



BALLOON OVERFLOW NOTEBOOK

Lund University

Geometry

```

1 def vecsub(a, b):
2     return (a[0] - b[0], a[1] - b[1])
3 def vecadd(a, b):
4     return (a[0] + b[0], a[1] + b[1])
5 def dot(a, b):
6     return a[0] * b[0] + a[1] * b[1]
7 def cross(a, b):
8     return a[0] * b[1] - a[1] * b[0]
9 def cross(a, b, o):
10    return cross(vecsub(a, o), vecsub(b, o))
11 def len2(a):
12    return a[0] ** 2 + a[1] ** 2
13 def dist2(a, b):
14    return len2(vecsub(a, b))
15 def sign(x):
16    return (x > 0) - (x < 0)
17 def zero(x):
18    return abs(x) < 1E-9

```

Distance between point and line

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

```

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

```

Distance between point and line segment

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

```

1 def distPS(s, e, p):
2     if s == e:
3         return sqrt(dist2(p, s))
4     se, sp = vecsub(e, s), vecsub(p, s)
5     d = len2(se)
6     t = min(d, max(0, dot(vecsub(p, s), vecsub(e, s))))
7     return sqrt(dist2((sp[0] * d, sp[1] * d), (se[0] * t, se[1] * t))) / d

```

Check if point is on line segment

```

1 def onSegment(s, e, p):
2     # return zero(distPS(s, e, p)) if floating-point is OK
3     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.

```

1 def projPL(a, b, p, refl = False):
2     v = vecsub(b, a)
3     s = (1 + refl) * cross(b, p, a) / len2(v)
4     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.

```

1 def intersectLL(s1, e1, s2, e2):
2     d = cross(vecsub(e1, s1), vecsub(e2, s2))
3     if zero(d): # parallel
4         return (-zero(cross(e1, s2, s1)), (0, 0))
5     p, q = cross(e1, e2, s2), cross(e2, s1, s2)
6     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

```

1 def intersectSS(s1, e1, s2, e2):
2     oa, ob, oc, od = cross(e2, s1, s2), cross(e2, e1, s2), cross(e1, s2, s1), cross(e1, e2, s1)
3     if sign(oa) * sign(ob) < 0 and sign(oc) * sign(od) < 0:
4         div = ob - oa
5         return [(s1[0] * ob - e1[0] * oa) / div, (s1[1] * ob - e1[1] * oa) / div]
6     s = set()
7     if onSegment(s2, e2, s1):
8         s.add(s1)
9     if onSegment(s2, e2, e1):
10        s.add(e1)
11    if onSegment(s1, e1, s2):
12        s.add(s2)
13    if onSegment(s1, e1, e2):
14        s.add(e2)
15    return list(s)

```

Intersection between two circles

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

```

1 def intersectCC(c1, c2, r1, r2):
2     if c1 == c2:
3         assert(r1 != r2)
4         return None
5     vec = vecsub(c2, c1)
6     d2, sm, dif = len2(vec), r1 + r2, r1 - r2
7     if sm ** 2 < d2 or dif ** 2 > d2:
8         return None
9     p = (d2 + r1 ** 2 - r2 ** 2) / (d2 * 2)
10    h2 = r1 ** 2 - p * p * d2
11    mid = (c1[0] + vec[0] * p, c1[1] + vec[1] * p)
12    plen = sqrt(max(0, h2) / d2)
13    per = (-vec[1] * plen, vec[0] * plen)
14    return (vecadd(mid, per), vecsub(mid, per))

```

Polygon area

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

```

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

```

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.

```

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

```

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: $\mathcal{O}(n \log n)$

```

1 def convexHull(pts):
2     if len(pts) <= 1:
3         return pts
4     pts.sort()
5     t, s, h = 0, 0, [0] * (len(pts) + 1)
6     for i in range(2):
7         for p in pts:
8             while t >= s + 2 and cross(h[t - 1], p, h[t - 2]) <= 0:
9                 t -= 1
10            h[t], t = p, t + 1
11            s = t - 1
12        pts.reverse()
13    return h[:t - (t == 2 and h[0] == h[1])]

```

Convex hull (C++)

