Swarthmore College

cout << 1/0 << endl;

Jay Leeds, Timothy Mou, Runze Wang

ICPC World Finals Egypt

Nov. 15, 2023

Contents

Templates (1)

2sat.cpp

a73310, 66 lines

```
/* kactl 2-SAT Solver
 * Negated variables are represented by bit-inversions (\
     texttt{\tilde{}x}).
 * Usage:
 *  TwoSat ts(number of boolean variables);
 *  ts.either(0, \tilde3); // Var 0 is true or var 3 is
     false
 *  ts.setValue(2); // Var 2 is true
 *  ts.atMostOne({0,\tilde1,2}); // <= 1 of vars 0, \tilde1
     and 2 are true
 *  ts.solve(); // Returns true iff it is solvable
 *  ts.values[0..N-1] holds the assigned values to the vars
     */

struct TwoSat {
  int N;
  vector<vi> gr;
  vi values; // 0 = false, 1 = true

  TwoSat(int n = 0) : N(n), gr(2*n) {}

  int addVar() { // (optional)
    gr.emplace_back();
    gr.emplace_back();
    return 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 setValue(int x) { either(x, x); }

  void atMostOne(const vi& li) { // (optional)
    if (sz(li) <= 1) return;
    int cur = ~li[0];
    FOR(i,2,sz(li)) {
      int next = addVar();
      either(cur, ~li[i]);
      either(cur, next);
      either(~li[i], next);
      cur = ~next;
    }
    either(cur, ~li[1]);
  }

  vi val, comp, z; int time = 0;
  int dfs(int i) {
    int low = val[i] = ++time, x; z.push_back(i);
    for(int 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(i,0,2*N) if (!comp[i]) dfs(i);
    FOR(i,0,N) if (comp[2*i] == comp[2*i+1]) return 0;
    return 1;
  }
};
```

3dHull.cpp

"Point3D.h" de4d99, 49 lines

```
typedef Point3D<double> P3;

struct PR {
  void ins(int x) { (a == -1 ? a : b) = x; }
  void rem(int x) { (a == x ? a : b) = -1; }
  int cnt() { return (a != -1) + (b != -1); }
  int a, b;
};

struct F { P3 q; int a, b, c; };

vector<F> hull3d(const vector<P3>& A) {
  assert(sz(A) >= 4);
  vector<vector<PR>> E(sz(A), vector<PR>(sz(A), {-1, -1}));
#define E(x,y) E[f.x][f.y]
  vector<F> FS;
  auto mf = [&](int i, int j, int k, int l) {
    P3 q = (A[j] - A[i]).cross((A[k] - A[i]));
    if (q.dot(A[l]) > q.dot(A[i]))
      q = q * -1;
    F f{q, i, j, k};
    E(a,b).ins(k); E(a,c).ins(j); E(b,c).ins(i);
    FS.push_back(f);
  };
  FOR(i,0,4) rep(j,i+1,4) rep(k,j+1,4)
    mf(i, j, k, 6 - i - j - k);

  FOR(i,4,sz(A)) {
    FOR(j,0,sz(FS)) {
      F f = FS[j];
      if(f.q.dot(A[i]) > f.q.dot(A[f.a])) {
        E(a,b).rem(f.c);
        E(a,c).rem(f.b);
        E(b,c).rem(f.a);
        swap(FS[j--], FS.back());
        FS.pop_back();
      }
    }
    int nw = sz(FS);
    FOR(j,0,nw) {
      F f = FS[j];
#define C(a, b, c) if (E(a,b).cnt() != 2) mf(f.a, f.b, i, f.
    c);
      C(a, b, c); C(a, c, b); C(b, c, a);
    }
  }
  trav(it, FS) if ((A[it.b] - A[it.a]).cross(
    A[it.c] - A[it.a]).dot(it.q) <= 0) swap(it.c, it.b);
  return FS;
};
```

AdvancedHash.cpp

838740, 46 lines

```
ll base1[MX], base2[MX];
int base;

const ll p1 = MOD;
const ll p2 = MOD+2;

// If you don't need to query arbitrary ranges,
// only maintain val1 and val2 to save space.
// get() = val1 * p2 + val2
struct hsh {
    ll val1, val2;
    vl val1s, val2s;
    vl nums;
    hsh() {
        val1 = 0;
        val2 = 0;
        val1s.pb(0);
        val2s.pb(0);
    }

    void push_back(ll v) {
        v++;
        val1 *= base; val1 += v; val1 %= p1;
        val2 *= base; val2 += v; val2 %= p2;

        val1s.pb(val1);
        val2s.pb(val2);
        nums.pb(v);
    }

    ll get(int L, int R) {
        ll A = (val1s[R+1] - (val1s[L] * base1[R-L+1]) % p1
          + p1) % p1;
        ll B = (val2s[R+1] - (val2s[L] * base2[R-L+1]) % p2
          + p2) % p2;
        return A*p2+B;
    }
};

void prepHash() {
    base = uid(MOD/5, MOD/2);

    base1[0] = 1; base2[0] = 1;
    FOR(i, 1, MX) {
        base1[i] = (base1[i-1] * base) % p1;
        base2[i] = (base2[i-1] * base) % p2;
    }
}
```

AhoCorasick.cpp

b658a8, 80 lines

```
// NOTE: val/num variables in v, cnt argument to add_string,
// and dfs/compute methods may be unnecessary
struct AhoCorasick {
    static const int K = 26;

    struct V {
        int nxt[K];
        bool leaf = false;
        int p = -1;
```

```
    char pch;
    int link = -1;
    int go[K];
    ll val = -1;
    ll num = 0;

    V(int p=-1, char ch='$') : p(p), pch(ch) {
        fill(begin(nxt), end(nxt), -1);
        fill(begin(go), end(go), -1);
    }
};

vector<V> t;

void init() {
    V v; t.pb(v);
}

void add_string(string const& s, int cnt) {
    int v = 0;
    trav(ch, s) {
        int c = ch - 'a';
        if (t[v].nxt[c] == -1) {
            t[v].nxt[c] = t.size();
            t.emplace_back(v, ch);
        }
        v = t[v].nxt[c];
    }
    t[v].leaf = true;
    t[v].num = cnt;
}

ll dfs(int v) {
    if (t[v].val != -1) {
        return t[v].val;
    }
    ll ans = t[v].num;
    ans += dfs(get_link(v));
    return t[v].val = ans;
}

// sets value for each node to sum of values
// over suffix links
void compute() {
    t[0].val = 0;
    FOR(i, 1, sz(t)) {
        dfs(i);
    }
}

int get_link(int v) {
    if (t[v].link == -1) {
        if (v == 0 || t[v].p == 0)
            t[v].link = 0;
        else
            t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
}

int go(int v, char ch) {
    int c = ch - 'a';
    if (t[v].go[c] == -1) {
        if (t[v].nxt[c] != -1)
            t[v].go[c] = t[v].nxt[c];
        else
            t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch
                );
    }
    return t[v].go[c];
  }
};
```

Angle.cpp

0f0602, 35 lines

```
struct Angle {
  int x, y;
  int t;
  Angle(int x, int y, int t=0) : x(x), y(y), t(t) {}
  Angle operator-(Angle b) const { return {x-b.x, y-b.y, t};
      }
  int half() const {
    assert(x || y);
    return y < 0 || (y == 0 && x < 0);
  }
  Angle t90() const { return {-y, x, t + (half() && x >= 0)}
      ; }
  Angle t180() const { return {-x, -y, t + half()}; }
  Angle t360() const { return {x, y, t + 1}; }
};
bool operator<(Angle a, Angle b) {
  // add a.dist2() and b.dist2() to also compare distances
  return make_tuple(a.t, a.half(), a.y * (ll)b.x) <
         make_tuple(b.t, b.half(), a.x * (ll)b.y);
}

// Given two points, this calculates the smallest angle
    between
// them, i.e., the angle that covers the defined line
    segment.
pair<Angle, Angle> segmentAngles(Angle a, Angle b) {
  if (b < a) swap(a, b);
  return (b < a.t180() ?
          make_pair(a, b) : make_pair(b, a.t360()));
}
Angle operator+(Angle a, Angle b) { // point a + vector b
  Angle r(a.x + b.x, a.y + b.y, a.t);
  if (a.t180() < r) r.t--;
  return r.t180() < a ? r.t360() : r;
}
Angle angleDiff(Angle a, Angle b) { // angle b - angle a
  int tu = b.t - a.t; a.t = b.t;
  return {a.x*b.x + a.y*b.y, a.x*b.y - a.y*b.x, tu - (b < a)
      };
}
```

Basis.cpp

95f6d3, 27 lines

```
const int D; //length of masks

ll basis[D]; // basis[i] keeps the mask of the vector whose
    f value is i

int bs = 0; //basis size

void insertVector(ll mask) {
  for (int i = 0; i < D; i++) {
    if ((mask & 1ll << i) == 0) continue;

    if (!basis[i]) {
      basis[i] = mask;
      ++bs;
      return;
    }

    mask ^= basis[i];
  }
}

bool inSpan(ll mask) {
  for (int i = 0; i < D; i++) {
    if ((mask & 1ll << i) == 0) continue;
    mask ^= basis[i];
  }
  return mask == 0;
}
```

Berlekamp-Massey.cpp

335c7c, 44 lines

```
vector<int> berlekamp_massey(const vector<int> &a) {
  vector<int> v, last;  // v is the answer, 0-based, p is
      the module
  int k = -1, delta = 0;

  for (int i = 0; i < (int)a.size(); i++) {
    int tmp = 0;
    for (int j = 0; j < (int)v.size(); j++)
      tmp = (tmp + (long long)a[i - j - 1] * v[j]) % p;

    if (a[i] == tmp) continue;

    if (k < 0) {
      k = i;
      delta = (a[i] - tmp + p) % p;
      v = vector<int>(i + 1);

      continue;
    }

    vector<int> u = v;
    int val = (long long)(a[i] - tmp + p) * power(delta, p -
        2) % p;

    if (v.size() < last.size() + i - k) v.resize(last.size()
         + i - k);

    (v[i - k - 1] += val) %= p;

    for (int j = 0; j < (int)last.size(); j++) {
      v[i - k + j] = (v[i - k + j] - (long long)val * last[j
          ]) % p;
      if (v[i - k + j] < 0) v[i - k + j] += p;
    }

    if ((int)u.size() - i < (int)last.size() - k) {
      last = u;
      k = i;
      delta = a[i] - tmp;
      if (delta < 0) delta += p;
    }
  }

  for (auto &x : v) x = (p - x) % p;
  v.insert(v.begin(), 1);
```

```
  return v;  // $\forall i, \sum_{j = 0} ^ m a_{i - j} v_j =
        0$
}
```

BinSearchSegtree.cpp

853b41, 54 lines

```
const ll identity = 0;
const ll SZ = 131072;

ll sum[2*SZ], lazy[2*SZ];

ll combine(ll A, ll B) {
    return A+B;
}

ll combineUpd(ll A, ll B) {
    return A+B;
}

void push(int index, ll L, ll R) {
    sum[index] = combineUpd(sum[index], lazy[index]);
    if (L != R) lazy[2*index] = combineUpd(lazy[2*index],
         lazy[index]), lazy[2*index+1] = combineUpd(lazy[2*
         index+1], lazy[index]);
    lazy[index] = identity;
}

void pull(int index) {
    sum[index] = combine(sum[2*index], sum[2*index+1]);
}

bool checkCondition(int index) {
    //FILL THIS IN
}

ll query(int lo = 0, int hi = SZ-1, int index = 1, ll L = 0,
      ll R = SZ-1) { //returns first node satisfying con
    push(index, L, R);
    if (lo > R || L > hi) return -1;
    bool condition = checkCondition(index);
    if (L == R) {
        return (condition ? L : -1);
    }
    int M = (L+R) / 2;
    if (checkCondition(2*index)) {
        return query(lo, hi, 2*index, L, M);
    }
    return query(lo, hi, 2*index+1, M+1, R);
}

void update(int lo, int hi, ll increase, int index = 1, ll L
      = 0, ll R = SZ-1) {
    push(index, L, R);
    if (hi < L || R < lo) return;
    if (lo <= L && R <= hi) {
        lazy[index] = increase;
        push(index, L, R);
        return;
    }

    int M = (L+R) / 2;
    update(lo, hi, increase, 2*index, L, M); update(lo, hi,
         increase, 2*index+1, M+1, R);
    pull(index);
}
```

BipartiteMatching.cpp

7a6c4b, 24 lines

```
//Storing the graph
vector<int> g[maxn];
//Storing whether we have visited a node
bool vis[maxn];
//Storing the vertex matched to
int match[maxn];

bool hungarian(int u){
  for (int i = 0;i < g[u].size();++i){
    int v = g[u][i];
    if (!vis[v]){
      vis[v] = true;
      if (!match[v] || hungarian(match[v])){
        match[u] = v; match[v] = u; return true;
      }
    }
  }
  return false;
}
//in main: call hungarian for each vertex on one side
for (int i = 1;i <= nl;++i){
    memset(vis,false,sizeof vis);
    if (hungarian(i)) ans++; //if we can match i
}
```

BipartiteMatchingWithWeights.cpp

c21d5f, 74 lines

```
ll g[maxn][maxn];
ll fx[maxn], fy[maxn], a[maxn], b[maxn], slack[maxn],pre[
     maxn];
bool visx[maxn], visy[maxn];
int q[maxn];
int n;

void augment(int v){
  if (!v) return; fy[v] = pre[v]; augment(fx[pre[v]]); fx[fy
       [v]] = v;
}

void bfs(int source){
  memset(visx, 0, sizeof visx);
  memset(visy, 0, sizeof visy);
  memset(slack, 127, sizeof slack);
  int head, tail; head = tail = 1;
  q[tail] = source;
  while (true){
    while (head <= tail){
      int u = q[head++];
      visx[u] = true;
      for (int v = 1;v <= n;++v){
        if (!visy[v]){
          if (a[u] + b[v] == g[u][v])
          {
            visy[v] = true; pre[v] = u;
            if (!fy[v]){
              augment(v); return;
            }
            q[++tail] = fy[v];continue;
          }
          if (slack[v] > a[u] + b[v] - g[u][v]){
            slack[v] = a[u] + b[v] - g[u][v];
            pre[v] = u;
          }
        }
      }
    }
    ll d = inf;
    for (int i = 1;i <= n;++i){
      if (!visy[i]) d = min(d, slack[i]);
    }
    for (int i = 1;i <= n;++i){
      if (visx[i]) a[i] -= d;
      if (visy[i]) b[i] += d;
      else slack[i] -= d;
    }
    for (int v = 1;v <= n;++v){
      if (!visy[v] && !slack[v]){
        visy[v] = true;
        if (!fy[v]){
          augment(v);
          return;
        }
        q[++tail] = fy[v];
      }
    }
  }
}

ll km(){
  for (int i = 1;i <= n;++i){
    a[i] = -inf;
    b[i] = 0;
    for (int j = 1;j <= n;++j) a[i] = max(a[i], g[i][j]);
  }
  memset(fx, 0, sizeof fx);
  memset(fy, 0, sizeof fy);
  for (int i = 1;i <= n;++i) bfs(i);
  ll ans = 0;
  for (int i = 1;i <= n;++i) ans += a[i] + b[i];
  //vertex i on left is matched to g2[i][fx[i]] * fx[i]
  //g2[a][b]=1 iff exists edge ab
  return ans;
}
```

BIT.cpp

992848, 14 lines

```
const int maxn = 30005;
int n, bit[maxn];
void add(int i, int x) {
    for (; i <= n; i += i & (-i))
        bit[i] += x;
}

int sum(int i) {
    int r = 0;
    for (; i; i -= i & (-i)) {
        r += bit[i];
    }
    return r;
}
```

BlockCut.cpp

512e87, 63 lines

```
// note: just need dfs if all you need is cutpoints or BCCs
// if all you need is BCCs, ignore id
// if all you need is cutpoints, ignore stk, id, comps
// can add LCA on top of this to check if a path from a-b
     that
// does not visit c exists
```

```
// assumes graph is simple; must dfs multiple times if not
    connected
// be careful about handling isolated vertices

int N;
vector<vi> graph(MX), comps;
vi stk, num(MX), lo(MX), is_cp(MX), id(MX);
int ct = 0;

void dfs(int v, int p) {
    num[v] = lo[v] = ++ct;
    if (sz(graph[v]) == 0) {
        comps.pb({v});
        return;
    }
    stk.pb(v);
    trav(a, graph[v]) {
        if (a == p) continue;
        if (num[a]) {
            lo[v] = min(lo[v], num[a]);
        } else {
            dfs(a, v);
            lo[v] = min(lo[v], lo[a]);
            if (lo[a] >= num[v]) {
                is_cp[v] = (num[v] > 1 || num[a] > 2);
                comps.pb({v});
                while (comps.back().back() != a) {
                    comps.back().pb(stk.back());
                    stk.pop_back();
                }
            }
        }
    }
}

vector<vi> bct;

void build_tree() {
    int nid = 0;
    FOR(i, N) {
        if (is_cp[i]) {
            id[i] = nid++;
            bct.pb({});
        }
    }

    trav(comp, comps) {
        int v = nid++;
        bct.pb({});
        trav(u, comp) {
            if (!is_cp[u]) {
                id[u] = v;
            } else {
                bct[v].pb(id[u]);
                bct[id[u]].pb(v);
            }
        }
    }
}
```

