Misc

Template

```cpp
#include <bits/stdc++.h>
#define all(x) begin(x),end(x)
using namespace std;
using ll = long long;

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
}
```

Optimization Pragma

```cpp
#pragma GCC optimize("Ofast")
```

Compilation Script

```bash
#!/bin/bash
g++ --std=c++17 -Wall -Wshadow -Wno-conversion -ftrapv -g $1 -o ${1%.cpp}.bin
```

Run Script

Usage: `./run.sh path/to/sample/folder ./solution.bin`

```bash
#!/bin/bash
folder=$1;shift
for f in $folder/*.in; do
    echo $f
    pre=${f%.in}
    out=$pre.out
    $* < $f > $out
    diff $out $pre.ans
```

Geometry

Geometry Template

```python
def vecsub(a, b):
    return (a[0] - b[0], a[1] - b[1])
def vecadd(a, b):
    return (a[0] + b[0], a[1] + b[1])
def dot(a, b):
    return a[0] * b[0] + a[1] * b[1]
def cross(a, b, o = (0, 0)):
    return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])
def len2(a):
    return a[0] ** 2 + a[1] ** 2
def dist2(a, b):
    return len2(vecsub(a, b))
def sign(x):
    return (x > 0) - (x < 0)
def zero(x):
    return abs(x) < 1E-9
```

Distance between point and line segment

Returns the distance from the point p to the line segment starting at s and ending at e.

```python
def distPS(s, e, p):
    if s == e:
        return sqrt(dist2(p, s))
    se, sp = vecsub(e, s), vecsub(p, s)
    d = len2(se)
    t = min(d, max(0, dot(vecsub(p, s), vecsub(e, s))))
    return sqrt(dist2((sp[0] * d, sp[1] * d), (se[0] * t, se[1] * t))) / d
```

Distance between point and line

Returns the signed distance from the point p to the line passing through the points a and b.

```
def distPL(a, b, p):
    return cross(b, p, a) / sqrt(dist2(a, b))
```

Check if point is on line segment

```
def onSegment(s, e, p):
    # return zero(distPS(s, e, p))   if floating-point is OK
    return cross(s, e, p) == 0 and dot(vecsub(s, p), vecsub(e, p)) <= 0
```

Project point to line (or reflect)

Projects the point p onto the line passing through a and b.
Set `refl=True` to get reflection of point p across the line instead.

```
def projPL(a, b, p, refl = False):
    v = vecsub(b, a)
    s = (1 + refl) * cross(b, p, a) / len2(v)
    return (p[0] + v[1] * s, p[1] - v[0] * s)
```

Intersection between two lines

If a unique intersection point of the lines going through `s1,e1` and `s2,e2` exists `(1,point)` is returned.
If no intersection point exists `(0,(0,0))` is returned and if infinitely many exist `(-1,(0,0))` is returned.

```
def intersectLL(s1, e1, s2, e2):
    d = cross(vecsub(e1, s1), vecsub(e2, s2))
    if zero(d): # parallel
        return (-zero(cross(e1, s2, s1)), (0, 0))
    p, q = cross(e1, e2, s2), cross(e2, s1, s2)
    return (1, ((s1[0] * p + e1[0] * q) / d, (s1[1] * p + e1[1] * q) / d))
```

Intersection between two line segments

If a unique intersection is found, returns a list with only this point. If the segments intersect in many points, returns a list of 2 elements containing the start and end of the common line segment. If no intersection, returns an empty list

```
def intersectSS(s1, e1, s2, e2):
    oa, ob, oc, od = cross(e2, s1, s2), cross(e2, e1, s2), cross(e1, s2, s1), cross(e1, e2, s1)
    if sign(oa) * sign(ob) < 0 and sign(oc) * sign(od) < 0:
        div = ob - oa
        return [( (s1[0] * ob - e1[0] * oa) / div, (s1[1] * ob - e1[1] * oa) / div )]
    s = set()
    if onSegment(s2, e2, s1):
        s.add(s1)
    if onSegment(s2, e2, e1):
        s.add(e1)
    if onSegment(s1, e1, s2):
        s.add(s2)
    if onSegment(s1, e1, e2):
        s.add(e2)
    return list(s)
```

Point inside polygon

Returns true if the point `pt` lies within the polygon `poly`. If strict is true, returns false for points on the boundary.

```
def pointInPolygon(poly, pt, strict = True):
    c = False
    for i in range(len(poly)):
        q = poly[i - 1]
        if onSegment(q, poly[i], pt):
            return not strict
        c ^= ((pt[1] < q[1]) - (pt[1] < poly[i][1])) * cross(q, poly[i], pt) > 0
    return c
```

Polygon area

Returns twice the signed area of a polygon. Clockwise enumeration gives negative area.

```python
def polygonArea2(v):
    return sum(map(lambda i: cross(v[i - 1], v[i]), range(len(v))))
```

