KMP(string s, string o) : s(s), o(o) {
        n = s.length();
        m = o.length();
        MX = max(n, m) + 1;
        fail.resize(MX, 0);

        run();
    }

    void run() {
        for(int i = 1, j = 0; i < m; i++){
            while(j > 0 && o[i] != o[j]) j = fail[j-1];
            if(o[i] == o[j]) fail[i] = ++j;
        }
        for(int i = 0, j = 0; i < n; i++) {
            while(j > 0 && s[i] != o[j]) {
                j = fail[j - 1];
            }
            if(s[i] == o[j]) {
                if(j == m - 1) {
                    // matching OK;
                    result.push_back(i - m + 1);
                    j = fail[j];
                }
                else {
                    j++;
                }
            }
        }
    }
};

```

### 6.2 Manacher

```

// Use space to insert space between each character
// To get even length palindromes!
// O(|str|)

vector<int> manacher(string &s) {

```

```

int n = s.size(), R = -1, p = -1;
vector<int> A(n);
for (int i = 0; i < n; i++) {
    if (i <= R) A[i] = min(A[2 * p - i], R - i);
    while (i - A[i] - 1 >= 0 && i + A[i] + 1 < n && s[i - A[i] - 1] == s[i + A[i]
        ] + 1))
        A[i]++;
    if (i + A[i] > R)
        R = i + A[i], p = i;
}
return A;
}

string space(string &s) {
    string t;
    for (char c : s) t += c, t += ' ';
    t.pop_back();
    return t;
}

int maxpalin(vector<int> &M, int i) {
    if (i % 2) return (M[i] + 1) / 2 * 2;
    return M[i] / 2 * 2 + 1;
}

```

## 6.3 Suffix Array

typedef char T;

*// calculates suffix array.*

*// O(n\*logn)*

```

vector<int> suffix_array(const vector<T>& in) {
    int n = (int)in.size(), c = 0;
    vector<int> temp(n), pos2bckt(n), bckt(n), bpos(n), out(n);
    for (int i = 0; i < n; i++) out[i] = i;
    sort(out.begin(), out.end(), [&](int a, int b) { return in[a] < in[b]; });
    for (int i = 0; i < n; i++) {
        bckt[i] = c;
        if (i + 1 == n || in[out[i]] != in[out[i + 1]]) c++;
    }
    for (int h = 1; h < n && c < n; h <= 1) {
        for (int i = 0; i < n; i++) pos2bckt[out[i]] = bckt[i];
        for (int i = n - 1; i >= 0; i--) bpos[bckt[i]] = i;
        for (int i = 0; i < n; i++)
            if (out[i] >= n - h) temp[bpos[bckt[i]]++] = out[i];
        for (int i = 0; i < n; i++)
            if (out[i] >= h) temp[bpos[pos2bckt[out[i] - h]]++] = out[i] - h;
        c = 0;
        for (int i = 0; i + 1 < n; i++) {
            int a = (bckt[i] != bckt[i + 1]) || (temp[i] >= n - h)
                || (pos2bckt[temp[i + 1] + h] != pos2bckt[temp[i] + h]);
            bckt[i] = c;
            c += a;
        }
        bckt[n - 1] = c++;
    }
}

```

```

        temp.swap(out);
    }
    return out;
}

```

*// calculates lcp array. it needs suffix array & original sequence.*

*// O(n)*

```

vector<int> lcp(const vector<T>& in, const vector<int>& sa) {
    int n = (int)in.size();
    if (n == 0) return vector<int>();
    vector<int> rank(n), height(n - 1);
    for (int i = 0; i < n; i++) rank[sa[i]] = i;
    for (int i = 0, h = 0; i < n; i++) {
        if (rank[i] == 0) continue;
        int j = sa[rank[i] - 1];
        while (i + h < n && j + h < n && in[i + h] == in[j + h]) h++;
        height[rank[i] - 1] = h;
        if (h > 0) h--;
    }
    return height;
}

```

## 6.4 2nd Suffix Array

```

struct SuffixComparator {
    const vector<int> &group;
    int t;
}

```

```

SuffixComparator(const vector<int> &_group, int _t) : group(_group), t(_t) {}
bool operator()(int a, int b) {
    if (group[a] != group[b]) return group[a] < group[b];
    return group[a + t] < group[b + t];
}
};

```