Bridge.cpp

a44485, 33 lines

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] > tin[v])
                IS_BRIDGE(v, to);
        }
    }
}

void find_bridges() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(i);
    }
}
```

CentroidDecomp.cpp

b8fb48, 54 lines

```
struct CentroidDecomposition {
  vector<set<int>> tree; // it's not vector<vector<int>>!
  vector<int> dad;
  vector<int> sub;
    vector<int> dep;

  CentroidDecomposition(vector<set<int>> &tree) : tree(tree)
      {
    int n = tree.size();
    dad.resize(n);
    sub.resize(n);
      dep.resize(n);
    build(0, -1);
  }

  void build(int u, int p) {
    int n = dfs(u, p); // find the size of each subtree
    int centroid = dfs(u, p, n); // find the centroid
      if (p == -1) {
          dep[centroid] = 0;
      } else {
          dep[centroid] = dep[p] + 1;
      }
    if (p == -1) p = centroid; // dad of root is the root
        itself
    dad[centroid] = p;
    // for each tree resulting from the removal of the
        centroid
      while (!tree[centroid].empty()) {
          int v = *(tree[centroid].begin());
       tree[centroid].erase(v); // remove the edge to
           disconnect
      tree[v].erase(centroid); // the component from the
          tree
      build(v, centroid);
        }

  }

  int dfs(int u, int p) {
    sub[u] = 1;

    for (auto v : tree[u])
      if (v != p) sub[u] += dfs(v, u);

    return sub[u];
  }

  int dfs(int u, int p, int n) {
    for (auto v : tree[u])
      if (v != p and sub[v] > n/2) return dfs(v, u, n);

    return u;
  }

  int operator[](int i) {
    return dad[i];
  }
};
```

ChordalGraph.cpp

78d46d, 44 lines

```
//Maximum Cardinality Search
while (cur) {
  p[cur] = h[nww];
  rnk[p[cur]] = cur;
  h[nww] = nxt[h[nww]];
  lst[h[nww]] = 0;
  lst[p[cur]] = nxt[p[cur]] = 0;
  tf[p[cur]] = true;
  for (vector<int>::iterator it = G[p[cur]].begin(); it != G
      [p[cur]].end();
      it++)
    if (!tf[*it]) {
      if (h[deg[*it]] == *it) h[deg[*it]] = nxt[*it];
      nxt[lst[*it]] = nxt[*it];
      lst[nxt[*it]] = lst[*it];
      lst[*it] = nxt[*it] = 0;
      deg[*it]++;
      nxt[*it] = h[deg[*it]];
      lst[h[deg[*it]]] = *it;
      h[deg[*it]] = *it;
    }
  cur--;
  if (h[nww + 1]) nww++;
  while (nww && !h[nww]) nww--;
}

//Checking if a sequence is a perfect elimination ordering
jud = true;
for (int i = 1; i <= n; i++) {
  cur = 0;
  for (vector<int>::iterator it = G[p[i]].begin(); it != G[p
      [i]].end(); it++)
    if (rnk[p[i]] < rnk[*it]) {
      s[++cur] = *it;
      if (rnk[s[cur]] < rnk[s[1]]) swap(s[1], s[cur]);
```

```
    }
  for (int j = 2; j <= cur; j++)
    if (!st[s[1]].count(s[j])) {
      jud = false;
      break;
    }
}
if (!jud)
  printf("Imperfect\n");
else
  printf("Perfect\n");
```

CircleIntersection.cpp

"Point.h" 84d6d3, 11 lines

```
typedef Point<double> P;
bool circleInter(P a,P b,double r1,double r2,pair<P, P>* out
    ) {
  if (a == b) { assert(r1 != r2); return false; }
  P vec = b - a;
  double d2 = vec.dist2(), sum = r1+r2, dif = r1-r2,
         p = (d2 + r1*r1 - r2*r2)/(d2*2), h2 = r1*r1 - p*p*
             d2;
  if (sum*sum < d2 || dif*dif > d2) return false;
  P mid = a + vec*p, per = vec.perp() * sqrt(fmax(0, h2) /
      d2);
  *out = {mid + per, mid - per};
  return true;
}
```

CircleLine.cpp

"Point.h", "lineDistance.h", "LineProjectionReflection.h" debf86, 8 lines

```
template<class P>
vector<P> circleLine(P c, double r, P a, P b) {
  double h2 = r*r - a.cross(b,c)*a.cross(b,c)/(b-a).dist2();
  if (h2 < 0) return {};
  P p = lineProj(a, b, c), h = (b-a).unit() * sqrt(h2);
  if (h2 == 0) return {p};
  return {p - h, p + h};
}
```

CirclePolygonIntersection.cpp

"../../content/geometry/Point.h" 3e5102, 19 lines

```
typedef Point<double> P;
#define arg(p, q) atan2(p.cross(q), p.dot(q))
double circlePoly(P c, double r, vector<P> ps) {
  auto tri = [&](P p, P q) {
    auto r2 = r * r / 2;
    P d = q - p;
    auto a = d.dot(p)/d.dist2(), b = (p.dist2()-r*r)/d.dist2
        ();
    auto det = a * a - b;
    if (det <= 0) return arg(p, q) * r2;
    auto s = max(0., -a-sqrt(det)), t = min(1., -a+sqrt(det)
        );
    if (t < 0 || 1 <= s) return arg(p, q) * r2;
    P u = p + d * s, v = p + d * t;
    return arg(p,u) * r2 + u.cross(v)/2 + arg(v,q) * r2;
  };
  auto sum = 0.0;
  FOR(i,0,sz(ps))
    sum += tri(ps[i] - c, ps[(i + 1) % sz(ps)] - c);
  return sum;
}
```

CircleTangents.cpp

"Point.h" b0153d, 13 lines

```
template<class P>
vector<pair<P, P>> tangents(P c1, double r1, P c2, double r2
    ) {
  P d = c2 - c1;
  double dr = r1 - r2, d2 = d.dist2(), h2 = d2 - dr * dr;
  if (d2 == 0 || h2 < 0)  return {};
  vector<pair<P, P>> out;
  for (double sign : {-1, 1}) {
    P v = (d * dr + d.perp() * sqrt(h2) * sign) / d2;
    out.push_back({c1 + v * r1, c2 + v * r2});
  }
  if (h2 == 0) out.pop_back();
  return out;
}
```

Circumcircle.cpp

"Point.h" 1caa3a, 9 lines

```
typedef Point<double> P;
double ccRadius(const P& A, const P& B, const P& C) {
  return (B-A).dist()*(C-B).dist()*(A-C).dist()/
      abs((B-A).cross(C-A))/2;
}
P ccCenter(const P& A, const P& B, const P& C) {
  P b = C-A, c = B-A;
  return A + (b*c.dist2()-c*b.dist2()).perp()/b.cross(c)/2;
}
```

ClosestPair.cpp

"Point.h" d31bbf, 17 lines

```
typedef Point<ll> P;
pair<P, P> closest(vector<P> v) {
  assert(sz(v) > 1);
  set<P> S;
  sort(all(v), [](P a, P b) { return a.y < b.y; });
  pair<ll, pair<P, P>> ret{LLONG_MAX, {P(), P()}};
  int j = 0;
  trav(p, v) {
    P d{1 + (ll)sqrt(ret.first), 0};
    while (v[j].y <= p.y - d.x) S.erase(v[j++]);
    auto lo = S.lower_bound(p - d), hi = S.upper_bound(p + d
        );
    for (; lo != hi; ++lo)
      ret = min(ret, {(*lo - p).dist2(), {*lo, p}});
    S.insert(p);
  }
  return ret.second;
}
```

ConvexHull.cpp

16cd48, 47 lines

```
// WARNING: May include multiple copies of the same point if
      duplicates are included in input
// Returns convex hull in clockwise order

struct pt {
    ld x, y;
};

bool cmp(pt a, pt b) {
    return a.x < b.x || (a.x == b.x && a.y < b.y);
}

bool cw(pt a, pt b, pt c) {
    return a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) < 0;
}

bool ccw(pt a, pt b, pt c) {
    return a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) > 0;
}

void convex_hull(vector<pt>& a) {
    if (sz(a) == 1)
        return;

    sort(all(a), &cmp);
    pt p1 = a[0], p2 = a.back();
    vector<pt> up, down;
    up.pb(p1);
    down.pb(p1);
    FOR(i, 1, sz(a)) {
        if (i == sz(a) - 1 || cw(p1, a[i], p2)) {
            while (sz(up) >= 2 && !cw(up[sz(up)-2], up[sz(up
                )-1], a[i]))
                up.pop_back();
            up.pb(a[i]);
        }
        if (i == sz(a) - 1 || ccw(p1, a[i], p2)) {
            while(sz(down) >= 2 && !ccw(down[sz(down)-2],
                down[sz(down)-1], a[i]))
                down.pop_back();
            down.pb(a[i]);
        }
    }

    a.clear();
    for (int i = 0; i < sz(up); i++)
        a.push_back(up[i]);
    for (int i = sz(down) - 2; i > 0; i--)
        a.push_back(down[i]);
}
```

ConvexHullTrick.cpp

875d66, 26 lines

```
//This represents those relying on j only;i.e intercept
inline ll y_axis(int j){
    return dp[j] + a * s[j] * s[j] - b * s[j];
}
//This represents those relying on both i and j; i.e slope
inline ll x_axis(int j){
    return s[j];
}
inline ld getSlope(int j,int k){
    ld y = y_axis(k) - y_axis(j);
    ld x = x_axis(k) - x_axis(j);
    return y / x;
}
//s stores prefix sum
int head = 1;int tail = 1;
q[head] = 0;
for (int i = 1;i <= n;i++){
  dp[i] = a*s[i]*s[i] + b*s[i] + c;
  while (head < tail && getSlope(q[head],q[head+1]) >= 2*a*s
      [i]) head++;
  if (head <= tail){
    int j = q[head];
    dp[i] = max(dp[i], dp[j] + a *(s[i] - s[j]) * (s[i] - s[
        j]) + b*(s[i] - s[j]) + c);
```

```
  }
  while (head < tail && getSlope(q[tail],i) >= getSlope(q[
      tail-1],q[tail])) tail--;
  q[++tail] = i;
}
```

CRT.cpp

118857, 32 lines

```
//each is x mod p_i = a_i
ll p[maxn], a[maxn];
//for quickmult see pollard rho
ll exgcd(ll x, ll y,ll & a, ll & b){
    if (y == 0){
        a = 1;b = 0;return x;
    }
    ll d = exgcd(y, x%y, a, b);
    ll temp = a; a = b; b = temp - (x / y) * b;
    return d;
}

int first_nontrivial = 0;
ll current_p ;
ll sol = 0; //this is the solution
for (int i = 1;i <= n;i++){
  if (p[i] != 1){
    first_nontrivial = i;
    current_p = p[i]; sol = a[i];
    break;
  }
}
for (int i = first_nontrivial+1;i <= n;i++){
  ll x,y;
  if (p[i] == 1) continue;
  ll d = exgcd(current_p, p[i], x, y);
  ll r = ((a[i] - sol) % p[i] + p[i])%p[i];
  ll temp = quickmult(x, r / d,p[i] / d);
  sol = sol + current_p * temp;
  current_p = current_p / d * p[i];
  sol = (sol % current_p + current_p) % current_p;
}
```

DelaunayTriangulation.cpp

"Point.h", "3dHull.h" d173fc, 10 lines

```
template<class P, class F>
void delaunay(vector<P>& ps, F trifun) {
  if (sz(ps) == 3) { int d = (ps[0].cross(ps[1], ps[2]) < 0)
      ;
    trifun(0,1+d,2-d); }
  vector<P3> p3;
  trav(p, ps) p3.emplace_back(p.x, p.y, p.dist2());
  if (sz(ps) > 3) trav(t, hull3d(p3)) if ((p3[t.b]-p3[t.a]).
      cross(p3[t.c]-p3[t.a]).dot(P3(0,0,1)) < 0)
    trifun(t.a, t.c, t.b);
}
```

Dinic.cpp

681177, 83 lines

```
//from https://cp-algorithms.com/graph/dinic.html
//Complexity: O(E*V^2)
struct Edge {
   int v, u;
   ll cap, flow = 0;
   Edge(int v, int u, ll cap) : v(v), u(u), cap(cap) {}
};
struct Dinic {
   const ll flow_inf = 1e18;
   vector<Edge> edges;
   vector<vi> adj;
   int n, m = 0;
   int s, t;
   vi lev, ptr;
   queue<int> q;

   Dinic(int n, int s, int t) : n(n), s(s), t(t) {
       adj.resize(n);
       lev.resize(n);
       ptr.resize(n);
   }

   void add_edge(int v, int u, ll cap) {
       edges.emplace_back(v, u, cap);
       edges.emplace_back(u, v, 0);
       adj[v].push_back(m);
       adj[u].push_back(m + 1);
       m += 2;
   }

   bool bfs() {
       while (!q.empty()) {
           int v = q.front();
           q.pop();
           trav(id, adj[v]) {
               if (edges[id].cap - edges[id].flow < 1)
                   continue;
               if (lev[edges[id].u] != -1)
                   continue;
               lev[edges[id].u] = lev[v] + 1;
               q.push(edges[id].u);
           }
       }
       return lev[t] != -1;
   }

   ll dfs(int v, ll pu) {
       if (pu == 0)
           return 0;
       if (v == t)
           return pu;
       for (int& cid = ptr[v]; cid < sz(adj[v]); cid++) {
           int id = adj[v][cid];
           int u = edges[id].u;
           if (lev[v] + 1 != lev[u] || edges[id].cap - edges
               [id].flow < 1)
               continue;
           ll tr = dfs(u, min(pu, edges[id].cap - edges[id].
               flow));
           if (tr == 0)
               continue;
           edges[id].flow += tr;
           edges[id ^ 1].flow -= tr;
           return tr;
       }
       return 0;
   }

   ll flow() {
       ll f = 0;
       while (true) {
           fill(all(lev), -1);
           lev[s] = 0;
           q.push(s);
           if (!bfs())
               break;
           fill(all(ptr), 0);
           while (ll pu = dfs(s, flow_inf)) {
               f += pu;
           }
       }
       return f;
   }
};
```

DSU.cpp

89f1c6, 19 lines

```
int parent[MX], si[MX];

void init(int N) {
    FOR(i, N) parent[i] = i, si[i] = 0;
}

int get(int x) {
    if (parent[x] != x) parent[x] = get(parent[x]);
    return parent[x];
}

void unify(int x, int y) {
    x = get(x); y = get(y);
    if (x == y) return;
    if (si[x] < si[y]) swap(x, y);
    if (si[x] == si[y]) si[x]++;
    parent[y] = x;

}
```

EulerPath.cpp

274951, 25 lines

```
int N, M;
vector<vpi> graph(MX); //{ed, edNum}
vector<vpi::iterator> its(MX);
vector<bool> used(MX);

vpi eulerPath(int r) {
    FOR(i, N) its[i] = begin(graph[i]);
    FOR(i, M) used[i] = false;
    vpi ans, s{{r, -1}};
    int lst = -1;
    while (sz(s)) {
        int x = s.back().f; auto &it = its[x], en = end(
            graph[x]);
        while (it != en && used[it->s]) it++;

        if (it == en) {
            if (lst != -1 && lst != x) return {};
            ans.pb(s.back()); s.pop_back(); if (sz(s)) lst =
                s.back().f;
        } else {
            s.pb(*it);
            used[it->s] = 1;
        }
    } // Returns path in reverse order if graph is directed.
    if (sz(ans) != M+1) return {};
    return ans;
}
```

FastDelaunay.cpp

"Point.h" 1c46ca, 88 lines

```
typedef Point<ll> P;
typedef struct Quad* Q;
typedef __int128_t lll; // (can be ll if coords are < 2e4)
P arb(LLONG_MAX,LLONG_MAX); // not equal to any other point

struct Quad {
  bool mark; Q o, rot; P p;
  P F() { return r()->p; }
  Q r() { return rot->rot; }
  Q prev() { return rot->o->rot; }
  Q next() { return r()->prev(); }
};

bool circ(P p, P a, P b, P c) { // is p in the circumcircle?
  lll p2 = p.dist2(), A = a.dist2()-p2,
      B = b.dist2()-p2, C = c.dist2()-p2;
  return p.cross(a,b)*C + p.cross(b,c)*A + p.cross(c,a)*B >
       0;
}
Q makeEdge(P orig, P dest) {
  Q q[] = {new Quad{0,0,0,orig}, new Quad{0,0,0,arb},
           new Quad{0,0,0,dest}, new Quad{0,0,0,arb}};
  FOR(i,0,4)
    q[i]->o = q[-i & 3], q[i]->rot = q[(i+1) & 3];
  return *q;
}
void splice(Q a, Q b) {
  swap(a->o->rot->o, b->o->rot->o); swap(a->o, b->o);
}
Q connect(Q a, Q b) {
  Q q = makeEdge(a->F(), b->p);
  splice(q, a->next());
  splice(q->r(), b);
  return q;
}

pair<Q,Q> rec(const vector<P>& s) {
  if (sz(s) <= 3) {
    Q a = makeEdge(s[0], s[1]), b = makeEdge(s[1], s.back())
        ;
    if (sz(s) == 2) return { a, a->r() };
    splice(a->r(), b);
    auto side = s[0].cross(s[1], s[2]);
    Q c = side ? connect(b, a) : 0;
    return {side < 0 ? c->r() : a, side < 0 ? c : b->r() };
  }