Intersection between two circles

Computes the pair of points at which two circles intersect. Returns `None` in case of no intersection.

```python
def intersectCC(c1, c2, r1, r2):
    if c1 == c2:
        assert(r1 != r2)
        return None
    vec = vecsub(c2, c1)
    d2, sm, dif = len2(vec), r1 + r2, r1 - r2
    if sm ** 2 < d2 or dif ** 2 > d2:
        return None
    p = (d2 + r1 ** 2 - r2 ** 2) / (d2 * 2)
    h2 = r1 ** 2 - p * p * d2
    mid = (c1[0] + vec[0] * p, c1[1] + vec[1] * p)
    plen = sqrt(max(0, h2) / d2)
    per = (-vec[1] * plen, vec[0] * plen)
    return (vecadd(mid, per), vecsub(mid, per))
```

Convex hull (python)

Returns a list of points on the convex hull in counter-clockwise order. Points on the edge of the hull between two other points are not considered part of the hull. Time complexity: $\mathcal{O}(n \log n)$

```python
def convexHull(pts):
    if len(pts) <= 1:
        return pts
    pts.sort()
    t, s, h = 0, 0, [0] * (len(pts) + 1)
    for i in range(2):
        for p in pts:
            while t >= s + 2 and cross(h[t - 1], p, h[t - 2]) <= 0:
                t -= 1
            h[t], t = p, t + 1
        s = t = t - 1
        pts.reverse()
    return h[:t - (t == 2 and h[0] == h[1])]
```

Convex hull (C++)

```cpp
using Point = pair<ll, ll>;
ll cross(Point a, Point b, Point c) {
    return (a.first - c.first) * (b.second - c.second) -
           (b.first - c.first) * (a.second - c.second);
}
vector<Point> convexHull(vector<Point> pts) {
    if (pts.size() <= 1) return pts;
    sort(all(pts));
    vector<Point> h(pts.size() + 1);
    ll t = 0, s = 0;
    for (ll i = 0; i < 2; i++) {
        for (Point p : pts) {
            while (t >= s + 2 && cross(h[t - 1], p, h[t - 2]) <= 0)
                t--;
            h[t++] = p;
        }
        s = --t;
        reverse(all(pts));
    }
    h.erase(h.begin() + t - (t == 2 && h[0] == h[1]), h.end());
    return h;
}
```

Data Structures

Segment Tree

```
struct SegTree {
    using T = ll; // use pair of value and index to get index from queries
    T f(T a, T b) { return a + b; }
    static constexpr T UNIT = 0; // neutral value for f

    vector<T> s; ll n;
    SegTree(ll len) : s(2 * len, UNIT), n(len) {}
    void set(ll pos, T val) {
        for (s[pos += n] = val; pos /= 2;)
            s[pos] = f(s[pos * 2], s[pos * 2 + 1]);
    }
    T query(ll lo, ll hi) { // query lo to hi (hi not included)
        T ra = UNIT, rb = UNIT;
        for (lo += n, hi += n; lo < hi; lo /= 2, hi /= 2) {
            if (lo % 2) ra = f(ra, s[lo++]);
            if (hi % 2) rb = f(s[--hi], rb);
        }
        return f(ra, rb);
    }
};
```

Fenwick Tree

```
struct FenwickTree {
    FenwickTree(ll n) : v(n + 1, 0) { }
    ll lsb(ll x) { return x & (-x); }
    ll prefixSum(ll n) { //sum of the first n items (nth not included)
        ll sum = 0;
        for (; n; n -= lsb(n))
            sum += v[n];
        return sum;
    }
    void adjust(ll i, ll delta) {
        for (i++; i < v.size(); i += lsb(i))
            v[i] += delta;
    }
    vector<ll> v;
};
```

Sparse Table

```
struct SparseTable {
    using T = ll;
    ll node(ll l, ll i) { return i + l * n; }
    ll n; vector<T> v;
    SparseTable(vector<T> values) : n(values.size()), v(move(values)) {
        ll d = log2(n);
        v.resize((d + 1) * n);
        for (ll L = 0, s = 1; L < d; L++, s *= 2) {
            for (ll i = 0; i < n; i++) {
                v[node(L + 1, i)] = min(v[node(L, i)], v[node(L, min(i + s, n - 1))]);
            }
        }
    }
    T query(ll lo, ll hi) { assert(hi > lo);
        ll l = (ll)log2(hi - lo);
        return min(v[node(l, lo)], v[node(l, hi - (1 << l))]);
    }
};
```