```

1 using Point = pair<ll, ll>;
2 ll cross(Point a, Point b, Point c) {
3     return (a.first - c.first) * (b.second - c.second) -
4           (b.first - c.first) * (a.second - c.second);
5 }
6 vector<Point> convexHull(vector<Point> pts) {
7     if (pts.size() <= 1) return pts;
8     sort(all(pts));
9     vector<Point> h(pts.size() + 1);
10    ll t = 0, s = 0;
11    for (ll i = 0; i < 2; i++) {
12        for (Point p : pts) {
13            while (t >= s + 2 && cross(h[t - 1], p, h[t - 2]) <= 0)
14                t--;
15            h[t++] = p;
16        }
17        s = --t;
18        reverse(all(pts));
19    }
20    h.erase(h.begin() + t - (t == 2 && h[0] == h[1]), h.end());
21    return h;
22 }

```

Data Structures

Segment Tree

```

1 struct SegTree {
2     using T = ll;
3     static constexpr T unit = 0;
4     T f(T a, T b) { return a + b; }
5     vector<T> s; ll n;
6     SegTree(ll n) : s(2*n, unit), n(n) {}
7     void set(ll pos, T val) {
8         for (s[pos += n] = val; pos /= 2;)
9             s[pos] = f(s[pos * 2], s[pos * 2 + 1]);
10    }
11    T query(ll lo, ll hi) { // query lo to hi (hi not included)
12        T ra = unit, rb = unit;
13        for (lo += n, hi += n; lo < hi; lo /= 2, hi /= 2) {
14            if (lo % 2) ra = f(ra, s[lo++]);
15            if (hi % 2) rb = f(s[--hi], rb);
16        }
17        return f(ra, rb);
18    }
19 };

```

Fenwick Tree

```

1 struct FenwickTree {
2     FenwickTree(ll n) : v(n + 1, 0) { }
3     ll lsb(ll x) { return x & (-x); }
4     ll prefixSum(ll n) { //sum of the first n items (nth not included)
5         ll sum = 0;
6         for (; n; n -= lsb(n))
7             sum += v[n];
8         return sum;
9     }
10    void adjust(ll i, ll delta) {
11        for (i++; i < v.size(); i += lsb(i))
12            v[i] += delta;
13    }
14    vector<ll> v;
15 };

```

Sparse Table

```

1 struct SparseTable {
2     using T = ll;
3     ll node(ll l, ll i) { return i + l * n; }
4     ll n; vector<T> v;
5     SparseTable(vector<T> values) : n(values.size()), v(move(values)) {
6         ll d = log2(n);
7         v.resize((d + 1) * n);
8         for (ll L = 0, s = 1; L < d; L++, s *= 2) {
9             for (ll i = 0; i < n; i++) {
10                 v[node(L + 1, i)] = min(v[node(L, i)], v[node(L, min(i + s, n - 1))]);
11             }
12         }
13     }
14     T query(ll lo, ll hi) { assert(hi > lo);
15         ll l = (ll)log2(hi - lo);
16         return min(v[node(l, lo)], v[node(l, hi - (1 << l))]);
17     }
18 };

```

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`

```

1 struct Line {
2     mutable ll k, m, p;
3     bool operator< (const Line& o) const { return k < o.k; }
4     bool operator< (ll x) const { return p < x; }
5 };
6 struct LineContainer : multiset<Line, less<>> {
7     const ll inf = LLONG_MAX;
8     ll div(ll a, ll b) { // floored division
9         return a / b - ((a ^ b) < 0 && a % b);
10    }
11    bool isect(iterator x, iterator y) {
12        if (y == end()) { x->p = inf; return false; }
13        if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
14        else x->p = div(y->m - x->m, x->k - y->k);
15        return x->p >= y->p;
16    }
17    void add(ll k, ll m) {
18        auto z = insert({k, m, 0}), y = z++, x = y;
19        while (isect(y, z)) z = erase(z);
20        if (x != begin() && isect(--x, y)) isect(x, y = erase(y));
21        while ((y = x) != begin() && (--x)->p >= y->p) isect(x, erase(y));
22    }
23    ll query(ll x) { assert(!empty());
24        auto l = *lower_bound(x);
25        return l.k * x + l.m;
26    }
27 };

```

Treap