#define H(e) e->F(), e->p
#define valid(e) (e->F().cross(H(base)) > 0)
  Q A, B, ra, rb;
  int half = sz(s) / 2;
  tie(ra, A) = rec({all(s) - half});
  tie(B, rb) = rec({sz(s) - half + all(s)});
  while ((B->p.cross(H(A)) < 0 && (A = A->next())) ||
         (A->p.cross(H(B)) > 0 && (B = B->r()->o)));
  Q base = connect(B->r(), A);
  if (A->p == ra->p) ra = base->r();
  if (B->p == rb->p) rb = base;

#define DEL(e, init, dir) Q e = init->dir; if (valid(e)) \
    while (circ(e->dir->F(), H(base), e->F())) { \
      Q t = e->dir; \
      splice(e, e->prev()); \
      splice(e->r(), e->r()->prev()); \
      e = t; \
    }
  for (;;) {
    DEL(LC, base->r(), o);  DEL(RC, base, prev());
    if (!valid(LC) && !valid(RC)) break;
    if (!valid(LC) || (valid(RC) && circ(H(RC), H(LC))))
      base = connect(RC, base->r());
    else
      base = connect(base->r(), LC->r());
  }
  return { ra, rb };
}

vector<P> triangulate(vector<P> pts) {
  sort(all(pts));  assert(unique(all(pts)) == pts.end());
  if (sz(pts) < 2) return {};
  Q e = rec(pts).first;
  vector<Q> q = {e};
  int qi = 0;
  while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
#define ADD { Q c = e; do { c->mark = 1; pts.push_back(c->p)
    ; \
  q.push_back(c->r()); c = c->next(); } while (c != e); }
  ADD; pts.clear();
  while (qi < sz(q)) if (!(e = q[qi++])->mark) ADD;
  return pts;
}
```

FastHashTable.cpp

<ext/pb_ds/assoc_container.hpp> f39118, 19 lines

```
using namespace __gnu_pbds;

struct chash {
    static uint64_t splitmix64(uint64_t x) {
        // http://xorshift.di.unimi.it/splitmix64.c
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM = chrono::
            steady_clock::now().time_since_epoch().count();
        return splitmix64(x + FIXED_RANDOM);
    }
};

template<typename T> using pb_set = gp_hash_table<T,
    null_type, chash>;
template<typename T, typename U> using pb_map =
    gp_hash_table<T, U, chash>;
```

FFT.cpp

89f37e, 61 lines

```
using cd = complex<double>;
const double PI = acos(-1);

int reverse(int num, int lg_n) {
    int res = 0;
    for (int i = 0; i < lg_n; i++) {
        if (num & (1 << i))
            res |= 1 << (lg_n - 1 - i);
    }
    return res;
}

void fft(vector<cd> & a, bool invert) {
    int n = a.size();
    int lg_n = 0;
    while ((1 << lg_n) < n)
        lg_n++;

    for (int i = 0; i < n; i++) {
        if (i < reverse(i, lg_n))
            swap(a[i], a[reverse(i, lg_n)]);
    }

    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * PI / len * (invert ? -1 : 1);
        cd wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            cd w(1);
            for (int j = 0; j < len / 2; j++) {
                cd u = a[i+j], v = a[i+j+len/2] * w;
                a[i+j] = u + v;
                a[i+j+len/2] = u - v;
                w *= wlen;
            }
        }
    }

    if (invert) {
        for (cd & x : a)
            x /= n;
    }
}
vector<ll> multiply(vector<ll> const& a, vector<ll> const& b
    ) {
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end()
        );
    int n = 1;
    while (n < a.size() + b.size())
        n <<= 1;
    fa.resize(n);
    fb.resize(n);

    fft(fa, false);
    fft(fb, false);
    for (int i = 0; i < n; i++)
        fa[i] *= fb[i];
    fft(fa, true);

    vector<ll> result(n);
    for (int i = 0; i < n; i++)
        result[i] = round(fa[i].real());
    return result;
}
```

GaussElim.cpp

369f57, 24 lines

```
int h = 0, k = 0;
while (h < sz(coef) && k < sz(coef[0])) {
    int i_max = h; ld max_val = abs(coef[h][k]);
    FOR(j, h+1, sz(coef)) if (ckmax(max_val, abs(coef[j][k])
        )) i_max = j;
    if (abs(coef[i_max][k]) < 1e-9) {
        k++;
    } else {
```

```
        FOR(i, sz(coef[0])) {
            swap(coef[h][i], coef[i_max][i]);
        }
        ld invVal = (ld) 1 / coef[h][k];
        FOR(i, sz(coef[0])) {
            coef[h][i] *= invVal;
        }
        FOR(i, sz(coef)) {
            if (i == h) continue;
            ld cur = coef[i][k];
            FOR(j, sz(coef[0])) {
                coef[i][j] -= cur * coef[h][j];
            }
        }
        h++; k++;
    }
}
```

GeneralMatching.cpp

53f4fc, 78 lines

```
//belong is a DSU; unit = union
int n, match[N], next[N], mark[N], vis[N], Q[N];
std::vector<int> e[N];
int rear;

int LCA(int x, int y){
  static int t = 0; t++;
  while (true) {
    if (x != -1) {
      x = findb(x);
      if (vis[x] == t) return x;
      vis[x] = t;
      if (match[x] != -1) x = next[match[x]];
      else x = -1;
    }
    std::swap(x, y);
  }
}

void group(int a, int p){
    while (a != p){
      int b = match[a], c = next[b];
      if (findb(c) != p) next[c] = b;
      if (mark[b] == 2) mark[Q[rear++] = b] = 1;
      if (mark[c] == 2) mark[Q[rear++] = c] = 1;
      unit(a, b); unit(b, c);
      a = c;
    }
}

void aug(int s){
  for (int i = 0; i < n; i++)
    next[i] = -1, belong[i] = i, mark[i] = 0, vis[i] = -1;
  mark[s] = 1;
  Q[0] = s; rear = 1;
  for (int front = 0; match[s] == -1 && front < rear; front
      ++){
    int x = Q[front];
    for (int i = 0; i < (int)e[x].size(); i++){
      int y = e[x][i];
      if (match[x] == y) continue;
      if (findb(x) == findb(y)) continue;
      if (mark[y] == 2) continue;
      if (mark[y] == 1){
        int r = LCA(x, y);
        if (findb(x) != r) next[x] = y;
        if (findb(y) != r) next[y] = x;
        group(x, r);
        group(y, r);
      }
      else if (match[y] == -1){
        next[y] = x;
        for (int u = y; u != -1; ){
          int v = next[u];
          int mv = match[v];
          match[v] = u, match[u] = v;u = mv;
        }
        break;
      }
      else{
        next[y] = x;
        mark[Q[rear++] = match[y]] = 1;
        mark[y] = 2;
      }
    }
  }
}
// the graph is stored as e[N] and g[N]
// for (int i = 0; i < n; i++) match[i] = -1;
// for (int i = 0; i < n; i++) if (match[i] == -1) aug(i);
// int tot = 0;
// for (int i = 0; i < n; i++){
//   if (match[i] != -1) tot++;
// }
// //matched pairs = tot/2
// printf("%d\n", tot/2);
// for (int i = 0; i < n; i++){
//     printf("%d ", match[i] + 1);
// }
```

GeometrySnippets.cpp

f647cc, 86 lines

```
/*
They are from KACTL, KTH's Team Reference Document
*/

//check point in convex hull
typedef Point<ll> P;

bool inHull(const vector<P>& l, P p, bool strict = true) {
  int a = 1, b = sz(l) - 1, r = !strict;
  if (sz(l) < 3) return r && onSegment(l[0], l.back(), p);
  if (sideOf(l[0], l[a], l[b]) > 0) swap(a, b);
  if (sideOf(l[0], l[a], p) >= r || sideOf(l[0], l[b], p)<=
      -r)
    return false;
  while (abs(a - b) > 1) {
    int c = (a + b) / 2;
    (sideOf(l[0], l[c], p) > 0 ? b : a) = c;
  }
  return sgn(l[a].cross(l[b], p)) < r;
}

//center of mass of polygon
typedef Point<double> P;
P polygonCenter(const vector<P>& v) {
  P res(0, 0); double A = 0;
  for (int i = 0, j = sz(v) - 1; i < sz(v); j = i++) {
    res = res + (v[i] + v[j]) * v[j].cross(v[i]);
    A += v[j].cross(v[i]);
  }
  return res / A / 3;
}

// Returns a vector with the vertices of a polygon with
    everything to the left
// of the line going from s to e cut away.
ypedef Point<double> P;
vector<P> polygonCut(const vector<P>& poly, P s, P e) {
  vector<P> res;
  FOR(i,0,sz(poly)) {
    P cur = poly[i], prev = i ? poly[i-1] : poly.back();
    bool side = s.cross(e, cur) < 0;
    if (side != (s.cross(e, prev) < 0))
      res.push_back(lineInter(s, e, cur, prev).second);
    if (side)
      res.push_back(cur);
  }
  return res;
}

//volumn of polyhedron, with face outwards
template<class V, class L>
double signed_poly_volume(const V& p, const L& trilist) {
  double v = 0;
  trav(i, trilist) v += p[i.a].cross(p[i.b]).dot(p[i.c]);
  return v / 6;
}

//intersection of two lines
template<class P>
pair<int, P> lineInter(P s1, P e1, P s2, P e2) {
  auto d = (e1 - s1).cross(e2 - s2);
  if (d == 0) // if parallel
    return {-(s1.cross(e1, s2) == 0), P(0, 0)};
  auto p = s2.cross(e1, e2), q = s2.cross(e2, s1);
  return {1, (s1 * p + e1 * q) / d};
}

// Returns where $p$ is as seen from $s$ towards $e$. 1/0/-1
     $\Leftrightarrow$ left/on line/right.

template<class P>
int sideOf(P s, P e, P p) { return sgn(s.cross(e, p)); }

template<class P>
int sideOf(const P& s, const P& e, const P& p, double eps) {
  auto a = (e-s).cross(p-s);
  double l = (e-s).dist()*eps;
  return (a > l) - (a < -l);
}

//f is longitude, t is latitude
double sphericalDistance(double f1, double t1,
  double f2, double t2, double radius) {
  double dx = sin(t2)*cos(f2) - sin(t1)*cos(f1);
  double dy = sin(t2)*sin(f2) - sin(t1)*sin(f1);
  double dz = cos(t2) - cos(t1);
  double d = sqrt(dx*dx + dy*dy + dz*dz);
  return radius*2*asin(d/2);
}
```

HalfPlaneIntersection.cpp

80d545, 86 lines

```
// Basic half-plane struct.
```

```
struct Halfplane {
    // 'p' is a passing point of the line and 'pq' is the
        direction vector of the line.
    Point p, pq;
    long double angle;
    Halfplane() {}
    Halfplane(const Point& a, const Point& b) : p(a), pq(b -
        a) {
        angle = atan2l(pq.y, pq.x);
    }
    // Check if point 'r' is outside this half-plane.
    // Every half-plane allows the region to the LEFT of its
        line.
    bool out(const Point& r) {
        return cross(pq, r - p) < -eps;
    }
    // Comparator for sorting.
    bool operator < (const Halfplane& e) const {
        return angle < e.angle;
    }
    // Intersection point of the lines of two half-planes.
        It is assumed they're never parallel.
    friend Point inter(const Halfplane& s, const Halfplane&
        t) {
        long double alpha = cross((t.p - s.p), t.pq) / cross
            (s.pq, t.pq);
        return s.p + (s.pq * alpha);
    }
};
// Actual algorithm
vector<Point> hp_intersect(vector<Halfplane>& H) {
    Point box[4] = {  // Bounding box in CCW order
        Point(inf, inf),
        Point(-inf, inf),
        Point(-inf, -inf),
        Point(inf, -inf)
    };
    for(int i = 0; i<4; i++) { // Add bounding box half-
        planes.
        Halfplane aux(box[i], box[(i+1) % 4]);
        H.push_back(aux);
    }
    // Sort by angle and start algorithm
    sort(H.begin(), H.end());
    deque<Halfplane> dq;
    int len = 0;
    for(int i = 0; i < int(H.size()); i++) {
        // Remove from the back of the deque while last half
            -plane is redundant
        while (len > 1 && H[i].out(inter(dq[len-1], dq[len
            -2]))) {
            dq.pop_back();
            --len;
        }
        // Remove from the front of the deque while first
            half-plane is redundant
        while (len > 1 && H[i].out(inter(dq[0], dq[1]))) {
            dq.pop_front();
            --len;
        }
        // Special case check: Parallel half-planes
        if (len > 0 && fabsl(cross(H[i].pq, dq[len-1].pq)) <
            eps) {
            // Opposite parallel half-planes that ended up
                checked against each other.
            if (dot(H[i].pq, dq[len-1].pq) < 0.0)
                return vector<Point>();
            // Same direction half-plane: keep only the
                leftmost half-plane.
            if (H[i].out(dq[len-1].p)) {
                dq.pop_back();
                --len;
            }
            else continue;
        }
        // Add new half-plane
        dq.push_back(H[i]);
        ++len;
    }
    // Final cleanup: Check half-planes at the front against
        the back and vice-versa
    while (len > 2 && dq[0].out(inter(dq[len-1], dq[len-2]))
        ) {
        dq.pop_back();
        --len;
    }
    while (len > 2 && dq[len-1].out(inter(dq[0], dq[1]))) {
        dq.pop_front();
        --len;
    }
    // Report empty intersection if necessary
    if (len < 3) return vector<Point>();
    // Reconstruct the convex polygon from the remaining
        half-planes.
    vector<Point> ret(len);
    for(int i = 0; i+1 < len; i++) {
        ret[i] = inter(dq[i], dq[i+1]);
    }
    ret.back() = inter(dq[len-1], dq[0]);
    return ret;
}
```

HopcroftKarp.cpp

Description: Fast bipartite matching algorithm. Graph g should be a list of neighbors of the left partition, and $btoa$ should be a vector full of -1's of the same size as the right partition. Returns the size of the matching. $btoa[i]$ will be the match for vertex i on the right side, or -1 if it's not matched.

Usage: `vi btoa(m, -1); hopcroftKarp(g, btoa);`

Time: $\mathcal{O}\left(\sqrt{V}E\right)$

2ff797, 42 lines

```
bool dfs(int a, int L, vector<vi>& g, vi& btoa, vi& A, vi& B
    ) {
  if (A[a] != L) return 0;
  A[a] = -1;
  for (int b : g[a]) if (B[b] == L + 1) {
    B[b] = 0;
    if (btoa[b] == -1 || dfs(btoa[b], L + 1, g, btoa, A, B))
      return btoa[b] = a, 1;
  }
  return 0;
}

int hopcroftKarp(vector<vi>& g, vi& btoa) {
  int res = 0;
  vi A(g.size()), B(btoa.size()), cur, next;
  for (;;) {
    fill(all(A), 0);
    fill(all(B), 0);
    cur.clear();
    for (int a : btoa) if(a != -1) A[a] = -1;
    FOR(a,0,sz(g)) if(A[a] == 0) cur.push_back(a);
    for (int lay = 1;; lay++) {
      bool islast = 0;
      next.clear();
      for (int a : cur) for (int b : g[a]) {
        if (btoa[b] == -1) {
          B[b] = lay;
          islast = 1;
        }
        else if (btoa[b] != a && !B[b]) {
          B[b] = lay;
          next.push_back(btoa[b]);
        }
      }
      if (islast) break;
      if (next.empty()) return res;
      for (int a : next) A[a] = lay;
      cur.swap(next);
    }
    FOR(a,0,sz(g))
      res += dfs(a, 0, g, btoa, A, B);
  }
}
```

HullDiameter.cpp

0c6e60, 12 lines

```
typedef Point<ll> P;
array<P, 2> hullDiameter(vector<P> S) {
  int n = sz(S), j = n < 2 ? 0 : 1;
  pair<ll, array<P, 2>> res({0, {S[0], S[0]}});
  FOR(i,0,j)
    for (;; j = (j + 1) % n) {
      res = max(res, {(S[i] - S[j]).dist2(), {S[i], S[j]}});
      if ((S[(j + 1) % n] - S[j]).cross(S[i + 1] - S[i]) >=
          0)
        break;
    }
  return res.second;
}
```

InsidePolygon.cpp

"Point.h", "OnSegment.h", "SegmentDistance.h" fd40d6, 11 lines

```
template<class P>
bool inPolygon(vector<P> &p, P a, bool strict = true) {
  int cnt = 0, n = sz(p);
  FOR(i,0,n) {
    P q = p[(i + 1) % n];
    if (onSegment(p[i], q, a)) return !strict;
    //or: if (segDist(p[i], q, a) <= eps) return !strict;
    cnt ^= ((a.y<p[i].y) - (a.y<q.y)) * a.cross(p[i], q) >
        0;
  }
  return cnt;
}
```

InsidePolygonFast.cpp

96e771, 52 lines