Lazy Segment Tree

Segment tree with support for range updates. Use T = pair of value and index to get index from queries.
All ranges are `(lo, hi]` (hi is not included). `fQuery` defines the function to be used for queries (currently min) and `fUpdate` defines the function to be used for updates (currently addition).

```
struct LazyST {
    using T = ll;

    T f(T a, T b) { return min(a, b); }
    static const T QUERY_UNIT = LLONG_MAX; // neutral value for f

    struct Node {
        T val = QUERY_UNIT; // current value of this segment
        optional<T> p; // value being pushed down into this segment
    };
    int len; vector<Node> nodes;

    LazyST(int l) : len(pow(2, ceil(log2(l)))), nodes(len * 2) { }
    void update(int lo, int hi, T val) { u(lo, hi, val, 1, 0, len); }
    T query(int lo, int hi) { return q(lo, hi, 1, 0, len); }

private:
    #define LST_NEXT int l = n * 2; int r = l + 1; int mid = (nlo + nhi) / 2
    void push(int n, int nlo, int nhi) {
        if (!nodes[n].p) return;
        LST_NEXT;
        u(nlo, nhi, *nodes[n].p, l, nlo, mid);
        u(nlo, nhi, *nodes[n].p, r, mid, nhi);
        nodes[n].p = {};
    }
    void u(int qlo, int qhi, T val, int n, int nlo, int nhi) {
        if (nhi <= qlo || nlo >= qhi) return;
        if (nlo >= qlo && nhi <= qhi) {
            //for interval set:
            nodes[n].p = val;
            nodes[n].val = val; // val * (nhi - nlo) for sum queries
            //for interval add:
            nodes[n].p = nodes[n].p.get_or(0) + val;
            nodes[n].val += val; // val * (nhi - nlo) for sum queries
        } else {
            push(n, nlo, nhi); LST_NEXT;
            u(qlo, qhi, val, l, nlo, mid);
            u(qlo, qhi, val, r, mid, nhi);
            nodes[n].val = f(nodes[l].val, nodes[r].val);
        }
    }
    T q(int qlo, int qhi, int n, int nlo, int nhi) {
        if (nhi <= qlo || nlo >= qhi) return QUERY_UNIT;
        if (nlo >= qlo && nhi <= qhi) return nodes[n].val;
        push(n, nlo, nhi); LST_NEXT;
        return f(q(qlo, qhi, l, nlo, mid), q(qlo, qhi, r, mid, nhi));
    }
};
```

Line Container

Container where you can add lines of the form $kx + m$, and query maximum values at points x. All operations are $\mathcal{O}(\log(n))$. For doubles, use `inf = 1/.0` and `div(a,b) = a/b`

```
struct Line {
    mutable ll k, m, p;
    bool operator<(const Line& o) const { return k < o.k; }
    bool operator<(ll x) const { return p < x; }
};
struct LineContainer : multiset<Line, less<>> {
    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()) { x->p = inf; return false; }
```

```
        if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
        else x->p = div(y->m - x->m, x->k - y->k);
        return x->p >= y->p;
    }
    void add(ll k, ll m) {
        auto z = insert({k, m, 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.k * x + l.m;
    }
};
```

Treap

```
struct Treap {
    Treap *l = 0, *r = 0;
    int val, y, c = 1;
    Treap(int v) : val(v), y(rand()) { }
};
int trCount(Treap* n) { // returns the number of nodes in treap n
    return n ? n->c : 0;
}
void trRecount(Treap* n) {
    n->c = trCount(n->l) + trCount(n->r) + 1;
}
Treap* trAt(Treap* n, int idx) { // returns the treap node at the specified index
    if (!n || idx == trCount(n->l)) return n;
    if (idx > trCount(n->l))
        return trAt(n->r, idx - trCount(n->l) - 1);
    return trAt(n->l, idx);
}
template<class F> void trForeach(Treap* n, F f) { // invokes f for every item in the treap n
    if (n) { trForeach(n->l, f); f(n->val); trForeach(n->r, f); }
}
pair<Treap*, Treap*> trSplit(Treap* n, int k) { // splits the treap n on index (or value) k
    if (!n) return {};
    if (trCount(n->l) >= k) { // use "if (n->val >= k) {" to split on value instead of index
        auto pa = trSplit(n->l, k);
        n->l = pa.second;
        trRecount(n);
        return {pa.first, n};
    } else {
        // use "auto pa = trSplit(n->r, k);" to split on value instead of index
        auto pa = trSplit(n->r, k - trCount(n->l) - 1);
        n->r = pa.first;
        trRecount(n);
        return {n, pa.second};
    }
}
Treap* trJoin(Treap* l, Treap* r) {
    if (!l) return r;
    if (!r) return l;
    if (l->y > r->y) {
        l->r = trJoin(l->r, r);
        trRecount(l);
        return l;
    } else {
        r->l = trJoin(l, r->l);
        trRecount(r);
        return r;
    }
}
// inserts the treap n into t at index pos (or value pos, depending on implementation of trSplit)
Treap* trInsert(Treap* t, Treap* n, int pos) {
    auto pa = trSplit(t, pos);
    return trJoin(trJoin(pa.first, n), pa.second);
}
```