```

vector<int> getSuffixArr(const string &s) {
    int n = s.size();
    int t = 1;

    vector<int> group(n + 1);

    for (int i = 0; i < n; i++) group[i] = s[i];
    group[n] = -1;

    vector<int> perm(n);
    for (int i = 0; i < n; i++) perm[i] = i;

    while (t < n) {
        SuffixComparator compare(group, t);
        sort(perm.begin(), perm.end(), compare);
        t *= 2;
        if (t >= n)
            break;

        vector<int> new_group(n + 1);
    }
}

```

```

new_group[n] = -1;
new_group[perm[0]] = 0;
for (int i = 1; i < n; i++)
    if (compare(perm[i - 1], perm[i]))
        new_group[perm[i]] = new_group[perm[i - 1]] + 1;
    else
        new_group[perm[i]] = new_group[perm[i - 1]];
group = new_group;
}
return perm;
}

int getHeight(const string &s, vector<int> &pos) {
    // 최장중복부분문자열의길이
    const int n = pos.size();
    vector<int> rank(n);
    for (int i = 0; i < n; i++)
        rank[pos[i]] = i;
    int h = 0, ret = 0;
    for (int i = 0; i < n; i++) {
        if (rank[i] > 0) {
            int j = pos[rank[i] - 1];
            while (s[i + h] == s[j + h])
                h++;
            ret = max(ret, h);
            if (h > 0)
                h--;
        }
    }
    return ret;
}

```

## 6.5 Trie

```

int chToIdx(char ch) { return ch - 'a'; }
struct Trie {
    int terminal = -1;
    Trie *fail; // fail, 은 output 아호코라식에사용
    vector<int> output;
    Trie *chil[ALPHABETS];
    Trie() {
        for (int i = 0; i < ALPHABETS; i++)
            chil[i] = NULL;
    }
    ~Trie() {
        for (int i = 0; i < ALPHABETS; i++)
            if (chil[i])
                delete chil[i];
    }
    // number -> 문자열번호 (ith string)
    void insert(string &s, int number, int idx) {
        if (idx == s.size()) {
            terminal = number;
            return;
        }
    }
}

```

```

int next = chToIdx(s[idx]);
if (chil[next] == NULL)
    chil[next] = new Trie();
chil[next]->insert(s, number, idx + 1);
}

int find(string &s, int idx = 0) {
    if (idx == s.size())
        return terminal;
    int next = chToIdx(s[idx]);
    if (chil[next] == NULL)
        return false;
    return chil[next]->find(s, idx + 1);
}
};

```

## 6.6 Aho-Corasick

```

void computeFail(Trie *root) {
    queue<Trie *> q;
    root->fail = root;
    q.push(root);
    while (!q.empty()) {
        Trie *here = q.front();
        q.pop();
        for (int i = 0; i < ALPHABETS; i++) {
            Trie *child = here->chil[i];
            if (!child) continue;
            if (here == root) child->fail = root;
            else {
                Trie *t = here->fail;
                while (t != root && t->chil[i] == NULL) t = t->fail;
                if (t->chil[i]) t = t->chil[i];
                child->fail = t;
            }
            child->output = child->fail->output;
            if (child->terminal != -1) child->output.push_back(child->terminal);
            q.push(child);
        }
    }
}

vector<pair<int, int>> ahoCorasick(string &s, Trie *root) {
    vector<pair<int, int>> ret;
    Trie *state = root;
    for (int i = 0; i < s.size(); i++) {
        int idx = chToIdx(s[i]);
        while (state != root && state->chil[idx] == NULL) state = state->fail;
        if (state->chil[idx]) state = state->chil[idx];
        for (int j = 0; j < state->output.size(); j++) ret.push_back({i, state->output[j]});
    }
    return ret;
}

```

## 6.7 Z Algorithm

```
// Z[i] : maximum common prefix length of &s[0] and &s[i]
// O(|s|)
using seq_t = string;
vector<int> z_func(const seq_t &s) {
    vector<int> z(s.size());
    z[0] = s.size();
    int l = 0, r = 0;

    for (int i = 1; i < s.size(); i++) {
        if (i > r) {
            int j;
            for (j = 0; i + j < s.size() && s[i + j] == s[j]; j++) ;
            z[i] = j; l = i; r = i + j - 1;
        } else if (z[i - l] < r - i + 1) {
            z[i] = z[i - l];
        } else {
            int j;
            for (j = 1; r + j < s.size() && s[r + j] == s[r - i + j]; j++) ;
            z[i] = r - i + j; l = i; r += j - 1;
        }
    }

    return z;
}
```

## 7 Dynamic Programming

### 7.1 LIS

```
struct LIS {
    llv1 ar;

    llv1 v, buffer;
    llv1::iterator vv;
    vector<pair<ll, ll> > d;

    void perform() {
        v.pb(2000000000ll);

        ll n = sz(ar);

        for1(0, n){
            if (ar[i] > *v.rbegin()) {
                v.pb(ar[i]);
                d.push_back({ v.size() - 1, ar[i] });
            }
            else {
                vv = lower_bound(v.begin(), v.end(), ar[i]);
                *vv = ar[i];
                d.push_back({ vv - v.begin(), ar[i] });
            }
        }

        for(int i = sz(d) - 1; i > -1; i--){
```

```
            if(d[i].first == sz(v)-1){
                buffer.pb(d[i].second);
                v.pop_back();
            }

            reverse(buffer.begin(), buffer.end());
        }

        ll length() {
            return buffer.size();
        }

        llv1 result() {
            return buffer;
        }
    };
```

### 7.2 LIS only length