```
bool lexComp(const pt &l, const pt &r) {
    return l.x < r.x || (l.x == r.x && l.y < r.y);
}
int sgn(long long val) { return val > 0 ? 1 : (val == 0 ? 0
    : -1); }
vector<pt> seq;
pt translation;
```

```
int n;
bool pointInTriangle(pt a, pt b, pt c, pt point) {
    long long s1 = abs(a.cross(b, c));
    long long s2 = abs(point.cross(a, b)) + abs(point.cross(
         b, c)) + abs(point.cross(c, a));
    return s1 == s2;
}
void prepare(vector<pt> &points) {
    n = points.size();
    int pos = 0;
    for (int i = 1; i < n; i++) {
        if (lexComp(points[i], points[pos]))
            pos = i;
    }
    rotate(points.begin(), points.begin() + pos, points.end
         ());

    n--;
    seq.resize(n);
    for (int i = 0; i < n; i++)
        seq[i] = points[i + 1] - points[0];
    translation = points[0];
}

bool pointInConvexPolygon(pt point) {
    point = point - translation;
    if (seq[0].cross(point) != 1 &&
            sgn(seq[0].cross(point)) != sgn(seq[0].cross(seq
                 [n - 1])))
        return false;
    if (seq[n - 1].cross(point) != 0 &&
            sgn(seq[n - 1].cross(point)) != sgn(seq[n - 1].
                 cross(seq[0])))
        return false;

    if (seq[0].cross(point) == 0)
        return seq[0].sqrLen() >= point.sqrLen();

    int l = 0, r = n - 1;
    while (r - l > 1) {
        int mid = (l + r) / 2;
        int pos = mid;
        if (seq[pos].cross(point) >= 0)
            l = mid;
        else
            r = mid;
    }
    int pos = l;
    return pointInTriangle(seq[pos], seq[pos + 1], pt(0, 0),
         point);
}
```

IntervalContainer.cpp

Description: Add and remove intervals from a set of disjoint intervals. Will merge the added interval with any overlapping intervals in the set when adding. Intervals are [inclusive, exclusive).

Time: $\mathcal{O}(\log N)$

edce47, 23 lines

```
set<pii>::iterator addInterval(set<pii>& is, int L, int R) {
  if (L == R) return is.end();
  auto it = is.lower_bound({L, R}), before = it;
  while (it != is.end() && it->first <= R) {
    R = max(R, it->second);
    before = it = is.erase(it);
  }
  if (it != is.begin() && (--it)->second >= L) {
    L = min(L, it->first);
    R = max(R, it->second);
    is.erase(it);
  }
  return is.insert(before, {L,R});
}

void removeInterval(set<pii>& is, int L, int R) {
  if (L == R) return;
  auto it = addInterval(is, L, R);
  auto r2 = it->second;
  if (it->first == L) is.erase(it);
  else (int&)it->second = L;
  if (R != r2) is.emplace(R, r2);
}
```

KdTree.cpp

Description: KD-tree (2d, can be extended to 3d)

"Point.h" bac5b0, 63 lines

```
typedef long long T;
typedef Point<T> P;
const T INF = numeric_limits<T>::max();

bool on_x(const P& a, const P& b) { return a.x < b.x; }
bool on_y(const P& a, const P& b) { return a.y < b.y; }

struct Node {
  P pt; // if this is a leaf, the single point in it
  T x0 = INF, x1 = -INF, y0 = INF, y1 = -INF; // bounds
  Node *first = 0, *second = 0;

  T distance(const P& p) { // min squared distance to a
       point
    T x = (p.x < x0 ? x0 : p.x > x1 ? x1 : p.x);
    T y = (p.y < y0 ? y0 : p.y > y1 ? y1 : p.y);
    return (P(x,y) - p).dist2();
  }

  Node(vector<P>&& vp) : pt(vp[0]) {
    for (P p : vp) {
      x0 = min(x0, p.x); x1 = max(x1, p.x);
      y0 = min(y0, p.y); y1 = max(y1, p.y);
    }
    if (vp.size() > 1) {
      // split on x if width >= height (not ideal...)
      sort(all(vp), x1 - x0 >= y1 - y0 ? on_x : on_y);
      // divide by taking half the array for each child (not
      // best performance with many duplicates in the middle
           )
      int half = sz(vp)/2;
      first = new Node({vp.begin(), vp.begin() + half});
      second = new Node({vp.begin() + half, vp.end()});
    }
  }
};

struct KDTree {
  Node* root;
  KDTree(const vector<P>& vp) : root(new Node({all(vp)})) {}

  pair<T, P> search(Node *node, const P& p) {
    if (!node->first) {
      // uncomment if we should not find the point itself:
      // if (p == node->pt) return {INF, P()};
      return make_pair((p - node->pt).dist2(), node->pt);
    }

    Node *f = node->first, *s = node->second;
    T bfirst = f->distance(p), bsec = s->distance(p);
    if (bfirst > bsec) swap(bsec, bfirst), swap(f, s);

    // search closest side first, other side if needed
    auto best = search(f, p);
    if (bsec < best.first)
      best = min(best, search(s, p));
    return best;
  }

  // find nearest point to a point, and its squared distance
  // (requires an arbitrary operator< for Point)
  pair<T, P> nearest(const P& p) {
    return search(root, p);
  }
};
```

KMP.cpp

7ef32f, 13 lines

```
vector<int> prefix_function(string s) {
    int n = sz(s);
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
```

KnuthOptimization.cpp

a5b59d, 38 lines

```
/*Class1 : Interval DP: f_{l,r} = min_{k=l}^{r-1} f_{l,k}+f_
     {k+1,r} + w(l,r)
weights w(l,r) satisfying the following inequality:
(1) For any l <= l'<= r'<= r, we have w(l',r') <= w(l,r).
(2) (The important one): For any l1 <= l2 <= r1 <= r2, we
     have
w(l1,r1) + w(l2,r2) <= w(l1,r2) + w(l2,r1).
*/

for (int len = 2; len <= n; ++len)  // Enumerate Interval
     Length
  for (int l = 1, r = len; r <= n; ++l, ++r) {
    // Enumerate Intervals of Length Len
    f[l][r] = INF;
    for (int k = m[l][r - 1]; k <= m[l + 1][r]; ++k)
      if (f[l][r] > f[l][k] + f[k + 1][r] + w(l, r)) {
        f[l][r] = f[l][k] + f[k + 1][r] + w(l, r);  //Update
             DP
        m[l][r] = k;  // Update Decision Point
      }
  }

/*Class2: 2D DP, f_{i,j} = min_{k <= j}{f_{i-1,k}} + w(k,j)
Where 1 <= i <= n, 1 <= j <= m
*/
int n;
```

```
long long C(int i, int j);
vector<long long> dp_before(n), dp_cur(n);
// compute dp_cur[l], ... dp_cur[r] (inclusive)
//Call compute for each possible i.
void compute(int l, int r, int optl, int optr) {
  if (l > r) return;
  int mid = (l + r) >> 1;
  pair<long long, int> best = {INF, -1};
  for (int k = optl; k <= min(mid, optr); k++) {
    best = min(best, {dp_before[k] + C(k, mid), k});
  }
  dp_cur[mid] = best.first;
  int opt = best.second;
  compute(l, mid - 1, optl, opt);
  compute(mid + 1, r, opt, optr);
}
```

LazySegtree.cpp

8ed0ff, 45 lines

```
const ll identity = 0;
const ll SZ = 131072;

ll sum[2*SZ], lazy[2*SZ];

ll combine(ll A, ll B) {
    return A+B;
}

ll combineUpd(ll A, ll B) {
    return A+B;
}

void push(int index, ll L, ll R) {
    sum[index] = combineUpd(sum[index], lazy[index]);
    if (L != R) lazy[2*index] = combineUpd(lazy[2*index],
         lazy[index]), lazy[2*index+1] = combineUpd(lazy[2*
         index+1], lazy[index]);
    lazy[index] = identity;
}

void pull(int index) {
    sum[index] = combine(sum[2*index], sum[2*index+1]);
}

ll query(int lo, int hi, int index = 1, ll L = 0, ll R = SZ
     -1) {
    push(index, L, R);
    if (lo > R || L > hi) return identity;
    if (lo <= L && R <= hi) return sum[index];

    int M = (L+R) / 2;
    return combine(query(lo, hi, 2*index, L, M), query(lo,
         hi, 2*index+1, M+1, R));
}

void update(int lo, int hi, ll increase, int index = 1, ll L
      = 0, ll R = SZ-1) {
    push(index, L, R);
    if (hi < L || R < lo) return;
    if (lo <= L && R <= hi) {
        lazy[index] = increase;
        push(index, L, R);
        return;
    }
    int M = (L+R) / 2;
    update(lo, hi, increase, 2*index, L, M); update(lo, hi,
         increase, 2*index+1, M+1, R);
    pull(index);
}
```

LCA.cpp

e1efce, 52 lines

```
const int L; //SET THIS TO CEIL(LOG(MX_N))
int N;
int anc[MX][L];
int dep[MX];
vector<vi> graph(MX);

int jmp(int x, int d) {
    FOR(i, L) {
        if (d&(1<<i)) {
            x = anc[x][i];
        }
    }
    return x;
}

int lca(int a, int b) {
    if (dep[a] < dep[b]) {
        swap(a, b);
    }

    a = jmp(a, dep[a] - dep[b]);
    if (a == b) return a;
    FORd(i, L) {
        if (anc[a][i] != anc[b][i]) {
            a = anc[a][i];
            b = anc[b][i];
        }
    }
    return anc[a][0];
}

void dfs(int v, int p) {
    anc[v][0] = p;
    trav(a, graph[v]) {
        if (a == p) continue;
        dep[a] = dep[v] + 1;
        dfs(a, v);
    }
}

void prep() {
    FOR(i, N) FOR(j, L) anc[i][j] = -1;
    dep[0] = 0;
    dfs(0, -1);
    FOR(j, 1, L) {
        FOR(i, N) {
            if (anc[i][j-1] != -1) {
                anc[i][j] = anc[anc[i][j-1]][j-1];
            }
        }
    }
}
```

LCT.cpp

1809e5, 116 lines

```
struct rec
{
    int ls, rs, p; //ls = left son; rs = right son; p =
         parent
    uint siz; //siz = size of the subtree
    uint key, sum; //sum: sum of weights in the subtree
    uint mult, add; //two lazy tags
    bool rev; //denote whether this segment has been
         reverted
};
rec splay[maxn];
void clear(){
    splay[0].p = splay[0].ls = splay[0].rs = splay[0].rev =
         splay[0].key = splay[0].sum = 0;
    splay[0].siz = 0;
}
 void update(int x){
    clear();
    splay[x].sum = splay[splay[x].ls].sum + splay[splay[x].
         rs].sum + splay[x].key;
    splay[x].sum %= modi;
    splay[x].siz = splay[splay[x].ls].siz + splay[splay[x].
         rs].siz + 1;
    splay[x].siz %= modi;
}
void zig(int x){
    int y = splay[x].p, z = splay[y].p;
    if (y == splay[z].ls) splay[z].ls = x;
    else if (y == splay[z].rs) splay[z].rs = x;
    splay[x].p = z;
    // Switch ls and rs for zag.
    if (splay[x].rs) splay[splay[x].rs].p = y;
    splay[y].ls = splay[x].rs;
    splay[x].rs = y;
    splay[y].p = x;
    update(y);
}
bool is_root(int x){
    return x != splay[splay[x].p].ls && x != splay[splay[x].
         p].rs;
}
void rev(int x){
    if (!x)return;
    swap(splay[x].ls, splay[x].rs);
    splay[x].rev ^= true;
}
void pushdown(int x){
    if (splay[x].rev){
        rev(splay[x].ls);
        rev(splay[x].rs);
        splay[x].rev = false;
    }
    //Todo: Push lazy tags here.
}
void set_root(int x){
    static int q[maxn];
    static int top;
    int i;
    for (i = x; !is_root(i); i = splay[i].p){
        q[++top] = i;
    }
    q[++top] = i;
    while (top){
        pushdown(q[top--]);
    }
    while (!is_root(x)){
        int y = splay[x].p;
```

```
        if (is_root(y)){
            if (x == splay[y].ls) zig(x); else zag(x);
        }
        else{
            int z = splay[y].p;
            if (y == splay[z].ls){
                if (x == splay[y].ls) zig(y), zig(x);
                else zag(x), zig(x);
            }
            else{
                if (x == splay[y].rs) zag(y), zag(x);
                else zig(x), zag(x);
            }
        }
    }
    update(x);
}
//this is a special operation on LCT
void access(int x)
{
    for (int t = 0; x; t = x, x = splay[x].p){
        set_root(x);
        splay[x].rs = t;
        update(x);
    }
}
//we will make x be the new root of the tree it belongs to
void makeroot(int x){access(x);set_root(x);rev(x);}
void split(int x, int y){makeroot(x);access(y);set_root(y);}
//link vertex x and vertex y
void link(int x, int y){makeroot(x);makeroot(y);splay[x].p = y;}

//cut the edge between x and y
void cut(int x, int y){
    split(x, y);
    splay[y].ls = splay[x].p = 0;
    update(y);
}
//find the root; x connected with y IFF findroot(x) = findroot(y)
int findroot(int x){
    access(x);
    set_root(x);
    while (splay[x].ls){
        pushdown(x);
        x = splay[x].ls;
    }
    set_root(x);
    return x;
}
//Adding edge between u and v: link(u, v);
//Removing edge between u and v: cut(u1, v1);
//Adding vertices on route between u and v by c :
/* split(u, v);
   calc(v, 1, c);*/
//Query the sum on route from u to v: split(u1,v1) print(splay[v1].sum);
```

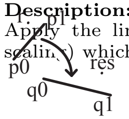
LeftistTree.cpp

7d92ca, 21 lines

```
struct node{
    node *l,*r;
    //key is the priority
    int key,id;
    //distanct to the leftist child - it is used to maintain the properties of the lefitst tree
    int dist;
    int rdist(){return (r==NULL)?0:r->dist;}
    int ldist(){return (l==NULL)?0:l->dist;}
};
node* merge(node*l,node*r)
{
    if (l == NULL) return r;
    if (r == NULL) return l;
    //we want to make sure the root has the smallest key
    if (l->key > r->key) swap(l,r);
    l->r = merge(l->r,r);
    //maintain the properties of the leftist tree
    if (l->ldist() < l->rdist()) swap(l->l,l->r);
    l->dist = l->rdist()+1;
    return l;
}
```

LinearTransformation.cpp

Description: Apply the linear transformation (translation, rotation and scaling) which takes line p0-p1 to line q0-q1 to point r.



"Point.h" 03a306, 6 lines

```
typedef Point<double> P;
P linearTransformation(const P& p0, const P& p1,
    const P& q0, const P& q1, const P& r) {
  P dp = p1-p0, dq = q1-q0, num(dp.cross(dq), dp.dot(dq));
  return q0 + P((r-p0).cross(num), (r-p0).dot(num))/dp.dist2();
}
```

LineDistance.cpp

"Point.h" f6bf6b, 7 lines

```
/*
Returns the signed distance between point p and the line containing points a and b. Positive value on left side and negative on right as seen from a towards b. a==b gives nan. P is supposed to be Point<T> or Point3D<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long. Using Point3D will always give a non-negative distance. For Point3D, call .dist on the result of the cross product.
*/
template<class P>
double lineDist(const P& a, const P& b, const P& p) {
  return (double)(b-a).cross(p-a)/(b-a).dist();
}
```

LineHullIntersection.cpp

Time: $\mathcal{O}(N + Q\log n)$

"Point.h" f78f76, 39 lines

```
typedef array<P, 2> Line;
#define cmp(i,j) sgn(dir.perp().cross(poly[(i)%n]-poly[(j)%n]))
#define extr(i) cmp(i + 1, i) >= 0 && cmp(i, i - 1 + n) < 0
int extrVertex(vector<P>& poly, P dir) {
  int n = sz(poly), lo = 0, hi = n;
  if (extr(0)) return 0;
  while (lo + 1 < hi) {
    int m = (lo + hi) / 2;
    if (extr(m)) return m;
    int ls = cmp(lo + 1, lo), ms = cmp(m + 1, m);
    (ls < ms || (ls == ms && ls == cmp(lo, m)) ? hi : lo) = m;
  }
  return lo;
}

#define cmpL(i) sgn(line[0].cross(poly[i], line[1]))
array<int, 2> lineHull(Line line, vector<P> poly) {
  int endA = extrVertex(poly, (line[0] - line[1]).perp());
  int endB = extrVertex(poly, (line[1] - line[0]).perp());
  if (cmpL(endA) < 0 || cmpL(endB) > 0)
    return {-1, -1};
  array<int, 2> res;
  FOR(i,0,2) {
    int lo = endB, hi = endA, n = sz(poly);
    while ((lo + 1) % n != hi) {
      int m = ((lo + hi + (lo < hi ? 0 : n)) / 2) % n;
      (cmpL(m) == cmpL(endB) ? lo : hi) = m;
    }
    res[i] = (lo + !cmpL(hi)) % n;
    swap(endA, endB);
  }
  if (res[0] == res[1]) return {res[0], -1};
  if (!cmpL(res[0]) && !cmpL(res[1]))
    switch ((res[0] - res[1] + sz(poly) + 1) % sz(poly)) {
      case 0: return {res[0], res[0]};
      case 2: return {res[1], res[1]};
    }
  return res;
}
```

LineContainer.cpp

8ec1c7, 34 lines