Graph Algorithms

2-SAT

Calculates a valid assignment to boolean variables in a 2-SAT problem. Negated variables are represented by bit-inversions (~x). Time complexity: $\mathcal{O}(N + E)$, where N is the number of boolean variables, and E is the number of clauses.

```
struct TwoSat {
    int N;
    vector<vector<int>> gr;
    vector<int> values; // 0 = false, 1 = true
    TwoSat(int n = 0) : N(n), gr(2 * n) {}
    void either(int f, int j) {
        f = max(2 * f, -1-2*f);
        j = max(2 * j, -1-2*j);
        gr[f].push_back(j ^ 1);
        gr[j].push_back(f ^ 1);
    }
    void set_value(int x) { either(x, x); }
    vector<int> val, comp, z; int time = 0;
    int dfs(int i) {
        int low = val[i] = ++time, x;
        z.push_back(i);
        for(auto& e : gr[i])
            if (!comp[e])
                low = min(low, val[e] ?: dfs(e));
        if (low == val[i]) do {
            x = z.back(); z.pop_back();
            comp[x] = low;
            if (values[x>>1] == -1)
                values[x>>1] = x&1;
        } while (x != i);
        return val[i] = low;
    }
    bool solve() {
        values.assign(N, -1);
        val.assign(2 * N, 0); comp = val;
        for (int i = 0; i < 2 * N; ++i)
            if (!comp[i])
                dfs(i);
        for (int i = 0; i < N; ++i)
            if (comp[2 * i] == comp[2 * i + 1])
                return 0;
        return 1;
    }

    /* optional */ int add_var() {
        gr.emplace_back();
        gr.emplace_back();
        return N++;
    }
    /* optional */ void at_most_one(const vector<int>& li) {
        if (li.size() <= 1) return;
        int cur = ~li[0];
        for(size_t i = 2; i < li.size(); i++) {
            int next = add_var();
            either(cur, ~li[i]);
            either(cur, next);
            either(~li[i], next);
            cur = ~next;
        }
        either(cur, ~li[1]);
    }
};
```

Usage example:

```
TwoSat ts(number of boolean variables);
ts.either(0, ~3); // Var 0 is true or var 3 is false
ts.set_value(2); // Var 2 is true
ts.at_most_one({0,~1,2}); // <= 1 of vars 0, ~1 and 2 are true
ts.solve(); // Returns true iff it is solvable. ts.values holds the assigned values to the variables
```

Dijkstra's Algorithm

```
template<typename T> vector<T> dijkstra(ll start, const vector<vector<pair<ll, T>>>& adj) {
    vector<T> dist(adj.size(), numeric_limits<T>::max());
    dist[start] = 0;
    priority_queue<pair<T, ll>, vector<pair<T, ll>>, greater<>> pq;
    pq.emplace(dist[start], start);
    while (!pq.empty()) {
        ll cn = pq.top().second; pq.pop();
        for (auto [n, d] : adj[cn]) {
            T nd = dist[cn] + d;
            if (nd < dist[n]) pq.emplace(dist[n] = nd, n);
        }
    }
    return dist;
}
```

Floyd Warshall

Calculates all-pairs shortest path in a directed graph. Input is a distance matrix m, where `m[i][j]=inf` if i and j are not adjacent. As output, `m[i][j]` is set to the shortest distance between i and j, `inf` if no path, or `-inf` if the path goes through a negative-weight cycle. Time complexity: $\mathcal{O}(N^3)$.

```
const ll inf = 1LL << 62;
void floydWarshall(vector<vector<ll>>& m) {
    int n = m.size();
    for(int i = 0; i < n; i++)
        m[i][i] = min(m[i][i], 0LL);
    for(int k = 0; k < n; k++)
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                if (m[i][k] != inf && m[k][j] != inf)
                    m[i][j] = min(m[i][j], max(m[i][k] + m[k][j], -inf));
    //only needed if weights can be negative:
    for(int k = 0; k < n; k++)
        if (m[k][k] < 0)
            for(int i = 0; i < n; i++)
                for(int j = 0; j < n; j++)
                    if (m[i][k] != inf && m[k][j] != inf)
                        m[i][j] = -inf;
}
```

Bellman Ford

Calculates shortest paths from s in a graph that might have negative edge weights. Unreachable nodes get `dist = inf`; nodes reachable through negative-weight cycles get `dist = -inf`. Assumes $V^2 \max |w_i| < 2^{63}$. Time complexity: $\mathcal{O}(VE)$

```
const ll inf = 1LL << 62;
struct Ed {
    int a, b, w;
    int s() { return a < b ? a : -a; }
};
struct Node { ll dist = inf; int prev = -1; };
void bellmanFord(vector<Node>& nodes, vector<Ed>& eds, int s) {
    nodes[s].dist = 0;
    sort(all(eds), [] (Ed a, Ed b) { return a.s() < b.s(); });
    int lim = nodes.size() / 2 + 2;
    for(int i = 0; i < lim; i++)
        for(auto& ed : eds) {
            Node cur = nodes[ed.a], &dest = nodes[ed.b];
            if (abs(cur.dist) == inf) continue;
            ll d = cur.dist + ed.w;
            if (d < dest.dist) {
                dest.prev = ed.a;
                dest.dist = (i < lim - 1 ? d : -inf);
            }
        }
    for(int i = 0; i < lim; i++)
        for(auto& e : eds)
            if (nodes[e.a].dist == -inf)
                nodes[e.b].dist = -inf;
}
```