```

1 struct Treap {
2     Treap *l = 0, *r = 0;
3     int y, c = 1;
4
5     int val;
6     Treap(int val) : y(rand()), val(val) { }
7 };
8
9 // returns the number of nodes in treap n
10 int trCount(Treap* n) {
11     return n ? n->c : 0;
12 }
13 void trRecount(Treap* n) {
14     n->c = trCount(n->l) + trCount(n->r) + 1;
15 }
16
17 // returns the treap node at the specified index
18 Treap* trAt(Treap* n, int idx) {
19     if (!n || idx == trCount(n->l)) return n;
20     if (idx > trCount(n->l))
21         return trAt(n->r, idx - trCount(n->l) - 1);
22     return trAt(n->l, idx);
23 }
24
25 // invokes f for every item in the treap n, ordered by index
26 template<class F> void trForeach(Treap* n, F f) {
27     if (n) { trForeach(n->l, f); f(n->val); trForeach(n->r, f); }
28 }
29
30 // splits the treap n on index k, returning the left and right treap respectively
31 pair<Treap*, Treap*> trSplit(Treap* n, int k) {
32     if (!n) return {};
33     if (trCount(n->l) >= k) { // use "if (n->val >= k) {" to split on value instead of index
34         auto pa = trSplit(n->l, k);
35         n->l = pa.second;
36         trRecount(n);
37         return {pa.first, n};
38     } else {
39         // use "auto pa = trSplit(n->r, k);" to split on value instead of index
40         auto pa = trSplit(n->r, k - trCount(n->l) - 1);
41         n->r = pa.first;
42         trRecount(n);
43         return {n, pa.second};
44     }
45 }
46
47 Treap* trJoin(Treap* l, Treap* r) {
48     if (!l) return r;
49     if (!r) return l;
50     if (l->y > r->y) {
51         l->r = trJoin(l->r, r);
52         trRecount(l);
53         return l;
54     } else {
55         r->l = trJoin(l, r->l);
56         trRecount(r);
57         return r;
58     }
59 }
60
61 // inserts the treap n into t at index pos (or value pos, depending on implementation of trSplit)
62 Treap* trInsert(Treap* t, Treap* n, int pos) {
63     auto pa = trSplit(t, pos);
64     return trJoin(trJoin(pa.first, n), pa.second);
65 }

```

Graph Algorithms

Floyd Warshall

Calculates all-pairs shortest path in a directed graph in $\mathcal{O}(N^3)$.

Input is an distance matrix m , where $m[i][j]=\text{inf}$ 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 $-\text{inf}$ if the path goes through a negative-weight cycle.

```

1  const ll inf = 1LL << 62;
2  void floydWarshall(vector<vector<ll>>& m) {
3      int n = m.size();
4      for(int i = 0; i < n; i++)
5          m[i][i] = min(m[i][i], 0LL);
6      for(int k = 0; k < n; k++)
7          for(int i = 0; i < n; i++)
8              for(int j = 0; j < n; j++)
9                  if (m[i][k] != inf && m[k][j] != inf)
10                     m[i][j] = min(m[i][j], max(m[i][k] + m[k][j], -inf));
11
12     //only needed if weights can be negative:
13     for(int k = 0; k < n; k++)
14         if (m[k][k] < 0)
15             for(int i = 0; i < n; i++)
16                 for(int j = 0; j < n; j++)
17                     if (m[i][k] != inf && m[k][j] != inf)
18                         m[i][j] = -inf;
19 }

```

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.

```

1  vector<int> val, comp, z, cont;
2  int Time, ncomps;
3  template<class G, class F> int dfs(int j, G& g, F& f) {
4      int low = val[j] = ++Time, x; z.push_back(j);
5      for(auto& e : g[j]) if (comp[e] < 0)
6          low = min(low, val[e] ? dfs(e, g, f));
7      if (low == val[j]) {
8          do {
9              x = z.back(); z.pop_back();
10             comp[x] = ncomps;
11             cont.push_back(x);
12         } while (x != j);
13         f(cont); cont.clear();
14         ncomps++;
15     }
16     return val[j] = low;
17 }
18 template<class G, class F> void scc(G& g, F f) {
19     int n = g.size();
20     val.assign(n, 0); comp.assign(n, -1);
21     Time = ncomps = 0;
22     for(int i = 0; i < n; i++)
23         if (comp[i] < 0) dfs(i, g, f);
24 }

```

2-SAT

Calculates a valid assignment to boolean variables a, b, c, \dots to 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.