```
ll lis(llv1& ar) {
    llv1 v, buffer;
    llv1::iterator vv;
    v.pb(2000000000ll);

    ll n = sz(ar);

    for1(0, n){
        if(ar[i] > *v.rbegin()) {
            v.pb(ar[i]);
        }
        else{
            vv = lower_bound(v.begin(), v.end(), ar[i]);
            *vv = ar[i];
        }
    }
    return sz(v);
}
```

### 7.3 KnapSack

```
ll N, maxWeight, ans;
ll D[2][11000];
ll weight[110], cost[110];
void knapsack() {
    for (int x = 1; x <= N; x++) {
        for (int y = 0; y <= maxWeight; y++) {
            if (y >= weight[x]) {
                D[x % 2][y] = max(D[(x + 1) % 2][y], D[(x + 1) % 2][y - weight[x]] + cost[x]);
            } else {
                D[x % 2][y] = D[(x + 1) % 2][y];
            }
        }
        ans = max(ans, D[x % 2][y]);
    }
```

```

    }
}
void input() {
    cin >> N >> maxWeight;
    for (int x = 1; x <= N; x++) {
        cin >> weight[x] >> cost[x];
    }
}
}

```

## 7.4 Coin Change

```

// 경우의수
ll CC(ll v1 &coin, ll money, ll MX) {
    ll D[MX];
    fill(D, D + MX, 0);
    D[0] = 1;
    for (int i = coin.size() - 1; i >= 0; i--) {
        for (int j = coin[i]; j <= money; j++) {
            D[j] += D[j - coin[i]];
            D[j] %= MOD;
        }
    }
    return D[money] % MOD;
}

```

## 7.5 Bit Field DP

```

#define MOD 9901;

int dp[1 << 14 + 1][200];
int n, m;

int solve(int pos, int check, int dep) {
    if (dp[check][pos] != 0) return dp[check][pos];
    int &ret = dp[check][pos];
    if (dep == n * m) return ret = 1;
    if ((check & 1)) return ret = solve(pos - 1, check >> 1, dep) % MOD;

    int sum = 0;
    if (!(check & 1) && (pos - 1) / m > 0)
        sum += solve(pos - 1, (check >> 1) | (1 << (m - 1)), dep + 2) % MOD;
    if (!(check & 1) && pos % m != 1 && !(check & 2) && pos >= 2 && m > 1)
        sum += solve(pos - 2, check >> 2, dep + 2) % MOD;
    // cout<<pos<<" "<<check<<" "<<dep<<" "<<sum<<endl;

    return ret = sum % MOD;
}
int main() {
    cin >> n >> m;

    if (n * m % 2 == 1)
        cout << 0;
    else

```

```

        cout << solve(n * m, 0, 0) % MOD;

    return 0;
}

```

## 7.6 Knuth Optimization

```

int solve(int n) {
    for (int m = 2; m <= n; m++) {
        for (int i = 0; m + i <= n; i++) {
            int j = i + m;
            for (int k = K[i][j - 1]; k <= K[i + 1][j]; k++) {
                int now = dp[i][k] + dp[k][j] + sum[j] - sum[i];
                if (dp[i][j] > now)
                    dp[i][j] = now, K[i][j] = k;
            }
        }
    }
    return dp[0][n];
}

int main() {
    int n;
    cin >> n;
    fill(&dp[0][0], &dp[MAX-1][MAX-1], INF);
    for (int i = 1; i <= n; i++){
        cin >> arr[i];
        sum[i] = sum[i - 1] + arr[i];
        K[i - 1][i] = i;
        dp[i - 1][i] = 0;
    }
    cout << solve(n) << "\n";
}

/*
if
C[a][c] + C[b][d] <= C[a][d] + C[b][c] (a<=b<=c<=d)
C[b][c] <= C[a][d] (a<=b<=c<=d)

then
dp[i][j] = min(dp[i][k] + dp[k][j]) + C[i][j]
range of k: A[i, j-1] <= A[i][j]=k <= A[i+1][j]
*/

```