Strongly Connected Components

Finds strongly connected components in a directed graph. If vertices u, v belong to the same component, we can reach u from v and vice versa. Time complexity: $\mathcal{O}(E+V)$

Usage: `scc(graph, [&](vector<int>& v) { ... })}` visits all components in reverse topological order. `comp[i]` holds the component index of a node (a component only has edges to components with lower index). ncomps will contain the number of components.

```
vector<int> val, comp, z, cont;
int Time, ncomps;
template<class G, class F> int dfs(int j, G& g, F& f) {
    int low = val[j] = ++Time, x; z.push_back(j);
    for(auto& e : g[j]) if (comp[e] < 0)
        low = min(low, val[e] ?: dfs(e,g,f));
    if (low == val[j]) {
        do {
            x = z.back(); z.pop_back();
            comp[x] = ncomps;
            cont.push_back(x);
        } while (x != j);
        f(cont); cont.clear();
        ncomps++;
    }
    return val[j] = low;
}
template<class G, class F> void scc(G& g, F f) {
    val.assign(g.size(), 0);
    comp.assign(g.size(), -1);
    Time = ncomps = 0;
    for(size_t i = 0; i < g.size(); i++)
        if (comp[i] < 0) dfs(i, g, f);
}
```

Biconnected Components

Finds all biconnected components in an undirected graph, and runs a callback for the edges in each. In a biconnected component there are at least two distinct paths between any two nodes. Note that a node can be in several components. An edge which is not in a component is a bridge, i.e., not part of any cycle. Time complexity: $\mathcal{O}(E+V)$

```
vector<int> num, st;
vector<vector<pair<int, int>>> ed;
int Time;
template<class F> int dfs(int at, int par, F& f) {
    int me = num[at] = ++Time, top = me;
    for(auto& pa : ed[at]) {
        if (pa.second == par) continue;
        auto [y, e] = pa;
        if (num[y]) {
            top = min(top, num[y]);
            if (num[y] < me) st.push_back(e);
        } else {
            int si = st.size();
            int up = dfs(y, e, f);
            top = min(top, up);
            if (up == me) {
                st.push_back(e);
                f(vector<int>(st.begin() + si, st.end()));
                st.resize(si);
            }
            else if (up < me) st.push_back(e);
            else { /* e is a bridge */ }
        }
    }
    return top;
}
template<class F>
void bicomps(F f) {
    num.assign(ed.size(), 0);
    for(int i = 0; i < (int)ed.size(); i++)
        if (!num[i]) dfs(i, -1, f);
}
```

Usage example for biconnected components:

```
int eid = 0; ed.resize(N);
for each edge (a,b) {
    ed[a].emplace_back(b, eid);
    ed[b].emplace_back(a, eid++);
}
bicomps([&](const vi& edgelist) {...});
```

Maximum Flow (Dinic's Algorithm)

Constructor takes number of nodes, call `addEdge` to add edges and `calc` to find maximum flow. To obtain the actual flow, look at positive values of `Edge::cap` only.

Time complexity: $\mathcal{O}(VE\log U)$ where $U = \max|\text{cap}|$. $\mathcal{O}(\min(E^{1/2}, V^{2/3})E)$ if $U = 1$.
$\mathcal{O}(\sqrt{V}E)$ for bipartite matching.

```
struct Dinic {
    struct Edge { ll to, rev, cap, flow; };
    vector<vector<Edge>> adj;
    Dinic(ll n) : lvl(n), ptr(n), q(n), adj(n) {}
    void addEdge(ll a, ll b, ll cap, ll rcap = 0) {
        adj[a].push_back({b, adj[b].size(), cap, 0});
        adj[b].push_back({a, adj[a].size() - 1, rcap, 0});
    }
    ll calc(ll src, ll snk) {
        ll flow = 0; q[0] = src;
        for(ll L = 0; L < 31; L++) do {
            lvl = ptr = vector<ll>(q.size());
            ll qi = 0, qe = lvl[src] = 1;
            while (qi < qe && !lvl[snk]) {
                ll v = q[qi++];
                for(auto& e : adj[v])
                    if (!lvl[e.to] && (e.cap - e.flow) >> (30 - L))
                        q[qe++] = e.to, lvl[e.to] = lvl[v] + 1;
            }
            while (ll p = dfs(src, snk, LLONG_MAX)) flow += p;
        } while (lvl[snk]);
        return flow;
    }
    vector<ll> lvl, ptr, q;
    ll dfs(ll v, ll t, ll f) {
        if (v == t || !f) return f;
        for (ll& i = ptr[v]; i < adj[v].size(); i++) {
            Edge& e = adj[v][i];
            if (lvl[e.to] == lvl[v] + 1)
                if (ll p = dfs(e.to, t, min(f, e.cap - e.flow))) {
                    e.flow += p, adj[e.to][e.rev].flow -= p;
                    return p;
                }
        }
        return 0;
    }
};
```