```
/* Author: KACTL Line Container
 * Description: Container where you can add lines of the form kx+m, and query maximum values at point x.
 *  Useful for dynamic programming (``convex hull trick'').
 * Time: O(\log N)*/
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<>> {
  // (for doubles, use inf = 1/.0, div(a,b) = a/b)
  static const ll inf = LLONG_MAX;
  ll div(ll a, ll b) { // floored division
    return a / b - ((a ^ b) < 0 && a % b); }
  bool isect(iterator x, iterator y) {
    if (y == end()) return x->p = inf, 0;
    if (x->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;
  }
};
```

Manacher.cpp

2fddb0, 28 lines

```
vector<int> manacher_odd(string s) {
  int n = s.size();
  s = "$" + s + "^";
  vector<int> p(n + 2);
  int l = 1, r = 1;
  for(int i = 1; i <= n; i++) {
    p[i] = max(0, min(r - i, p[l + (r - i)]));
    while(s[i - p[i]] == s[i + p[i]]) {
      p[i]++;
    }
    if(i + p[i] > r) {
      l = i - p[i], r = i + p[i];
    }
  }
  return vector<int>(begin(p) + 1, end(p) - 1);
}

vector<int> manacher(string s) {
  string t;
  for(auto c: s) {
    t += string("#") + c;
  }
  auto res = manacher_odd(t + "#");
  for (auto& x: res) x--;
  return vector<int>(begin(res) + 1, end(res) - 1);
}
// returns array P of length 2N-1, p[i] = length of longest
    odd/even palindrome
// abcbcba: 1 0 1 0 3 0 7 0 3 0 1 0 1
```

MinCostMaxFlow.cpp

Description: Min-cost max-flow. cap[i][j] != cap[j][i] is allowed; double edges are not. If costs can be negative, call setpi before maxflow, but note that negative cost cycles are not supported. To obtain the actual flow, look at positive values only.
Time: Approximately $\mathcal{O}\left(E^2\right)$

<ext/pb_ds/priority_queue.hpp> 45af93, 80 lines

```
const ll INF = numeric_limits<ll>::max() / 4;
typedef vector<ll> VL;

using pii = pair<int, int>;
struct MCMF {
  int N;
  vector<vi> ed, red;
  vector<VL> cap, flow, cost;
  vi seen;
  VL dist, pi;
  vector<pii> par;

  MCMF(int N) :
    N(N), ed(N), red(N), cap(N, VL(N)), flow(cap), cost(cap)
      ,
    seen(N), dist(N), pi(N), par(N) {}

  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 (int i : ed[s]) if (!seen[i])
        relax(i, cap[s][i] - flow[s][i], cost[s][i], 1);
      for (int i : red[s]) if (!seen[i])
        relax(i, flow[i][s], -cost[i][s], 0);
    }
    FOR(i,0,N) 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(i,0,N) FOR(j,0,N) totcost += cost[i][j] * flow[i][j
        ];
    return {totflow, totcost};
  }

  // If some costs can be negative, call this before maxflow
      :
  void setpi(int s) { // (otherwise, leave this out)
    fill(all(pi), INF); pi[s] = 0;
    int it = N, ch = 1; ll v;
    while (ch-- && it--)
      FOR(i,0,N) if (pi[i] != INF)
        for (int 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
  }
};
```

MinCostMaxFlowPR.cpp

9f8191, 160 lines

```
// Push-Relabel implementation of the cost-scaling algorithm
// Runs in O( <max_flow> * log(V * max_edge_cost)) = O( V^3
    * log(V * C))
// Operates on integers

template<typename flow_t = int, typename cost_t = int>
struct mcSFlow{
    struct Edge{
        cost_t c;
        flow_t f;
        int to, rev;
        Edge(int _to, cost_t _c, flow_t _f, int _rev):c(_c),
            f(_f), to(_to), rev(_rev){}
    };
    const cost_t INFCOST = numeric_limits<cost_t>::max()/2;
    const cost_t INFFLOW = numeric_limits<flow_t>::max()/2;
    cost_t epsilon;
    int N, S, T;
    vector<vector<Edge> > G;
    vector<unsigned int> isEnqueued, state;
    mcSFlow(int _N, int _S, int _T):epsilon(0), N(_N), S(_S)
        , T(_T), G(_N){}
    void add_edge(int a, int b, cost_t cost, flow_t cap){
        if(a==b){assert(cost>=0); return;}
        cost*=N;// to preserve integer-values
        epsilon = max(epsilon, abs(cost));
        assert(a>=0&&a<N&&b>=0&&b<N);
        G[a].emplace_back(b, cost, cap, G[b].size());
        G[b].emplace_back(a, -cost, 0, G[a].size()-1);
    }
    flow_t calc_max_flow(){ // Dinic max-flow
        vector<flow_t> dist(N), state(N);
        vector<Edge*> path(N);
        auto cmp = [](Edge*a, Edge*b){return a->f < b->f;};
        flow_t addFlow, retflow=0;;
        do{
            fill(dist.begin(), dist.end(), -1);
            dist[S]=0;
            auto head = state.begin(), tail = state.begin();
            for(*tail++ = S;head!=tail;++head){
                for(Edge const&e:G[*head]){
                    if(e.f && dist[e.to]==-1){
                        dist[e.to] = dist[*head]+1;
                        *tail++=e.to;
                    }
                }
            }
            addFlow = 0;
            fill(state.begin(), state.end(), 0);
            auto top = path.begin();
            Edge dummy(S, 0, INFFLOW, -1);
            *top++ = &dummy;
            while(top != path.begin()){
                int n = (*prev(top))->to;
                if(n==T){
                    auto next_top = min_element(path.begin()
                        , top, cmp);
                    flow_t flow = (*next_top)->f;
                    while(--top!=path.begin()){
                        Edge &e=**top, &f=G[e.to][e.rev];
```

```
                e.f-=flow;
                f.f+=flow;
            }
            addFlow=1;
            retflow+=flow;
            top = next_top;
            continue;
        }
        for(int &i=state[n], i_max = G[n].size(),
             need = dist[n]+1;;++i){
            if(i==i_max){
                dist[n]=-1;
                --top;
                break;
            }
            if(dist[G[n][i].to] == need && G[n][i].f
                 ){
                *top++ = &G[n][i];
                break;
            }
        }
    }
}while(addFlow);
return retflow;
}
vector<flow_t> excess;
vector<cost_t> h;
void push(Edge &e, flow_t amt){
    if(e.f < amt) amt=e.f;
    e.f-=amt;
    excess[e.to]+=amt;
    G[e.to][e.rev].f+=amt;
    excess[G[e.to][e.rev].to]-=amt;
}
void relabel(int vertex){
    cost_t newHeight = -INFCOST;
    for(unsigned int i=0;i<G[vertex].size();++i){
        Edge const&e = G[vertex][i];
        if(e.f && newHeight < h[e.to]-e.c){
            newHeight = h[e.to] - e.c;
            state[vertex] = i;
        }
    }
    h[vertex] = newHeight - epsilon;
}
const int scale=2;
pair<flow_t, cost_t> minCostFlow(){
    cost_t retCost = 0;
    for(int i=0;i<N;++i){
        for(Edge &e:G[i]){
            retCost += e.c*(e.f);
        }
    }
    //find feasible flow
    flow_t retFlow = calc_max_flow();
    excess.resize(N);h.resize(N);
    queue<int> q;
    isEnqueued.assign(N, 0); state.assign(N,0);
    for(;epsilon;epsilon>>=scale){
        //refine
        fill(state.begin(), state.end(), 0);
        for(int i=0;i<N;++i)
            for(auto &e:G[i])
                if(h[i] + e.c - h[e.to] < 0 && e.f) push
                     (e, e.f);
        for(int i=0;i<N;++i){
            if(excess[i]>0){
                q.push(i);
                isEnqueued[i]=1;
            }
        }
        while(!q.empty()){
            int cur=q.front();q.pop();
            isEnqueued[cur]=0;
            // discharge
            while(excess[cur]>0){
                if(state[cur] == G[cur].size()){
                    relabel(cur);
                }
                for(unsigned int &i=state[cur], max_i =
                     G[cur].size();i<max_i;++i){
                    Edge &e=G[cur][i];
                    if(h[cur] + e.c - h[e.to] < 0){
                        push(e, excess[cur]);
                        if(excess[e.to]>0 && isEnqueued[
                             e.to]==0){
                            q.push(e.to);
                            isEnqueued[e.to]=1;
                        }
                        if(excess[cur]==0) break;
                    }
                }
            }
        }
        if(epsilon>1 && epsilon>>scale==0){
            epsilon = 1<<scale;
        }
    }
    for(int i=0;i<N;++i){
        for(Edge &e:G[i]){
            retCost -= e.c*(e.f);
        }
    }
    return make_pair(retFlow, retCost/2/N);
}
flow_t getFlow(Edge const &e){
    return G[e.to][e.rev].f;
}
};
```

MinimumEnclosingCircle.cpp

6d8e96, 44 lines

```
point a[maxn];

void getCenter2(point a,point b,point & c)
{
    c.x = (a.x+b.x)/2;
    c.y = (a.y+b.y)/2;
}

void getCenter3(point a,point b,point c,point &d)
{
    double a1 = b.x-a.x, b1 = b.y-a.y, c1 = (a1*a1+b1*b1)/2;
    double a2 = c.x-a.x, b2 = c.y-a.y, c2 = (a2*a2+b2*b2)/2;
    double de = (a1 * b2 - b1 * a2);
    d.x = a.x + (c1 * b2 - c2 * b1)/de;
    d.y = a.y + (a1 * c2 - a2 * c1)/de;
}

    //randomP_shuffle before using
    radius = 0;
    center = a[0];
    for (int i = 1;i < n;++i)
    {
        if (!isIn(a[i], center, radius))
        {
            radius = 0;
            center = a[i];
            for (int j = 0;j < i;++j)
                if (!isIn(a[j], center, radius))
                {
                    getCenter2(a[i], a[j], center);
                    radius = dis(a[i],center);
                    for (int k = 0;k < j;++k) if (!isIn(a[k
                        ], center, radius))
                    {
                        getCenter3(a[i], a[j], a[k], center)
                            ;
                        radius = dis(a[k],center);
                    }
                }
        }
    }
    //printf("%.2lf\n%.2lf %.2lf\n",radius,center.x,center.y
        );
    printf("%.3lf\n",radius);
    return 0;
}
```

MinimumVertexCover.cpp

Description: Finds a minimum vertex cover in a bipartite graph. The size is the same as the size of a maximum matching, and the complement is a maximum independent set.

"DFSMatching.h" 8c9aed, 20 lines

```
vi cover(vector<vi>& g, int n, int m) {
  vi match(m, -1);
  int res = dfsMatching(g, match);
  vector<bool> lfound(n, true), seen(m);
  for (int it : match) if (it != -1) lfound[it] = false;
  vi q, cover;
  FOR(i,0,n) if (lfound[i]) q.push_back(i);
  while (!q.empty()) {
    int i = q.back(); q.pop_back();
    lfound[i] = 1;
    for (int e : g[i]) if (!seen[e] && match[e] != -1) {
      seen[e] = true;
      q.push_back(match[e]);
    }
  }
  FOR(i,0,n) if (!lfound[i]) cover.push_back(i);
  FOR(i,0,m) if (seen[i]) cover.push_back(n+i);
  assert(sz(cover) == res);
  return cover;
}
```

MinkowskiSum.cpp

18ebb1, 31 lines

```
void reorder_polygon(vector<pt> & P){
    size_t pos = 0;
    for(size_t i = 1; i < P.size(); i++){
        if(P[i].y < P[pos].y || (P[i].y == P[pos].y && P[i].
             x < P[pos].x))
            pos = i;
    }
    rotate(P.begin(), P.begin() + pos, P.end());
```

```
}

vector<pt> minkowski(vector<pt> P, vector<pt> Q){
    // the first vertex must be the lowest
    reorder_polygon(P);
    reorder_polygon(Q);
    // we must ensure cyclic indexing
    P.push_back(P[0]);
    P.push_back(P[1]);
    Q.push_back(Q[0]);
    Q.push_back(Q[1]);
    // main part
    vector<pt> result;
    size_t i = 0, j = 0;
    while(i < P.size() - 2 || j < Q.size() - 2){
        result.push_back(P[i] + Q[j]);
        auto cross = (P[i + 1] - P[i]).cross(Q[j + 1] - Q[j
            ]);
        if(cross >= 0)
            ++i;
        if(cross <= 0)
            ++j;
    }
    return result;
}
```

Mo.cpp

aab82d, 45 lines

```
void remove(idx);  // TODO: remove value at idx from data
    structure
void add(idx);     // TODO: add value at idx from data
    structure
int get_answer();  // TODO: extract the current answer of
    the data structure

int block_size;

struct Query {
    int l, r, idx;
    bool operator<(Query other) const
    {
        return make_pair(l / block_size, r) <
               make_pair(other.l / block_size, other.r);
    }
};

vector<int> mo_s_algorithm(vector<Query> queries) {
    vector<int> answers(queries.size());
    sort(queries.begin(), queries.end());

    // TODO: initialize data structure

    int cur_l = 0;
    int cur_r = -1;
    // invariant: data structure will always reflect the
        range [cur_l, cur_r]
    for (Query q : queries) {
        while (cur_l > q.l) {
            cur_l--;
            add(cur_l);
        }
        while (cur_r < q.r) {
            cur_r++;
            add(cur_r);
        }
        while (cur_l < q.l) {
            remove(cur_l);
            cur_l++;
        }
        while (cur_r > q.r) {
            remove(cur_r);
            cur_r--;
        }
        answers[q.idx] = get_answer();
    }
    return answers;
}
```

NTT.cpp

f921a6, 38 lines

```
const ll MOD = (119 << 23) + 1, root = 62; // = 998244353
// For p < 2^30 there is also e.g. 5 << 25, 7 << 26, 479 <<
    21
// and 483 << 21 (same root). The last two are > 10^9.
ll modExp(ll a, ll b) {
  ll res = 1;
  for (; b; a = (a * a) % MOD, b >>= 1)
    if (b & 1) res = (res * a) % MOD;
  return res;
}
void ntt(vl &a) {
  int n = sz(a), L = 31 - __builtin_clz(n);
  static vl rt(2, 1);
  for (static int k = 2, s = 2; k < n; k *= 2, s++) {
    rt.resize(n);
    ll z[] = {1, modExp(root, MOD >> s)};
    FOR(i,k,2*k) rt[i] = rt[i / 2] * z[i & 1] % MOD;
  }
  vi rev(n);
  FOR(i,0,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  FOR(i,0,n) 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(j,k) {
      ll z = rt[j + k] * a[i + j + k] % MOD, &ai = a[i + j];
      a[i + j + k] = ai - z + (z > ai ? MOD: 0);
      ai += (ai + z >= MOD ? z - MOD : z);
    }
}
vl conv(const vl &a, const vl &b) {
  if (a.empty() || b.empty()) return {};
  int s = sz(a) + sz(b) - 1, B = 32 - __builtin_clz(s), n =
      1 << B;
  int inv = modExp(n, MOD - 2);
  vl L(a), R(b), out(n);
  L.resize(n), R.resize(n);
  ntt(L), ntt(R);
  FOR(i,n) out[-i & (n - 1)] = (ll)L[i] * R[i] % MOD * inv %
      MOD;
  ntt(out);
  return {out.begin(), out.begin() + s};
}
```

orderedset.cpp

<ext/pb_ds/assoc_container.hpp> 647225, 6 lines

```
using namespace __gnu_pbds;

template<typename T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;
/* order_of_key(k), how many elements < k */
/* find_by_order(k), k'th element (from 0) */
```

PlanarPointLocation.cpp

2512ba, 199 lines

```
/*
 * CP algorithm point point_location
 * This implementation assumes that the subdivision is
       correctly stored inside a DCEL
   and the outer face is numbered -1.
   For each query a pair (1,i) is returned if point is
        strictly inside face number i.
   (0,i) returned if point lies on the edge number i.
*/

bool edge_cmp(Edge* edge1, Edge* edge2)
{
    const pt a = edge1->l, b = edge1->r;
    const pt c = edge2->l, d = edge2->r;
    int val = sgn(a.cross(b, c)) + sgn(a.cross(b, d));
    if (val != 0)
        return val > 0;
    val = sgn(c.cross(d, a)) + sgn(c.cross(d, b));
    return val < 0;
}

enum EventType { DEL = 2, ADD = 3, GET = 1, VERT = 0 };

struct Event {
    EventType type;
    int pos;
    bool operator<(const Event& event) const { return type <
         event.type; }
};

vector<Edge*> sweepline(vector<Edge*> planar, vector<pt>
    queries)
{
    using pt_type = decltype(pt::x);

    // collect all x-coordinates
    auto s =
        set<pt_type, std::function<bool(const pt_type&,
            const pt_type&)>>(lt);
    for (pt p : queries)
        s.insert(p.x);
    for (Edge* e : planar) {
        s.insert(e->l.x);
        s.insert(e->r.x);
    }

