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 == b:
3          return sqrt(dist2(p, s))
4      se, sp = vecsub(b, 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

```python
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)
```

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))
```

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))))
```

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.

```python
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
```

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

```cpp
struct SegTree {
    using T = ll;
    static constexpr T unit = 0;
    T f(T a, T b) { return a + b; }
    vector<T> s; ll n;
    SegTree(ll n) : s(2*n, unit), n(n) {}
    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

```
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 };
```

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] = 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.

```cpp
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;
}
```

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, [&](vi& 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.

```cpp
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) {
    int n = g.size();
    val.assign(n, 0); comp.assign(n, -1);
    Time = ncomps = 0;
    for(int i = 0; i < n; i++)
        if (comp[i] < 0) dfs(i, g, f);
}
```

2-SAT

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

```cpp
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]);
    }
};
```

```cpp
//Usage:
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
```

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

```cpp
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, e, y, top = me;
    for(auto& pa : ed[at]) {
        if (pa.second == par) continue;
        tie(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);
}
```

```cpp
//Usage:
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) {...});
```

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 $U = \max |\mathsf{cap}|$. $\mathcal{O}(\min(E^{1/2}, V^{2/3})E)$ 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;
```

```cpp
            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;
}
```

Numerical

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

```cpp
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;
}
```

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();
```

```
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  }
```

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 };
```