Minimum Cost Maximum Flow

Calculates min-cost max-flow. `cap[i][j]!=cap[j][i]` is allowed; double edges are not. To obtain the actual flow, look at positive values only. Time complexity: Approximately $\mathcal{O}(E^2)$.

If costs can be negative, call `setpi` before `maxflow`, but note that negative cost cycles are not supported.

```
#include <bits/extc++.h>
const ll INF = LLONG_MAX / 4;
struct MCMF {
    int N;
    vector<vector<int>> ed, red;
    vector<vector<ll>> cap, flow, cost;
    vector<int> seen;
    vector<ll> dist, pi;
    vector<pair<int, int> > par;
    MCMF(int N) : N(N), ed(N), red(N), cap(N, vector<ll>(N)),
        flow(cap), cost(cap), seen(N), dist(N), pi(N), par(N) { }
```

```cpp
    void addEdge(int from, int to, ll cap, ll cost) {
        this->cap[from][to] = cap;
        this->cost[from][to] = cost;
        ed[from].push_back(to);
        red[to].push_back(from);
    }
    void path(int s) {
        fill(all(seen), 0);
        fill(all(dist), INF);
        dist[s] = 0; ll di;
        __gnu_pbds::priority_queue<pair<ll, int>> q;
        vector<decltype(q)::point_iterator> its(N);
        q.push({0, s});
        auto relax = [&](int i, ll cap, ll cost, int dir) {
            ll val = di - pi[i] + cost;
            if (cap && val < dist[i]) {
                dist[i] = val;
                par[i] = {s, dir};
                if (its[i] == q.end())
                    its[i] = q.push({-dist[i], i});
                else
                    q.modify(its[i], {-dist[i], i});
            }
        };
        while (!q.empty()) {
            s = q.top().second; q.pop();
            seen[s] = 1;
            di = dist[s] + pi[s];
            for (auto& i : ed[s]) if (!seen[i])
                relax(i, cap[s][i] - flow[s][i], cost[s][i], 1);
            for (auto& i : red[s]) if (!seen[i])
                relax(i, flow[i][s], -cost[i][s], 0);
        }
        for(int i = 0; i < N; i++)
            pi[i] = min(pi[i] + dist[i], INF);
    }
    pair<ll, ll> maxflow(int s, int t) {
        ll totflow = 0, totcost = 0;
        while (path(s), seen[t]) {
            ll fl = INF;
            for (int p,r,x = t; tie(p,r) = par[x], x != s; x = p)
                fl = min(fl, r ? cap[p][x] - flow[p][x] : flow[x][p]);
            totflow += fl;
            for (int p,r,x = t; tie(p,r) = par[x], x != s; x = p) {
                if (r) flow[p][x] += fl;
                else flow[x][p] -= fl;
            }
        }
        for(int i = 0; i < N; i++)
            for(int j = 0; j < N; j++)
                totcost += cost[i][j] * flow[i][j];
        return { totflow, totcost };
    }

    // optional, if some costs can be negative, call this before maxflow
    void setpi(int s) {
        fill(all(pi), INF); pi[s] = 0;
        int it = N, ch = 1; ll v;
        while (ch-- && it--)
            for(int i = 0; i < N; i++) if (pi[i] != INF)
                for(auto& to : ed[i]) if (cap[i][to])
                    if ((v = pi[i] + cost[i][to]) < pi[to])
                        pi[to] = v, ch = 1;
        assert(it >= 0); // negative cost cycle
    }
};
```