```

1 struct TwoSat {
2     int N;
3     vector<vector<int>> gr;
4     vector<int> values; // 0 = false, 1 = true
5     TwoSat(int n = 0) : N(n), gr(2 * n) {}
6     void either(int f, int j) {
7         f = max(2 * f, -1-2*f);
8         j = max(2 * j, -1-2*j);
9         gr[f].push_back(j ^ 1);
10        gr[j].push_back(f ^ 1);
11    }
12    void set_value(int x) { either(x, x); }
13    vector<int> val, comp, z; int time = 0;
14    int dfs(int i) {
15        int low = val[i] = ++time, x;
16        z.push_back(i);
17        for(auto& e : gr[i])
18            if (!comp[e])
19                low = min(low, val[e] ? dfs(e));
20        if (low == val[i]) do {
21            x = z.back(); z.pop_back();
22            comp[x] = low;
23            if (values[x>>1] == -1)
24                values[x>>1] = x&1;
25        } while (x != i);
26        return val[i] = low;
27    }
28    bool solve() {
29        values.assign(N, -1);
30        val.assign(2 * N, 0); comp = val;
31        for (int i = 0; i < 2 * N; ++i)
32            if (!comp[i])
33                dfs(i);
34        for (int i = 0; i < N; ++i)
35            if (comp[2 * i] == comp[2 * i + 1])
36                return 0;
37        return 1;
38    }
39
40    /* optional */ int add_var() {
41        gr.emplace_back();
42        gr.emplace_back();
43        return N++;
44    }
45    /* optional */ void at_most_one(const vector<int>& li) {
46        if (li.size() <= 1) return;
47        int cur = ~li[0];
48        for(size_t i = 2; i < li.size(); i++) {
49            int next = add_var();
50            either(cur, ~li[i]);
51            either(cur, next);
52            either(~li[i], next);
53            cur = ~next;
54        }
55        either(cur, ~li[1]);
56    }
57 };

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

```


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: $\mathcal{O}(E + V)$

```

1 vector<int> num, st;
2 vector<vector<pair<int, int>>> ed;
3 int Time;
4 template<class F> int dfs(int at, int par, F& f) {
5     int me = num[at] = ++Time, e, y, top = me;
6     for(auto& pa : ed[at]) {
7         if (pa.second == par) continue;
8         tie(y, e) = pa;
9         if (num[y]) {
10             top = min(top, num[y]);
11             if (num[y] < me)
12                 st.push_back(e);
13         } else {
14             int si = st.size();
15             int up = dfs(y, e, f);
16             top = min(top, up);
17             if (up == me) {
18                 st.push_back(e);
19                 f(vector<int>(st.begin() + si, st.end()));
20                 st.resize(si);
21             }
22             else if (up < me) st.push_back(e);
23             else { /* e is a bridge */ }
24         }
25     }
26     return top;
27 }
28 template<class F>
29 void bicomps(F f) {
30     num.assign(ed.size(), 0);
31     for(int i = 0; i < (int)ed.size(); i++)
32         if (!num[i]) dfs(i, -1, f);
33 }

1 //Usage:
2 int eid = 0;
3 ed.resize(N);
4 for each edge (a,b) {
5     ed[a].emplace_back(b, eid);
6     ed[b].emplace_back(a, eid++);
7 }
8 bicomps([&](const vi& edgelist) {...});

```

Matching & Flow

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

```

1 struct Dinic {
2     struct Edge { ll to, rev, cap, flow; };
3     vector<vector<Edge>> adj;
4     Dinic(ll n) : lvl(n), ptr(n), q(n), adj(n) {}
5     void addEdge(ll a, ll b, ll cap, ll rcap = 0) {
6         adj[a].push_back({b, adj[b].size(), cap, 0});
7         adj[b].push_back({a, adj[a].size() - 1, rcap, 0});
8     }
9     ll calc(ll src, ll snk) {
10        ll flow = 0; q[0] = src;
11        for(ll L = 0; L < 31; L++) do {
12            lvl = ptr = vector<ll>(q.size());
13            ll qi = 0, qe = lvl[src] = 1;
14            while (qi < qe && !lvl[snk]) {
15                ll v = q[qi++];
16                for(auto& e : adj[v])
17                    if (!lvl[e.to] && (e.cap - e.flow) >> (30 - L))
18                        q[qe++] = e.to, lvl[e.to] = lvl[v] + 1;
19            }
20            while (ll p = dfs(src, snk, LLONG_MAX)) flow += p;
21        } while (lvl[snk]);
22        return flow;
23    }
24    vector<ll> lvl, ptr, q;
25    ll dfs(ll v, ll t, ll f) {
26        if (v == t || !f) return f;
27        for (ll& i = ptr[v]; i < adj[v].size(); i++) {
28            Edge& e = adj[v][i];
29            if (lvl[e.to] == lvl[v] + 1)
30                if (ll p = dfs(e.to, t, min(f, e.cap - e.flow))) {
31                    e.flow += p, adj[e.to][e.rev].flow -= p;
32                    return p;
33                }
34        }
35        return 0;
36    }
37 };

```

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. If costs can be negative, call `setpi` before `maxflow`, but note that negative cost cycles are not supported. Time complexity: Approximately $\mathcal{O}(E^2)$.