    // map all x-coordinates to ids
    int cid = 0;
    auto id =
        map<pt_type, int, std::function<bool(const pt_type&,
            const pt_type&)>>(
          lt);
    for (auto x : s)
        id[x] = cid++;

    // create events
    auto t = set<Edge*, decltype(*edge_cmp)>(edge_cmp);
    auto vert_cmp = [](const pair<pt_type, int>& l,
                       const pair<pt_type, int>& r) {
        if (!eq(l.first, r.first))
            return lt(l.first, r.first);
```

```
        return l.second < r.second;
    };
    auto vert = set<pair<pt_type, int>, decltype(vert_cmp)>(
        vert_cmp);
    vector<vector<Event>> events(cid);
    for (int i = 0; i < (int)queries.size(); i++) {
        int x = id[queries[i].x];
        events[x].push_back(Event{GET, i});
    }
    for (int i = 0; i < (int)planar.size(); i++) {
        int lx = id[planar[i]->l.x], rx = id[planar[i]->r.x
            ];
        if (lx > rx) {
            swap(lx, rx);
            swap(planar[i]->l, planar[i]->r);
        }
        if (lx == rx) {
            events[lx].push_back(Event{VERT, i});
        } else {
            events[lx].push_back(Event{ADD, i});
            events[rx].push_back(Event{DEL, i});
        }
    }

    // perform sweep line algorithm
    vector<Edge*> ans(queries.size(), nullptr);
    for (int x = 0; x < cid; x++) {
        sort(events[x].begin(), events[x].end());
        vert.clear();
        for (Event event : events[x]) {
            if (event.type == DEL) {
                t.erase(planar[event.pos]);
            }
            if (event.type == VERT) {
                vert.insert(make_pair(
                    min(planar[event.pos]->l.y, planar[event
                        .pos]->r.y),
                    event.pos));
            }
            if (event.type == ADD) {
                t.insert(planar[event.pos]);
            }
            if (event.type == GET) {
                auto jt = vert.upper_bound(
                    make_pair(queries[event.pos].y, planar.
                        size()));
                if (jt != vert.begin()) {
                    --jt;
                    int i = jt->second;
                    if (ge(max(planar[i]->l.y, planar[i]->r.
                        y),
                          queries[event.pos].y)) {
                        ans[event.pos] = planar[i];
                        continue;
                    }
                }
                Edge* e = new Edge;
                e->l = e->r = queries[event.pos];
                auto it = t.upper_bound(e);
                if (it != t.begin())
                    ans[event.pos] = *(--it);
                delete e;
            }
        }

        for (Event event : events[x]) {
            if (event.type != GET)
                continue;
            if (ans[event.pos] != nullptr &&
                eq(ans[event.pos]->l.x, ans[event.pos]->r.x)
                    )
                continue;

            Edge* e = new Edge;
            e->l = e->r = queries[event.pos];
            auto it = t.upper_bound(e);
            delete e;
            if (it == t.begin())
                e = nullptr;
            else
                e = *(--it);
            if (ans[event.pos] == nullptr) {
                ans[event.pos] = e;
                continue;
            }
            if (e == nullptr)
                continue;
            if (e == ans[event.pos])
                continue;
            if (id[ans[event.pos]->r.x] == x) {
                if (id[e->l.x] == x) {
                    if (gt(e->l.y, ans[event.pos]->r.y))
                        ans[event.pos] = e;
                }
            } else {
                ans[event.pos] = e;
            }
        }
    }
    return ans;
}

struct DCEL {
    struct Edge {
        pt origin;
        Edge* nxt = nullptr;
        Edge* twin = nullptr;
        int face;
    };
    vector<Edge*> body;
};

vector<pair<int, int>> point_location(DCEL planar, vector<pt
    > queries)
{
    vector<pair<int, int>> ans(queries.size());
    vector<Edge*> planar2;
    map<intptr_t, int> pos;
    map<intptr_t, int> added_on;
    int n = planar.body.size();
    for (int i = 0; i < n; i++) {
        if (planar.body[i]->face > planar.body[i]->twin->
            face)
            continue;
        Edge* e = new Edge;
        e->l = planar.body[i]->origin;
        e->r = planar.body[i]->twin->origin;
        added_on[(intptr_t)e] = i;
        pos[(intptr_t)e] =
            lt(planar.body[i]->origin.x, planar.body[i]->
                twin->origin.x)
                ? planar.body[i]->face
                : planar.body[i]->twin->face;
        planar2.push_back(e);
    }
    auto res = sweepline(planar2, queries);
    for (int i = 0; i < (int)queries.size(); i++) {
        if (res[i] == nullptr) {
            ans[i] = make_pair(1, -1);
            continue;
        }
        pt p = queries[i];
        pt l = res[i]->l, r = res[i]->r;
        if (eq(p.cross(l, r), 0) && le(p.dot(l, r), 0)) {
            ans[i] = make_pair(0, added_on[(intptr_t)res[i
                ]]);
            continue;
        }
        ans[i] = make_pair(1, pos[(intptr_t)res[i]]);
    }
    for (auto e : planar2)
        delete e;
    return ans;
}
```

PersistentSegtree.cpp

ed1804, 86 lines

```
//Define the node of a persistent segment tree
struct node{
    int l,r,sum;
};
//the persistent segment tree. Warning: Check memory limit
     before using persistent segment tree!
node tree[maxn*32];
//Storing the root of versions of segment tree
int head[maxn];
//allocate next position. You can implement in a way that
     support garbage collection.
int nextPos(){
    static int ct;return ++ct;
}

//Building the first version of our segmetn tree
void build(int cur,int l,int r){
  tree[cur].sum = 0;
  tree[cur].l = nextPos();
  tree[cur].r = nextPos();
  if (l == r){
    tree[cur].l = tree[cur].r = 0;
  }
  else{
    int mid = (l+r)>>1;
    build(tree[cur].l,l,mid);
    build(tree[cur].r,mid+1,r);
  }
}

//This function is: currently we are at node cur, which is a
      node in the latest version of segment tree
//we want to make modifications based on some past segment
     tree, and the corresponding node in the last version is
     at last
//we want to add 1 at position key
void modify(int cur,int last,int key,int l,int r){
```

```
  //this is creating the node for our latest version
  tree[cur].sum = tree[last].sum;
  tree[cur].l = nextPos();
  tree[cur].r = nextPos();
  if (l == r){
    //base case:add on current version of our segment tree
    tree[cur].sum++;
    tree[cur].l = tree[cur].r = 0;
  }
  else{
    int mid = (l+r)>>1;
    if (key <= mid){
      //we are going to modify in the left part, so we can
          reuse the right child
      tree[cur].r = tree[last].r;
      modify(tree[cur].l, tree[last].l, key,l, mid);
      //update information for the current version of
          segment tree
      tree[cur].sum++;
    }
    else
    {
      tree[cur].l = tree[last].l;
      modify(tree[cur].r, tree[last].r, key,mid+1, r);
      tree[cur].sum++;
    }
  }
}

int query(int cur,int last,int l,int r,int k){
  if (l == r) return l;
  int mid = (l+r) >> 1;
  //notice the subtraction here - we want tgo see the
       dfifference between today's version and old versions.
  int ct = tree[tree[cur].l].sum - tree[tree[last].l].sum ;
  //if there are to many larger than mid, the k-th element
       should be in the left
  if (ct >= k){
    return query(tree[cur].l,tree[last].l,l,mid,k);
  }
  //otherwise,the k-th element should be in the right
  else{
    return query(tree[cur].r, tree[last].r, mid+1, r, k-ct);
  }
}

//Build segment tree to support queres k-th element in a
     subinterval
void build(int n){
  for (int i = 0;i <= n;++i){
    head[i] = nextPos();
  }
  build(head[0], 1, n);
  for (int i = 1;i <= n;++i){
    modify(head[i], head[i-1],c[i], 1, n);
  }
}
/*Query the k-th element in [l,r]:
printf("%d\n",a[query(head[r], head[l-1], 1, n, k)].key);*/
```

Point3d.cpp

8058ae, 32 lines

```
template<class T> struct Point3D {
  typedef Point3D P;
  typedef const P& R;
  T x, y, z;
  explicit Point3D(T x=0, T y=0, T z=0) : x(x), y(y), z(z) {
      }
  bool operator<(R p) const {
    return tie(x, y, z) < tie(p.x, p.y, p.z); }
  bool operator==(R p) const {
    return tie(x, y, z) == tie(p.x, p.y, p.z); }
  P operator+(R p) const { return P(x+p.x, y+p.y, z+p.z); }
  P operator-(R p) const { return P(x-p.x, y-p.y, z-p.z); }
  P operator*(T d) const { return P(x*d, y*d, z*d); }
  P operator/(T d) const { return P(x/d, y/d, z/d); }
  T dot(R p) const { return x*p.x + y*p.y + z*p.z; }
  P cross(R p) const {
    return P(y*p.z - z*p.y, z*p.x - x*p.z, x*p.y - y*p.x);
  }
  T dist2() const { return x*x + y*y + z*z; }
  double dist() const { return sqrt((double)dist2()); }
  //Azimuthal angle (longitude) to x-axis in interval [-pi,
      pi]
  double phi() const { return atan2(y, x); }
  //Zenith angle (latitude) to the z-axis in interval [0, pi
      ]
  double theta() const { return atan2(sqrt(x*x+y*y),z); }
  P unit() const { return *this/(T)dist(); } //makes dist()
      =1
  //returns unit vector normal to *this and p
  P normal(P p) const { return cross(p).unit(); }
  //returns point rotated 'angle' radians ccw around axis
  P rotate(double angle, P axis) const {
    double s = sin(angle), c = cos(angle); P u = axis.unit()
      ;
    return u*dot(u)*(1-c) + (*this)*c - cross(u)*s;
  }
};
```

Point.cpp

ec47f9, 93 lines

```
template <class T> int sgn(T x) { return (x > 0) - (x < 0);
    }
template<class T>
struct Point {
  typedef Point P;
  T x, y;
  explicit Point(T x=0, T y=0) : x(x), y(y) {}
  bool operator<(P p) const { return tie(x,y) < tie(p.x,p.y)
      ; }
  bool operator==(P p) const { return tie(x,y)==tie(p.x,p.y)
      ; }
  P operator+(P p) const { return P(x+p.x, y+p.y); }
  P operator-(P p) const { return P(x-p.x, y-p.y); }
  P operator*(T d) const { return P(x*d, y*d); }
  P operator/(T d) const { return P(x/d, y/d); }
  T dot(P p) const { return x*p.x + y*p.y; }
  T cross(P p) const { return x*p.y - y*p.x; }
  T cross(P a, P b) const { return (a-*this).cross(b-*this);
      }
  T dist2() const { return x*x + y*y; }
  double dist() const { return sqrt((double)dist2()); }
  // angle to x-axis in interval [-pi, pi]
  double angle() const { return atan2(y, x); }
  P unit() const { return *this/dist(); } // makes dist()=1
  P perp() const { return P(-y, x); } // rotates +90 degrees
  P normal() const { return perp().unit(); }
  // returns point rotated 'a' radians ccw around the origin
  P rotate(double a) const {
    return P(x*cos(a)-y*sin(a),x*sin(a)+y*cos(a)); }
  friend ostream& operator<<(ostream& os, P p) {
    return os << "(" << p.x << "," << p.y << ")"; }
};

//Line distance
template<class P>
double lineDist(const P& a, const P& b, const P& p) {
  return (double)(b-a).cross(p-a)/(b-a).dist();
}

//LineProjectionReflection
template<class P>
P lineProj(P a, P b, P p, bool refl=false) {
  P v = b - a;
  return p - v.perp()*(1+refl)*v.cross(p-a)/v.dist2();
}

//Point-Segment Distance
typedef Point<double> P;
double segDist(P& s, P& e, P& p) {
  if (s==e) return (p-s).dist();
  auto d = (e-s).dist2(), t = min(d,max(.0,(p-s).dot(e-s)));
  return ((p-s)*d-(e-s)*t).dist()/d;
}

//Segment-Segment Distance
double TwoSegMinDist(P A,P B,P C,P D)
{
    return min(min(segDist(A,B,C),segDist(A,B,D)),
               min(segDist(C,D,A),segDist(C,D,B)));
}

//On Segment
template<class P> bool onSegment(P s, P e, P p) {
  return p.cross(s, e) == 0 && (s - p).dot(e - p) <= 0;
}

//Segment Intersection
/*If a unique intersection point between the line segments
    going from s1 to e1 and from s2 to e2 exists then it is
    returned.
If no intersection point exists an empty vector is returned.
    If infinitely many exist a vector with 2 elements is
    returned, containing the endpoints of the common line
    segment.
The wrong position will be returned if P is Point<ll> and
    the intersection point does not have integer coordinates
    .
Products of three coordinates are used in intermediate steps
    so watch out for overflow if using int or long long.*/
template<class P> vector<P> segInter(P a, P b, P c, P d) {
  auto oa = c.cross(d, a), ob = c.cross(d, b),
       oc = a.cross(b, c), od = a.cross(b, d);
  // Checks if intersection is single non-endpoint point.
  if (sgn(oa) * sgn(ob) < 0 && sgn(oc) * sgn(od) < 0)
    return {(a * ob - b * oa) / (ob - oa)};
  set<P> s;
  if (onSegment(c, d, a)) s.insert(a);
  if (onSegment(c, d, b)) s.insert(b);
  if (onSegment(a, b, c)) s.insert(c);
  if (onSegment(a, b, d)) s.insert(d);
  return {all(s)};
}
```

```
//The notations from here might be different
//Find the symmetric point of point p about line p1p2
Point SymPoint(Point p,Line l)
{
    Point result;
    double a=l.p2.x-l.p1.x;
    double b=l.p2.y-l.p1.y;
    double t=((p.x-l.p1.x)*a+(p.y-l.p1.y)*b)/(a*a+b*b);
    result.x=2*l.p1.x+2*a*t-p.x;
    result.y=2*l.p1.y+2*b*t-p.y;
    return result;
}
```

PollardRho.cpp

d9508f, 82 lines

```
//Quick Multiplication - Calculate x * y mod modi
    efficiently
//where x and y is in long long range
ll quickmult(ll x, ll y, ll p){
    ll temp = x * y - ((ll)((long double)x / p * y + 0.5)) *
        p;
    return (temp < 0) ? temp + p : temp;
}

//Prime Test via Miller-Rabin
bool prime_test(ll p){
    static int tests[] = {2, 3, 5, 7, 11, 13, 17, 19, 23,
        29, 31, 37};
    int r = 0;
    ll b = p - 1;
    if (p == 2) return true;
    if (p == 1 || (p & 1) == 0) return false;

    while ((b & 1) == 0){
        r++;
        b >>= 1;
    }
    ll d = (p - 1) / (1ll << r);
    for (int i = 0;i < 12;i++){
        if (p == tests[i]){
            return true;
        }
        ll x = quickpow2(tests[i], d, p);
        for (int j = 1;j <= r;j++){
            ll y = quickmult(x, x, p);
            if (y == 1 && x != 1 && x != p - 1) return false
                ;
            x = y;
        }
        if (x != 1) return false;
    }
    return true;
}

//We will store factors in a global variable to save time
map<ll, int> factors;

ll gcd(ll x, ll y){
    if (y == 0) return x;
    return gcd(y, x % y);
}

l get_next(ll x, ll addi,ll modi){
    return (quickmult(x, x, modi) + addi);
}

//find a prime factor of n based on the seed, if we cannot
    find it return -1
ll rho_find(ll n, ll seed, ll addi){
    ll a = seed;
    ll b = get_next(seed, addi, n);
    while (a != b){
        ll p = gcd(abs(a - b), n);
        if (p > 1){
            if (p == n) return -1;
            return p;
        }
        a = get_next(a, addi, n);
        b = get_next(get_next(b, addi, n), addi, n);
    }
    return -1;
}

//factorizing n via pollard rho
void pollard_rho(ll n){
    if (n == 1){
        return;
    }
    if (prime_test(n)){
        factors[n]++;
        return;
    }
    ll p = -1;
    while (p == -1){
        ll addi = rand() % (n - 1) + 1;
        p = rho_find(n, rand() % (n - 1) + 1, addi);
        if (p != -1){
            pollard_rho(p);
            pollard_rho(n / p);
            return;
        }
    }
}
```

PolygonUnion.cpp

"Point.h", "sideOf.h" b73c5a, 39 lines