Minimum Cost Bipartite Matching

Cost matrix must be square! L and R are outputs describing the matching. Negate costs for max cost. Time complexity: $\mathcal{O}(n^3)$

```cpp
template <typename T>
T minCostMatching(const vector<vector<T>>& cost, vector<int>& L, vector<int>& R) {
    int n = cost.size(), mated = 0;
    vector<T> dist(n), u(n), v(n);
    vector<int> dad(n), seen(n);
    for(int i = 0; i < n; i++) {
        u[i] = cost[i][0];
        for(int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);
    }
    for(int j = 0; j < n; ++j) {
        v[j] = cost[0][j] - u[0];
        for(int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);
    }
    L = R = vector<int>(n, -1);
    for(int i = 0; i < n; i++) for(int j = 0; j < n; j++) {
        if (R[j] != -1) continue;
        if (fabs(cost[i][j] - u[i] - v[j]) < 1E-10) {
            L[i] = j; R[j] = i; mated++; break;
        }
    }
    for (; mated < n; mated++) {
        int s = 0;
        while (L[s] != -1) s++;
        fill(all(dad), -1); fill(all(seen), 0);
        for(int k = 0; k < n; k++)
            dist[k] = cost[s][k] - u[s] - v[k];
        int j = 0;
        while (true) {
            j = -1;
            for(int k = 0; k < n; k++){
                if (seen[k]) continue;
                if (j == -1 || dist[k] < dist[j]) j = k;
            }
            seen[j] = 1;
            int i = R[j];
            if (i == -1) break;
            for (int k = 0; k < n; k++) {
                if (seen[k]) continue;
                auto new_dist = dist[j] + cost[i][k] - u[i] - v[k];
                if (dist[k] > new_dist) {
                    dist[k] = new_dist;
                    dad[k] = j;
                }
            }
        }
        for (int k = 0; k < n; k++) {
            if (k == j || !seen[k]) continue;
            auto w = dist[k] - dist[j];
            v[k] += w, u[R[k]] -= w;
        }
        u[s] += dist[j];
        while (dad[j] >= 0) {
            int d = dad[j];
            R[j] = R[d];
            L[R[j]] = j;
            j = d;
        }
        R[j] = s; L[s] = j;
    }
    T value = 0;
    for (int i = 0; i < n; i++) value += cost[i][L[i]];
    return value;
}
```

Math

Solve Linear System of Equations

Solves $Ax = b$. If there are multiple solutions, an arbitrary one is returned. Returns rank, or -1 if no solutions. Time complexity: $\mathcal{O}(n^2 m)$

```
int solveLinear(vector<vector<double>> A, vector<double> b, vector<double>& x) {
    const double eps = 1e-12;
    int n = A.size(), m = x.size(), rank = 0, br, bc;
    if (n) assert((int)A[0].size() == m);
    vector<int> col(m); iota(all(col), 0);
    for(int i = 0; i < n; i++) {
        double v, bv = 0;
        for(int r = i; r < n; ++r) for(int c = i; c < m; c++)
            if ((v = fabs(A[r][c])) > bv)
                br = r, bc = c, bv = v;
        if (bv <= eps) {
            for(int j = i; j < n; j++)
                if (fabs(b[j]) > eps) return -1;
            break;
        }
        swap(A[i], A[br]);
        swap(b[i], b[br]);
        swap(col[i], col[bc]);
        for(int j = 0; j < n; j++)
            swap(A[j][i], A[j][bc]);
        bv = 1 / A[i][i];
        for(int j = i + 1; j < n; j++) {
            double fac = A[j][i] * bv;
            b[j] -= fac * b[i];
            for(int k = i+1; k < (m); ++k)
                A[j][k] -= fac*A[i][k];
        }
        rank++;
    }
    x.assign(m, 0);
    for (int i = rank; i--;) {
        b[i] /= A[i][i];
        x[col[i]] = b[i];
        for (int j = 0; j < i; j++)
            b[j] -= A[j][i] * b[i];
    }
    return rank;
}
```

Extended Euclidean Algorithm (python)

Finds the Greatest Common Divisor to the integers a and b. Also finds two integers x and y, such that $ax + by = \gcd(a, b)$. Returns a tuple of $(\gcd(a, b), x, y)$. If a and b are coprime, then x is the inverse of $a \pmod b$.

```
def extEuclid(a, b):
    if b:
        d, x, y = extEuclid(b, a % b)
        return (d, y, x - a // b * y)
    return (a, 1, 0)
```

Extended Euclidean Algorithm (C++)

```
ll extEuclid(ll a, ll b, ll& x, ll& y) {
    if (b) {
        ll d = extEuclid(b, a % b, y, x);
        return y -= a / b * x, d;
    }
    return x = 1, y = 0, a;
}
```

Chinese Remainder Theorem (python)

Finds the smallest number x satisfying a system of congruences, each in the form $x \equiv r_i \pmod{m_i}$. All pairs of m_i must be coprime. `eq` is a list of tuples describing the equations, the i:th of which should be (r_i, m_i).

```python
def crt(eq):
    p, res = 1, 0
    for rem, md in eq:
        p *= md
    for rem, md in eq:
        pp = p // md
        res = (res + rem * extEuclid(pp, md)[1] * pp) % p
    return res
```

Chinese Remainder Theorem (C++)

Similar to python version. Make sure that the product of all m_i is less than 2^{62}.

```cpp
ll crt(const vector<pair<ll, ll>>& eq) {
    ll p = 1, res = 0;
    for (auto e : eq) p *= e.second;
    for (auto e : eq) {
        ll pp = p / e.second, ppi, y;
        extEuclid(pp, e.second, ppi, y);
        res = (res + e.first * ppi * pp) % p;
    }
    return res;
}
```

Polynomial Roots

Finds the real roots of a polynomial. Time complexity: $\mathcal{O}(n^2 \log(1/\epsilon))$.