```

1 #include <bits/extc++.h>
2 const ll INF = LLONG_MAX / 4;
3 struct MCMF {
4     int N;
5     vector<vector<int>> ed, red;
6     vector<vector<ll>> cap, flow, cost;
7     vector<int> seen;
8     vector<ll> dist, pi;
9     vector<pair<int, int>> par;
10    MCMF(int N) :
11        N(N), ed(N), red(N), cap(N, vector<ll>(N)), flow(cap), cost(cap),
12        seen(N), dist(N), pi(N), par(N) {}
13
14    void addEdge(int from, int to, ll cap, ll cost) {
15        this->cap[from][to] = cap;
16        this->cost[from][to] = cost;

```

```

17         ed[from].push_back(to);
18         red[to].push_back(from);
19     }
20     void path(int s) {
21         fill(all(seen), 0);
22         fill(all(dist), INF);
23         dist[s] = 0; ll di;
24         __gnu_pbds::priority_queue<pair<ll, int>> q;
25         vector<decltype(q)::point_iterator> its(N);
26         q.push({0, s});
27         auto relax = [&](int i, ll cap, ll cost, int dir) {
28             ll val = di - pi[i] + cost;
29             if (cap && val < dist[i]) {
30                 dist[i] = val;
31                 par[i] = {s, dir};
32                 if (its[i] == q.end()) its[i] = q.push({-dist[i], i});
33                 else q.modify(its[i], {-dist[i], i});
34             }
35         };
36         while (!q.empty()) {
37             s = q.top().second; q.pop();
38             seen[s] = 1; di = dist[s] + pi[s];
39             for(auto& i : ed[s]) if (!seen[i])
40                 relax(i, cap[s][i] - flow[s][i], cost[s][i], 1);
41             for(auto& i : red[s]) if (!seen[i])
42                 relax(i, flow[i][s], -cost[i][s], 0);
43         }
44         for(int i = 0; i < N; i++)
45             pi[i] = min(pi[i] + dist[i], INF);
46     }
47     pair<ll, ll> maxflow(int s, int t) {
48         ll totflow = 0, totcost = 0;
49         while (path(s), seen[t]) {
50             ll fl = INF;
51             for (int p,r,x = t; tie(p,r) = par[x], x != s; x = p)
52                 fl = min(fl, r ? cap[p][x] - flow[p][x] : flow[x][p]);
53             totflow += fl;
54             for (int p,r,x = t; tie(p,r) = par[x], x != s; x = p)
55                 if (r) flow[p][x] += fl;
56                 else flow[x][p] -= fl;
57         }
58         for(int i = 0; i < N; i++)
59             for(int j = 0; j < N; j++)
60                 totcost += cost[i][j] * flow[i][j];
61         return {totflow, totcost};
62     }
63     // Optional, if some costs can be negative, call this before maxflow:
64     void setpi(int s) {
65         fill(all(pi), INF); pi[s] = 0;
66         int it = N, ch = 1; ll v;
67         while (ch-- && it--)
68             for(int i = 0; i < N; i++) if (pi[i] != INF)
69                 for(auto& to : ed[i]) if (cap[i][to])
70                     if ((v = pi[i] + cost[i][to]) < pi[to])
71                         pi[to] = v, ch = 1;
72         assert(it >= 0); // negative cost cycle
73     }
74 };

```

Minimum Cost Bipartite Matching

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