```
/* KACTL Polygon Union
 * Description: Calculates the area of the union of $n$
     polygons (not necessarily
 * convex). The points within each polygon must be given in
     CCW order.
 * (Epsilon checks may optionally be added to sideOf/sgn,
     but shouldn't be needed.)
 * Time: $O(N^2)$, where $N$ is the total number of points*/

typedef Point<double> P;
double rat(P a, P b) { return sgn(b.x) ? a.x/b.x : a.y/b.y;
    }
double polyUnion(vector<vector<P>>& poly) {
  double ret = 0;
  FOR(i,0,sz(poly)) rep(v,0,sz(poly[i])) {
    P A = poly[i][v], B = poly[i][(v + 1) % sz(poly[i])];
    vector<pair<double, int>> segs = {{0, 0}, {1, 0}};
    FOR(j,0,sz(poly)) if (i != j) {
      FOR(u,0,sz(poly[j])) {
        P C = poly[j][u], D = poly[j][(u + 1) % sz(poly[j])
            ];
        int sc = sideOf(A, B, C), sd = sideOf(A, B, D);
        if (sc != sd) {
          double sa = C.cross(D, A), sb = C.cross(D, B);
          if (min(sc, sd) < 0)
            segs.emplace_back(sa / (sa - sb), sgn(sc - sd));
        } else if (!sc && !sd && j<i && sgn((B-A).dot(D-C))
            >0){
          segs.emplace_back(rat(C - A, B - A), 1);
          segs.emplace_back(rat(D - A, B - A), -1);
        }
      }
    }
    sort(all(segs));
    for (auto& s : segs) s.first = min(max(s.first, 0.0),
        1.0);
    double sum = 0;
    int cnt = segs[0].second;
    FOR(j,1,sz(segs)) {
      if (!cnt) sum += segs[j].first - segs[j - 1].first;
      cnt += segs[j].second;
    }
    ret += A.cross(B) * sum;
  }
  return ret / 2;
}
```

Polynomial.cpp

92b62a, 87 lines

```
constexpr int maxn = 262144;
constexpr int mod = 998244353;

using i64 = long long;
using poly_t = int[maxn];
using poly = int *const;

//Find f^{-1} mod x^n.
void polyinv(const poly &h, const int n, poly &f) {
  /* f = 1 / h = f_0 (2 - f_0 h) */
  static poly_t inv_t;
  std::fill(f, f + n + n, 0);
  f[0] = fpow(h[0], mod - 2);
  for (int t = 2; t <= n; t <<= 1) {
    const int t2 = t << 1;
    std::copy(h, h + t, inv_t);
    std::fill(inv_t + t, inv_t + t2, 0);

    DFT(f, t2);
    DFT(inv_t, t2);
    for (int i = 0; i != t2; ++i)
      f[i] = (i64)f[i] * sub(2, (i64)f[i] * inv_t[i] % mod)
          % mod;
    IDFT(f, t2);

    std::fill(f + t, f + t2, 0);
  }
}

//Find h(x) such that h^2(x) = f(x) mod x^{deg}.
inline void sqrt(int deg, int *f, int *h) {
  if (deg == 1) {
    h[0] = 1;
    return;
  }
  sqrt(deg + 1 >> 1, f, h);
  int len = 1;
  while (len < deg * 2) {  // doubling
    len *= 2;
  }
  fill(g, g + len, 0);
```

```
  inv(deg, h, g);
  copy(f, f + deg, t);
  fill(t + deg, t + len, 0);
  NTT(t, len, 1);
  NTT(g, len, 1);
  NTT(h, len, 1);
  for (int i = 0; i < len; i++) {
    h[i] = (long long)1 * inv2 *
           ((long long)1 * h[i] % mod + (long long)1 * g[i]
                * t[i] % mod) % mod;
  }
  NTT(h, len, -1);
  fill(h + deg, h + len, 0);
}

/*This is Fast Walsh Transformation
Goal: Given A, B, compute C_i = \sum_{j ? k = i}A_jB_k
? is or, and, xor*/
void FWT(int *f,int pd){
    for(int d=1;d<n;d<<=1)
        for(int m=d<<1,i=0;i<n;i+=m)
            for(int j=0;j<d;j++){
                int x=f[i+j],y=f[i+j+d];
                if(pd==0) f[i+j+d]=(x+y)%p; // or
                if(pd==1) f[i+j]=(x+y)%p; // and
                if(pd==2) f[i+j]=(x+y)%p,f[i+j+d]=(x-y+p)%p;
                      //xor
            }
}
void IFWT(int *f,int pd){
    for(int d=1;d<n;d<<=1)
        for(int m=d<<1,i=0;i<n;i+=m)
            for(int j=0;j<d;j++){
                int x=f[i+j],y=f[i+j+d];
                if(pd==0) f[i+j+d]=(y-x+p)%p; //OR
                if(pd==1) f[i+j]=(x-y+p)%p;; // AND
                if(pd==2) f[i+j]=1ll*(x+y)*inv%p,f[i+j+d]=1
                     ll*(x-y+p)*inv%p; //XOR
            }
}

void solve_or()
{
    memcpy(a,A,sizeof a);
    memcpy(b,B,sizeof b);
    FWT(a,0);FWT(b,0);
    for(int i=0;i<n;i++)
        a[i]=1ll*a[i]*b[i]%p;
    IFWT(a,0);
}
```

PushRelabel.cpp

Description: Push-relabel using the highest label selection rule and the gap heuristic. Quite fast in practice. To obtain the actual flow, look at positive values only.

Time: $\mathcal{O}\left(V^2\sqrt{E}\right)$

be9cf6, 48 lines

```
struct PushRelabel {
  struct Edge {
    int dest, back;
    ll f, c;
  };
  vector<vector<Edge>> g;
  vector<ll> ec;
  vector<Edge*> cur;
  vector<vi> hs; vi H;
  PushRelabel(int n) : g(n), ec(n), cur(n), hs(2*n), H(n) {}

  void addEdge(int s, int t, ll cap, ll rcap=0) {
    if (s == t) return;
    g[s].push_back({t, sz(g[t]), 0, cap});
    g[t].push_back({s, sz(g[s])-1, 0, rcap});
  }

  void addFlow(Edge& e, ll f) {
    Edge &back = g[e.dest][e.back];
    if (!ec[e.dest] && f) hs[H[e.dest]].push_back(e.dest);
    e.f += f; e.c -= f; ec[e.dest] += f;
    back.f -= f; back.c += f; ec[back.dest] -= f;
  }
  ll calc(int s, int t) {
    int v = sz(g); H[s] = v; ec[t] = 1;
    vi co(2*v); co[0] = v-1;
    FOR(i,0,v) cur[i] = g[i].data();
    for (Edge& e : g[s]) addFlow(e, e.c);

    for (int hi = 0;;) {
      while (hs[hi].empty()) if (!hi--) return -ec[s];
      int u = hs[hi].back(); hs[hi].pop_back();
      while (ec[u] > 0)  // discharge u
        if (cur[u] == g[u].data() + sz(g[u])) {
          H[u] = 1e9;
          for (Edge& e : g[u]) if (e.c && H[u] > H[e.dest
               ]+1)
            H[u] = H[e.dest]+1, cur[u] = &e;
          if (++co[H[u]], !--co[hi] && hi < v)
            FOR(i,0,v) if (hi < H[i] && H[i] < v)
              --co[H[i]], H[i] = v + 1;
          hi = H[u];
        } else if (cur[u]->c && H[u] == H[cur[u]->dest]+1)
          addFlow(*cur[u], min(ec[u], cur[u]->c));
        else ++cur[u];
    }
  }
  bool leftOfMinCut(int a) { return H[a] >= sz(g); }
};
```

QuasiExgcdSum.cpp

b1006b, 60 lines

```
/*
Using Quasi_Exgcd to sum
f(a,b,c,n) = sum_{i=0}^n \floor{(ai+b)/c}
g(a,b,c,n) = sum_{i=0}^n \floor i{(ai+b)/c}
h(a,b,c,n) = sum_{i=0}^n (\floor{(ai+b)/c})^2
all are done under mod p
*/

struct rec{
    ll f, g, h;
};

//add, sub, quickpow omitted
ll inv2 = quickpow(2, modi-2);
ll inv6 = quickpow(6, modi - 2);

rec solve(ll a, ll b, ll c, ll n){
  rec ans;
  if (a == 0){
    ans.f = (b / c) * (n + 1) % modi;
    ans.g = (b / c) * (n + 1) % modi * n % modi * inv2 %
         modi;
    ans.h = (b / c) * (b / c) % modi * (n+1) % modi;
    return ans;
  }
  ans.f = ans.g = ans.h = 0;
  if (a >= c || b >= c){
    rec temp = solve(a % c, b % c, c, n);
    add(ans.f, (a/c)*n%modi*(n+1)%modi*inv2%modi);
    add(ans.f, (b/c)*(n+1)%modi);
    add(ans.f, temp.f);
    add(ans.g, (a/c)*n%modi*(n+1)%modi*
    ((2*n+1)%modi)%modi*inv6%modi);
    add(ans.g, (b/c)*n%modi*(n+1)%modi*inv2 % modi);
    add(ans.g, temp.g);
    add(ans.h, (a/c)*(a/c)%modi*n%modi*
    (n+1)%modi*((2*n+1)%modi)%modi*inv6% modi);
    add(ans.h, (b/c)*(b/c)%modi*(n+1)%modi);
    add(ans.h, (a/c)*(b/c)%modi*n%modi*(n+1)%modi);
    add(ans.h, temp.h);
    add(ans.h, 2LL * (a/c)%modi*temp.g%modi);
    add(ans.h, 2LL * (b/c)%modi*temp.f%modi);
    return ans;
  }
  if (a < c && b < c){
    ll m = (a * n + b) / c;
    rec temp = solve(c, c - b - 1, a, m - 1);
    ans.f = n * m % modi;
    sub(ans.f, temp.f);
    ans.g = n * (n + 1) % modi * m % modi;
    sub(ans.g, temp.f);
    sub(ans.g, temp.h);
    ans.g = ans.g * inv2 % modi;
    ans.h = n * m % modi * (m + 1) % modi;
    sub(ans.h, 2LL * temp.g % modi);
    sub(ans.h, 2LL * temp.f % modi);
    sub(ans.h, ans.f);
    return ans;
  }
  return ans;
}
```

QuickPhiSum.cpp

024bac, 29 lines

```
/*
This algorithm concerns efficient evaluation of sum of
     number theoric functions like phi or mu.
We know that using Eulerian sieve, we can only archieve O(n)
     time complexity.
What we are doing is to archieve O(n^{2/3}) time complexity.
The example program shows how to evaluate sum of phi and sum
     of mu efficientyl.
For smaller n (n less than (N^{2/3}), we use calculate them
     as usual.
For larger n, see getphi and getmu
*/

//See Sieve for more technical details.
//When i is prime, phi[i] = i-1;mu[i] = -1; Otherwise,
     inside the inner loop, let p = primes[j].
//Then it follows phi[p*i] = phi[i] * (p-1); mu[p*i] = -mu[i
     ];
//finally, when i % p == 0, phi[p*i] = phi[i]*p and mu[p*i]
     = 0;
```

```
ll getphi(ll n)
{
  if (n <= m) return phi[n];
  if (phi_cheat.find(n) != phi_cheat.end()) return phi_cheat
      [n];
  ll ans = (ll)n*(n + 1) / 2; //this is \sum_{i = 1}^n \sum_
      {d | n} \phi(d)
  //when getting mu, ans = 1
  ll last;
  for (ll i = 2;i <= n;i = last + 1)
  {
    last = n / (n / i);
    ans -= (last-i+1)*getphi(n / i);
  }
  phi_cheat[n] = ans;
  return ans;
}
```

Random.cpp

a59bbc, 2 lines

```
#define uid(a, b) uniform_int_distribution<int>(a, b)(rng)
mt19937 rng(chrono::steady_clock::now().time_since_epoch().
     count());
```

RootNonTree.cpp

9d4bb6, 38 lines

```
void dfs(int u){
  static int top;
  stk[++top] = u;
  for (int i = 0;i < g[u].size();i++){
    int v = g[u][i];
    if (v != p[u]){
        p[v] = u;
        dfs(v);
        if (siz[u] + siz[v] >= magic){
          siz[u] = 0;
          tot_cols++;
          cap[tot_cols] = u;
          while (stk[top] != u){
            col[stk[top--]] = tot_cols;
          }
        }
        else siz[u] += siz[v];
    }
  }
  siz[u]++;
}

void paint(int u,int c){
    if (col[u]) c = col[u];
    else col[u] = c;
    for (int i = 0;i < g[u].size();i++){
        int v = g[u][i];
        if (v != p[u]){
          paint(v,c);
        }
    }
}
//actual blokcing; magic = block size
dfs(1);
if (!tot_cols){
    cap[++tot_cols] = 1;
}
paint(1,tot_cols);
```

Sam.cpp

<cstdio>, <cstdlib>, <cstring>, <algorithm> 6196dc, 58 lines

```
using namespace std;

typedef long long ll;

const int maxn = 510000;
const int sigma = 26;

struct edge
{
    int v, next;
};

edge g[maxn << 1];
int trie[maxn << 1][sigma], fa[maxn << 1], maxi[maxn << 1],
     sizia[maxn << 1];
char str[maxn];
int head[maxn << 1];
int siz,last;

void insert(int u, int v)
{
    static int id;
    g[++id].v = v;
    g[id].next = head[u];
    head[u] = id;
}

//This is the core of SAM
void add(int id)
{
    int p = last;
    int np = last = ++siz;
    sizia[np] = 1;
    maxi[np] = maxi[p] + 1;
    while (p && !trie[p][id]){
        trie[p][id] = np;
        p = fa[p];
    }
    if (!p){
        fa[np] = 1;
    }
    else{
        int q = trie[p][id];
        if (maxi[p] + 1 == maxi[q]){
            fa[np] = q;
        }
        else{
            int nq = ++siz;
            maxi[nq] = maxi[p] + 1;
            memcpy(trie[nq], trie[q], sizeof trie[q]);
            fa[nq] = fa[q];
            fa[np] = fa[q] = nq;
            while (trie[p][id] == q){
                trie[p][id] = nq;
                p = fa[p];
            }
        }
    }
}
```

Simplex.cpp

Description: Solves a general linear maximization problem: maximize $c^T x$ subject to $Ax \le b$, $x \ge 0$. Returns -inf if there is no solution, inf if there are arbitrarily good solutions, or the maximum value of $c^T x$ otherwise. The input vector is set to an optimal x (or in the unbounded case, an arbitrary solution fulfilling the constraints). Numerical stability is not guaranteed. For better performance, define variables such that $x = 0$ is viable.

Usage: `vvd A = {{1,-1}, {-1,1}, {-1,-2}};`
`vd b = {1,1,-4}, c = {-1,-1}, x;`
`T val = LPSolver(A, b, c).solve(x);`

Time: $\mathcal{O}(NM \cdot \#pivots)$, where a pivot may be e.g. an edge relaxation. $\mathcal{O}\left(2^N\right)$ in the general case.

f40c06, 71 lines

```
typedef vector<vd> vvd;
const ld eps = 1e-8, inf = 1/.0;

#define ltj(X) if (s == -1 || make_pair(X[j],N[j]) <
     make_pair(X[s],N[s])) s=j

struct LPSolver {
  int m, n; // # contraints, # variables
  vi N, B;
  vvd D;
  LPSolver(const vvd& A, const vd& b, const vd& c) :
    m(sz(b)), n(sz(c)), N(n+1), B(m), D(m+2, vd(n+2)) {
      FOR(i,m) FOR(j,n) D[i][j] = A[i][j];
      FOR(i,m) {
        B[i] = n+i, D[i][n] = -1, D[i][n+1] = b[i];
        // B[i]: add basic variable for each constraint,
             convert ineqs to eqs
        // D[i][n]: artificial variable for testing
             feasibility
      }
      FOR(j,n) {
        N[j] = j; // non-basic variables, all zero
        D[m][j] = -c[j]; // minimize -c^T x
      }
      N[n] = -1; D[m+1][n] = 1;
    }

  void pivot(int r, int s) { // r = row, c = column
    ld *a = D[r].data(), inv = 1/a[s];
    FOR(i,m+2) if (i != r && abs(D[i][s]) > eps) {
      ld *b = D[i].data(), binv = b[s]*inv;
      FOR(j,n+2) b[j] -= a[j]*binv; // make column
           corresponding to s all zeroes
      b[s] = a[s]*binv; // swap N[s] with B[r]
    }
    // equation corresponding to r scaled so x_r coefficient
         equals 1
    FOR(j,n+2) if (j != s) D[r][j] *= inv;
    FOR(i,m+2) if (i != r) D[i][s] *= -inv;
    D[r][s] = inv; swap(B[r], N[s]); // swap basic w/ non-
         basic
  }
  bool simplex(int phase) {
    int x = m+phase-1;
    while (1) {
      int s = -1; FOR(j,n+1) if (N[j] != -phase) ltj(D[x]);
           // find most negative col for nonbasic variable
      if (D[x][s] >= -eps) return true; // can't get better
           sol by increasing non-basic variable, terminate
      int r = -1;
      FOR(i,m) {
```

```
        if (D[i][s] <= eps) continue;
        if (r == -1 || mp(D[i][n+1] / D[i][s], B[i])
              < mp(D[r][n+1] / D[r][s], B[r])) r = i;
        // find smallest positive ratio, aka max we can
             increase nonbasic variable
      }
      if (r == -1) return false; // increase N[s] infinitely
           -> unbounded
      pivot(r,s);
    }
  }

  ld solve(vd &x) {
    int r = 0; FOR(i,1,m) if (D[i][n+1] < D[r][n+1]) r = i;
    if (D[r][n+1] < -eps) { // x=0 not feasible, run simplex
         to find smth feasible
      pivot(r, n); // N[n] = -1 is artificial variable,
           initially set to smth large
      if (!simplex(2) || D[m+1][n+1] < -eps) return -inf;
      // D[m+1][n+1] is max possible value of the negation
           of
      // artificial variable, optimal value should be zero
      // if exists feasible solution
      FOR(i,m) if (B[i] == -1) { // ?
        int s = 0; FOR(j,1,n+1) ltj(D[i]);
        pivot(i,s);
      }
    }
    bool ok = simplex(1); x = vd(n);
    FOR(i,m) if (B[i] < n) x[B[i]] = D[i][n+1];
    return ok ? D[m][n+1] : inf;
  }
};
```

SCC.cpp

a6b342, 37 lines