Usage (solves $x^2 - 3x + 2 = 0$): `poly_roots({{ 2, -3, 1 }},-1e9,1e9)`

```cpp
struct Poly {
    vector<double> a;
    double operator()(double x) const {
        double val = 0;
        for(int i = a.size(); i--;)
            (val *= x) += a[i];
        return val;
    }
    void diff() {
        for (size_t i = 1; i < a.size(); i++)
            a[i - 1] = i * a[i];
        a.pop_back();
    }
};
vector<double> poly_roots(Poly p, double xmin, double xmax) {
    if (p.a.size() == 2) return { -p.a[0] / p.a[1] };
    vector<double> ret;
    Poly der = p;
    der.diff();
    auto dr = poly_roots(der, xmin, xmax);
    dr.push_back(xmin - 1);
    dr.push_back(xmax + 1);
    sort(all(dr));
    for (size_t i = 0; i < dr.size() - 1; i++) {
        double l = dr[i], h = dr[i + 1];
        bool sign = p(l) > 0;
        if (sign ^ (p(h) > 0)) {
            for (int it = 0; it < 60; it++) {
                double m = (l + h) / 2, f = p(m);
                if ((f <= 0) ^ sign) l = m;
                else h = m;
            }
            ret.push_back((l + h) / 2);
        }
    }
    return ret;
}
```

Fast Fourier Transform

`fft(a)` computes $\hat{f}(k) = \sum_x a[x] \exp(2\pi i \cdot kx/N)$ for all k.
Useful for convolution: `conv(a, b)=c`, where $c[x] = \sum a[i]b[x-i]$.
Rounding is safe if $(\sum a_i^2 + \sum b_i^2) \log_2 N < 9 \cdot 10^{14}$ (in practice 10^{16}; higher for random inputs).
Time complexity: $\mathcal{O}(N \log N)$ with $N = |A| + |B|$ (about $1s$ for $N = 4 \cdot 10^6$)

```
typedef complex<double> C;
void fft(vector<C>& a) {
    int n = a.size(), L = 31 - __builtin_clz(n);
    static vector<complex<long double>> R(2, 1);
    static vector<C> rt(2, 1);  // (^ 10% faster if double)
    for (int k = 2; k < n; k *= 2) {
        R.resize(n); rt.resize(n);
        auto x = polar(1.0L, M_PIl / k);
        for (int i = k; i < 2 * k; i++)
            rt[i] = R[i] = i & 1 ? R[i / 2] * x : R[i / 2];
    }
    vector<int> rev(n);
    for (int i = 0; i < n; i++)
        rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
    for (int i = 0; i < n; i++)
        if (i < rev[i]) swap(a[i], a[rev[i]]);
    for (int k = 1; k < n; k *= 2)
        for (int i = 0; i < n; i += 2 * k)
            for(int j = 0; j < k; j++) {
                auto x = (double*)&rt[j + k], y = (double*)&a[i + j + k];
                C z(x[0] * y[0] - x[1] * y[1], x[0] * y[1] + x[1] * y[0]);
                a[i + j + k] = a[i + j] - z;
                a[i + j] += z;
            }
}
vector<double> conv(const vector<double>& a, const vector<double>& b) {
    if (a.empty() || b.empty()) return { };
    vector<double> res(a.size() + a.size() - 1);
    int L = 32 - __builtin_clz(res.size()), n = 1 << L;
    vector<C> in(n), out(n);
    copy(all(a), begin(in));
    for (size_t i = 0; i < a.size(); i++)
        in[i].imag(b[i]);
    fft(in);
    for (C& x : in) x *= x;
    for (int i = 0; i < n; i++)
        out[i] = in[-i & (n - 1)] - conj(in[i]);
    fft(out);
    for (size_t i = 0; i < res.size(); i++)
        res[i] = imag(out[i]) / (4 * n);
    return res;
}
```

Polynomial Hash

```
using lll = __int128_t;
ll P = 1223372036854778 9LL;
ll B = 260;
struct PolyHash {
    vector<ll> hashes, ex;
    PolyHash(const string& s) : hashes(s.size() + 1), ex(s.size() + 1) {
        hashes[0] = 1; ex[0] = 1; ex[1] = B;
        for (size_t i = 0; i < s.size(); i++) {
            hashes[i + 1] = ((hashes[i] * B) % P + s[i] + 1) % P;
            ex[i + 1] = (ex[i] * B) % P;
        }
    }
    ll hash(ll lo, ll hi) {
        return ((lll)hashes[hi] - (lll)hashes[lo] * (lll)ex[hi - lo] % P + P) % P;
    }
};
```