```

1 template <typename T>
2 T minCostMatching(const vector<vector<T>>& cost, vector<int>& L, vector<int>& R) {
3     int n = cost.size(), mated = 0;
4     vector<T> dist(n), u(n), v(n);
5     vector<int> dad(n), seen(n);
6     for(int i = 0; i < n; i++) {
7         u[i] = cost[i][0];
8         for(int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);
9     }
10    for(int j = 0; j < n; ++j) {
11        v[j] = cost[0][j] - u[0];
12        for(int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);
13    }
14    L = R = vector<int>(n, -1);
15    for(int i = 0; i < n; i++) for(int j = 0; j < n; j++) {
16        if (R[j] != -1) continue;
17        if (fabs(cost[i][j] - u[i] - v[j]) < 1E-10) {
18            L[i] = j; R[j] = i; mated++; break;
19        }
20    }
21    for (; mated < n; mated++) {
22        int s = 0;
23        while (L[s] != -1) s++;
24        fill(all(dad), -1); fill(all(seen), 0);
25        for(int k = 0; k < n; k++)
26            dist[k] = cost[s][k] - u[s] - v[k];
27        int j = 0;
28        while (true) {
29            j = -1;
30            for(int k = 0; k < n; k++){
31                if (seen[k]) continue;
32                if (j == -1 || dist[k] < dist[j]) j = k;
33            }
34            seen[j] = 1;
35            int i = R[j];
36            if (i == -1) break;
37            for (int k = 0; k < n; k++) {
38                if (seen[k]) continue;
39                auto new_dist = dist[j] + cost[i][k] - u[i] - v[k];
40                if (dist[k] > new_dist) {
41                    dist[k] = new_dist;
42                    dad[k] = j;
43                }
44            }
45        }
46        for (int k = 0; k < n; k++) {
47            if (k == j || !seen[k]) continue;
48            auto w = dist[k] - dist[j];
49            v[k] += w, u[R[k]] -= w;
50        }
51        u[s] += dist[j];
52        while (dad[j] >= 0) {
53            int d = dad[j];
54            R[j] = R[d];
55            L[R[j]] = j;
56            j = d;
57        }
58        R[j] = s; L[s] = j;
59    }
60    T value = 0;
61    for (int i = 0; i < n; i++) value += cost[i][L[i]];
62    return value;
63 }
```

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: $\mathcal{O}(n^2m)$

```

1 int solveLinear(vector<vector<double>> A, vector<double> b, vector<double>& x) {
2     const double eps = 1e-12;
3     int n = A.size(), m = x.size(), rank = 0, br, bc;
4     if (n) assert((int)A[0].size() == m);
5     vector<int> col(m); iota(all(col), 0);
6     for(int i = 0; i < n; i++) {
7         double v, bv = 0;
8         for(int r = i; r < n; ++r) for(int c = i; c < m; c++)
9             if ((v = fabs(A[r][c])) > bv)
10                 br = r, bc = c, bv = v;
11         if (bv <= eps) {
12             for(int j = i; j < n; j++)
13                 if (fabs(b[j]) > eps) return -1;
14             break;
15         }
16         swap(A[i], A[br]);
17         swap(b[i], b[br]);
18         swap(col[i], col[bc]);
19         for(int j = 0; j < n; j++)
20             swap(A[j][i], A[j][bc]);
21         bv = 1 / A[i][i];
22         for(int j = i + 1; j < n; j++) {
23             double fac = A[j][i] * bv;
24             b[j] -= fac * b[i];
25             for(int k = i + 1; k < (m); ++k)
26                 A[j][k] -= fac * A[i][k];
27         }
28         rank++;
29     }
30     x.assign(m, 0);
31     for (int i = rank; i--;) {
32         b[i] /= A[i][i];
33         x[col[i]] = b[i];
34         for (int j = 0; j < i; j++)
35             b[j] -= A[j][i] * b[i];
36     }
37     return rank;
38 }

```

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)`

```

1 struct Poly {
2     vector<double> a;
3     double operator()(double x) const {
4         double val = 0;
5         for(int i = a.size(); i--;)
6             (val *= x) += a[i];
7         return val;
8     }
9     void diff() {
10         for (size_t i = 1; i < a.size(); i++)
11             a[i - 1] = i * a[i];
12         a.pop_back();
13     }
14 };
15 vector<double> poly_roots(Poly p, double xmin, double xmax) {
16     if (p.a.size() == 2) return { -p.a[0] / p.a[1] };
17     vector<double> ret;
18     Poly der = p;
19     der.diff();

```

```

20     auto dr = poly_roots(der, xmin, xmax);
21     dr.push_back(xmin - 1);
22     dr.push_back(xmax + 1);
23     sort(all(dr));
24     for (size_t i = 0; i < dr.size() - 1; i++) {
25         double l = dr[i], h = dr[i + 1];
26         bool sign = p(l) > 0;
27         if (sign ^ (p(h) > 0)) {
28             for (int it = 0; it < 60; it++) {
29                 double m = (l + h) / 2, f = p(m);
30                 if ((f <= 0) ^ sign) l = m;
31                 else h = m;
32             }
33             ret.push_back((l + h) / 2);
34         }
35     }
36     return ret;
37 }

```

Fast Fourier Transform

$\text{fft}(a)$ computes $\hat{f}(k) = \sum_x a[x] \exp(2\pi i \cdot kx/N)$ for all k .

Useful for convolution: $\text{conv}(a, b) = c$, 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 $N = 4 \cdot 10^6$)

```

1  typedef complex<double> C;
2  void fft(vector<C>& a) {
3      int n = a.size(), L = 31 - __builtin_clz(n);
4      static vector<complex<long double>> R(2, 1);
5      static vector<C> rt(2, 1); // (^ 10% faster if double)
6      for (int k = 2; k < n; k *= 2) {
7          R.resize(n); rt.resize(n);
8          auto x = polar(1.0L, M_PI / k);
9          for (int i = k; i < 2 * k; i++)
10             rt[i] = R[i] = i & 1 ? R[i / 2] * x : R[i / 2];
11     }
12     vector<int> rev(n);
13     for (int i = 0; i < n; i++)
14         rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
15     for (int i = 0; i < n; i++)
16         if (i < rev[i]) swap(a[i], a[rev[i]]);
17     for (int k = 1; k < n; k *= 2)
18         for (int i = 0; i < n; i += 2 * k)
19             for (int j = 0; j < k; j++) {
20                 auto x = (double*)&rt[j + k], y = (double*)&a[i + j + k];
21                 C z(x[0] * y[0] - x[1] * y[1], x[0] * y[1] + x[1] * y[0]);
22                 a[i + j + k] = a[i + j] - z;
23                 a[i + j] += z;
24             }
25     }
26 vector<double> conv(const vector<double>& a, const vector<double>& b) {
27     if (a.empty() || b.empty()) return {};
28     vector<double> res(a.size() + a.size() - 1);
29     int L = 32 - __builtin_clz(res.size()), n = 1 << L;
30     vector<C> in(n), out(n);
31     copy(all(a), begin(in));
32     for (size_t i = 0; i < a.size(); i++)
33         in[i].imag(b[i]);
34     fft(in);
35     for (C& x : in) x *= x;
36     for (int i = 0; i < n; i++)
37         out[i] = in[-i & (n - 1)] - conj(in[i]);
38     fft(out);
39     for (size_t i = 0; i < res.size(); i++)
40         res[i] = imag(out[i]) / (4 * n);
41     return res;
42 }

```

Misc

Template

```

1 #pragma GCC optimize("Ofast")
2 #include <bits/stdc++.h>
3 #define all(x) begin(x),end(x)
4 using namespace std;
5 using ll = long long;
6
7 int main() {
8     ios_base::sync_with_stdio(false);
9     cin.tie(nullptr);
10 }

```

Polynomial Hash

```

1 using lll = __int128_t;
2 ll P = 12233720368547789LL;
3 ll B = 260;
4 struct PolyHash {
5     vector<ll> hashes, ex;
6     PolyHash(const string& s) : hashes(s.size()), ex(s.size() + 1) {
7         hashes[0] = s[0] + 1;
8         ex[0] = 1; ex[1] = B;
9         for (size_t i = 1; i < s.size(); i++) {
10             hashes[i] = ((hashes[i - 1] * B) % P + s[i] + 1) % P;
11             ex[i + 1] = (ex[i] * B) % P;
12         }
13     }
14     ll hash(ll lo, ll hi) {
15         if (lo == 0) return hashes[hi];
16         return (hashes[hi] - ((lll)hashes[lo - 1] * ex[hi - lo]) % P + P) % P;
17     }
18 };

```

Compilation Script

```

1 #!/bin/bash
2 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

```

1 #!/bin/bash
2 folder=$1;shift
3 for f in $folder/*.in; do
4     echo $f
5     pre=${f%.in}
6     out=$pre.out
7     $* < $f > $out
8     diff $out $pre.ans

```