```
vector < vector<int> > g, gr; //g stores graph, gr stores
     graph transposed
vector<bool> used;
vector<int> order, component;

void dfs1 (int v) {
    used[v] = true;
    for (size_t i=0; i<g[v].size(); ++i)
        if (!used[ g[v][i] ])
            dfs1 (g[v][i]);
    order.push_back (v);
}

void dfs2 (int v) {
    used[v] = true;
    component.push_back (v);
    for (size_t i=0; i<gr[v].size(); ++i)
        if (!used[ gr[v][i] ])
            dfs2 (gr[v][i]);
}

void findSCCs() {
    order.clear();
    used.assign (n, false);
    for (int i=0; i<n; ++i)
        if (!used[i])
            dfs1 (i);
    used.assign (n, false);
    for (int i=0; i<n; ++i) {
        int v = order[n-1-i];
        if (!used[v]) {
            dfs2 (v);
            //SCC FOUND, DO SOMETHING
            component.clear();
        }
    }

}
```

Schreier-Sims.cpp

a288c1, 102 lines

```
// time complexity : O(n^2 log^3 |G| + t n log |G|)
// memory complexity : O(n^2 log |G| + tn)
// t : number of generators
// |G| : group size, obviously <= (n!)

vector<int> inv(vector<int>& p){
  vector<int> ret(p.size());
  for (int i = 0; i < p.size(); i++) ret[p[i]] = i;
  return ret;
}
vector<int> operator * (vector<int>& a, vector<int>& b ){
  vector<int> ret(a.size());
  for (int i = 0 ; i < a.size(); i++) ret[i] = b[a[i]];
  return ret;
}
// a group contains all subset products of generators
struct Group {
  int n, m;
  vector<vector<int>> lookup;
  vector<vector<vector<int>>> buckets, ibuckets;
  int yo(vector<int> p, bool add_to_group = 1){
    n = buckets.size();
    for (int i = 0; i < n ; i++){
      int res = lookup[i][p[i]];
      if (res == -1 ){
        if (add_to_group){
          buckets[i].push_back(p);
          ibuckets[i].push_back(inv(p));
          lookup[i][p[i]] = buckets[i].size() - 1;
        }
        return i;
      }
      p = p * ibuckets[i][res];
    }
    return -1;
  }
  ll size() {
    ll ret = 1;
    for (int i = 0; i < n; i++) ret *= buckets[i].size();
    return ret;
  }
  bool in_group(vector<int> g) { return yo(g, false) == -1;
       }
  Group(vector<vector<int>> &gen, int _n){
    n = _n, m = gen.size(); // m permutations of size n, 0
         indexed
    lookup.resize(n);
    buckets.resize(n);
    ibuckets.resize(n);
    for (int i = 0; i < n ; i++){
      lookup[i].resize(n);
      fill(lookup[i].begin(), lookup[i].end(), -1);
    }
    vector<int> id(n);
    for (int i = 0; i < n ; i++) id[i] = i;
    for (int i = 0; i < n ; i++){
      buckets[i].push_back(id);
      ibuckets[i].push_back(id);
      lookup[i][i] = 0;
    }
    for (int i = 0; i < m ; i++) yo(gen[i]);
    queue<pair<pair<int, int>,pair<int, int>>> q;
    for (int i = 0; i < n ; i++) {
      for (int j = i; j < n ; j++) {
        for (int k = 0; k < buckets[i].size(); k++) {
          for (int l = 0; l < buckets[j].size(); l++) {
            q.push({pair<int, int>(i, k), pair<int, int>(j,
                 l)});
          }
        }
      }
    }
    while(!q.empty()) {
      pair<int, int> a = q.front().first;
      pair<int, int> b = q.front().second;
      q.pop();
      int res = yo(buckets[a.first][a.second] * buckets[b.
           first][b.second]);
      if (res == -1) continue;
      pair<int, int> cur(res, (int)buckets[res].size() - 1);
      for (int i = 0; i < n; i ++) {
        for (int j = 0; j < (int)buckets[i].size(); ++j){
          if (i <= res) q.push(make_pair(pair<int, int>(i ,
               j), cur));
          if (res <= i) q.push(make_pair(cur, pair<int, int
               >(i, j)));
        }
      }
    }
  }
};
int32_t main() {
  ios_base::sync_with_stdio(0);
  cin.tie(0);
  int k, n; cin >> k >> n;
  vector<vector<int>> a;
  while (k--) {
   vector<int> v;
   for  (int i = 0; i < n; i++) {
     int x; cin >> x;
     v.push_back(x - 1);
   }
   a.push_back(v);
  }
  Group g(a, n);
  cout << g.size() << '\n';
  return 0;
}
```

Segtree.cpp

4515af, 20 lines

```
const ll SZ = 262144; //set this to power of two
ll seg[2*SZ];

ll combine(ll a, ll b) { return max(a, b); }

void build() { F0Rd(i,SZ) seg[i] = combine(seg[2*i],seg[2*i
     +1]); }
```

```
void update(int p, ll value) {
    for (seg[p += SZ] = value; p > 1; p >>= 1)
        seg[p>>1] = combine(seg[(p|1)^1], seg[p|1]);
}

ll query(int l, int r) {  // sum on interval [l, r]
    ll resL = 0, resR = 0; r++;
    for (l += SZ, r += SZ; l < r; l >>= 1, r >>= 1) {
        if (l&1) resL = combine(resL,seg[l++]);
        if (r&1) resR = combine(seg[--r],resR);
    }
    return combine(resL,resR);
}
```

Sieve.cpp

559ed1, 14 lines

```
vi primes, leastFac;
void compPrimes(int N) {
  leastFac.resize(N, 0);
  leastFac[0] = 1; leastFac[1] = 1;
  FOR(i, 2, N) {
    if (leastFac[i] == 0) {
      primes.pb(i);
      leastFac[i] = i;
    }
    for (int j = 0; j < sz(primes) && i*primes[j] < N &&
         primes[j] <= leastFac[i]; j++) {
      leastFac[i*primes[j]] = primes[j];
    }
  }
}
```

Simpson.cpp

88ae08, 20 lines

```
/*
This is a template for solving simpson integration
We are going to integrate \frac{cx+d}{ax+b} over L and R
*/
ld simpson(ld lower, ld upper){
    ld mid = (lower + upper) / 2;
    return (f(lower) + 4 * f(mid) + f(upper)) * (upper-lower
         ) / 6;
}

ld simpson_integration(ld lower, ld upper, ld target){
    ld mid = (lower + upper) / 2;
    ld left_sum = simpson(lower, mid);
    ld right_sum = simpson(mid, upper);
    if (fabs(left_sum + right_sum - target) < eps){
        return left_sum + right_sum;
    }
    return simpson_integration(lower, mid, left_sum) +
         simpson_integration(mid, upper, right_sum);
}

//Call: simpson_integration(lower, upper, simpson(lower,
     upper)) << endl;
```

stress.cpp

95d57c, 22 lines

```
#!/usr/bin/env bash

for ((testNum=0;testNum<$4;testNum++))
do
    ./$3 > input
    ./$2 < input > outSlow
    ./$1 < input > outWrong
    H1=`md5sum outWrong`
    H2=`md5sum outSlow`
    if !(cmp -s "outWrong" "outSlow")
    then
        echo "Error found!"
        echo "Input:"
        cat input
        echo "Wrong Output:"
        cat outWrong
        echo "Slow Output:"
        cat outSlow
        exit
    fi
done
echo Passed $4 tests
```

SuffixArray.cpp

52732a, 74 lines

```
vector<int> sort_cyclic_shifts(string const& s) {
    int n = s.size();
    const int alphabet = 256;
     vector<int> p(n), c(n), cnt(max(alphabet, n), 0);
    for (int i = 0; i < n; i++)
        cnt[s[i]]++;
    for (int i = 1; i < alphabet; i++)
        cnt[i] += cnt[i-1];
    for (int i = 0; i < n; i++)
        p[--cnt[s[i]]] = i;
    c[p[0]] = 0;
    int classes = 1;
    for (int i = 1; i < n; i++) {
        if (s[p[i]] != s[p[i-1]])
            classes++;
        c[p[i]] = classes - 1;
    }
    vector<int> pn(n), cn(n);
    for (int h = 0; (1 << h) < n; ++h) {
        for (int i = 0; i < n; i++) {
            pn[i] = p[i] - (1 << h);
            if (pn[i] < 0)
                pn[i] += n;
        }
        fill(cnt.begin(), cnt.begin() + classes, 0);
        for (int i = 0; i < n; i++)
            cnt[c[pn[i]]]++;
        for (int i = 1; i < classes; i++)
            cnt[i] += cnt[i-1];
        for (int i = n-1; i >= 0; i--)
            p[--cnt[c[pn[i]]]] = pn[i];
        cn[p[0]] = 0;
        classes = 1;
        for (int i = 1; i < n; i++) {
            pair<int, int> cur = {c[p[i]], c[(p[i] + (1 << h
                 )) % n]};
            pair<int, int> prev = {c[p[i-1]], c[(p[i-1] + (1
                 << h)) % n]};
            if (cur != prev)
                ++classes;
            cn[p[i]] = classes - 1;
        }
        c.swap(cn);
    }
    return p;
}

vector<int> suffix_array_construction(string s) {
    s += "$";
    vector<int> sorted_shifts = sort_cyclic_shifts(s);
    sorted_shifts.erase(sorted_shifts.begin());
    return sorted_shifts;
}

vector<int> lcp_construction(string const& s, vector<int>
     const& p) {
    int n = s.size();
    vector<int> rank(n, 0);
    for (int i = 0; i < n; i++)
        rank[p[i]] = i;

    int k = 0;
    vector<int> lcp(n-1, 0);
    for (int i = 0; i < n; i++) {
        if (rank[i] == n - 1) {
            k = 0;
            continue;
        }
        int j = p[rank[i] + 1];
        while (i + k < n && j + k < n && s[i+k] == s[j+k])
            k++;
        lcp[rank[i]] = k;
        if (k)
            k--;
    }
    return lcp;
}
```

SuffixTree.cpp

Description: Ukkonen's algorithm for online suffix tree construction. Each node contains indices [l, r) into the string, and a list of child nodes. Suffixes are given by traversals of this tree, joining [l, r) substrings. The root is 0 (has l = -1, r = 0), non-existent children are -1. To get a complete tree, append a dummy symbol – otherwise it may contain an incomplete path (still useful for substring matching, though).
Time: $\mathcal{O}(26N)$

07bb84, 50 lines

```
struct SuffixTree {
  enum { N = 200010, ALPHA = 26 }; // N ~ 2*maxlen+10
  int toi(char c) { return c - 'a'; }
  string a; // v = cur node, q = cur position
  int t[N][ALPHA],l[N],r[N],p[N],s[N],v=0,q=0,m=2;

  void ukkadd(int i, int c) { suff:
    if (r[v]<=q) {
      if (t[v][c]==-1) { t[v][c]=m;  l[m]=i;
        p[m++]=v; v=s[v]; q=r[v];  goto suff; }
      v=t[v][c]; q=l[v];
    }
    if (q==-1 || c==toi(a[q])) q++; else {
      l[m+1]=i;  p[m+1]=m;  l[m]=l[v];  r[m]=q;
      p[m]=p[v];  t[m][c]=m+1;  t[m][toi(a[q])]=v;
      l[v]=q;  p[v]=m;  t[p[m]][toi(a[l[m]])]=m;
      v=s[p[m]];  q=l[m];
      while (q<r[m]) { v=t[v][toi(a[q])];  q+=r[v]-l[v]; }
      if (q==r[m])  s[m]=v;  else s[m]=m+2;
      q=r[v]-(q-r[m]);  m+=2;  goto suff;
    }
  }
```

```
  SuffixTree(string a) : a(a) {
    fill(r,r+N,sz(a));
    memset(s, 0, sizeof s);
    memset(t, -1, sizeof t);
    fill(t[1],t[1]+ALPHA,0);
    s[0] = 1; l[0] = l[1] = -1; r[0] = r[1] = p[0] = p[1] =
        0;
    FOR(i,0,sz(a)) ukkadd(i, toi(a[i]));
  }

  // example: find longest common substring (uses ALPHA =
      28)
  pii best;
  int lcs(int node, int i1, int i2, int olen) {
    if (l[node] <= i1 && i1 < r[node]) return 1;
    if (l[node] <= i2 && i2 < r[node]) return 2;
    int mask = 0, len = node ? olen + (r[node] - l[node]) :
        0;
    FOR(c,0,ALPHA) if (t[node][c] != -1)
      mask |= lcs(t[node][c], i1, i2, len);
    if (mask == 3)
      best = max(best, {len, r[node] - len});
    return mask;
  }
  static pii LCS(string s, string t) {
    SuffixTree st(s + (char)('z' + 1) + t + (char)('z' + 2))
        ;
    st.lcs(0, sz(s), sz(s) + 1 + sz(t), 0);
    return st.best;
  }
};
```

Template.cpp

"bits/stdc++.h" a51dcb, 38 lines

```
using namespace std;
typedef long long ll;
typedef long double ld; // change to double if appropriate
typedef pair<int, int> pi;
typedef pair<ll, ll> pl;
typedef pair<ld, ld> pd;

typedef vector<int> vi;
typedef vector<ld> vd;
typedef vector<ll> vl;
typedef vector<pi> vpi;
typedef vector<pl> vpl;

#define FOR(i, a, b) for (int i = a; i < (b); i++)
#define F0R(i, a) for (int i = 0; i < (a); i++)
#define FORd(i, a, b) for (int i = (b) - 1; i >= (a); i--)
#define F0Rd(i, a) for (int i = (a) - 1; i >= 0; i--)
#define trav(a, x) for (auto &a : x)
#define sz(x) (int)(x).size()
#define pb push_back
#define f first
#define s second
#define lb lower_bound
#define ub upper_bound
#define all(x) x.begin(), x.end()
#define ins insert

const char nl = '\n';

void solve() {
}

int main() {
  ios_base::sync_with_stdio(0); cin.tie(0);
  solve();
  return 0;
}
```

Treap.cpp

396fd3, 50 lines

```
struct node
{
    node *ch[2]; //ch[0] = left child; ch[1] = right child;
    int ct,priority,size,key;
    int lsize(){return(ch[0] == NULL)?0:ch[0]->size;}
    int rsize(){return(ch[1] == NULL)?0:ch[1]->size;}
};
typedef node* tree;
void update(tree & o){//this part depends on the actual info
      to maintain
    o->size = o->ct; o->size += o->lsize(); o->size += o->
        rsize();
}
void rotate(tree & o,int dir){ //dir = 0: left rotate
    tree temp = o->ch[dir^1]; o->ch[dir^1] = temp->ch[dir];
        temp->ch[dir] = o;
    update(o); update(temp); o = temp;
}
void insert(tree & o,int key){
    if (o == NULL){
        o = new node;
        o->size = o->ct = 1;o->priority = rand();o->ch[0]=o
            ->ch[1]=NULL;o->key=key;
    }
    else if (key == o->key){
        o->ct++;o->size++;
    }
    else{
        int dir = (key<o->key)?0:1;
        insert(o->ch[dir],key);
        if (o->ch[dir]->priority>o->priority) rotate(o,dir^
            1);
        update(o);
    }
}
void remove(tree & o,int key){
    if (key == o->key){
        if (o->ct > 1){
            o->ct--;o->size--;return;
        }
        else if (o->ch[0]==NULL||o->ch[1]==NULL){
            int d = (o->ch[0]==NULL)?0:1;
            tree temp = o; o = o->ch[d^1]; delete temp;
        }
        else{
            int d =(o->ch[0]->priority > o->ch[1]->priority)
                ?1:0;
            rotate(o,d);remove(o,key);
        }
    }
    else{
        int d = (key<o->key)?0:1;
        remove(o->ch[d],key);
    }
    if (o) update(o);
}
```

validate.cpp

26fb12, 22 lines

```
#!/usr/bin/env bash

for ((testNum=0;testNum<$4;testNum++))
do
    ./$3 > input
    ./$1 < input > out
    cat input out > data
    ./$2 < data > res
    result=$(cat res)
    if [ "${result:0:2}" != "OK" ];
    then
        echo "Error found!"
        echo "Input:"
        cat input
        echo "Output:"
        cat out
        echo "Validator Result:"
        cat res
        exit
    fi
done
echo Passed $4 tests
```

Vimrc.cpp

11 lines

```
source $VIMRUNTIME/defaults.vim
set ts=4 sw=4 ai cin nu cino+=L0
syntax on

inoremap {<CR>  {<CR>}<Esc>O
imap jk         <Esc>

" Select region and then type :Hash to hash your selection.
" Useful for verifying that there aren't mistypes.
ca Hash w !cpp -dD -P -fpreprocessed \| tr -d '[:space:]' \
 \| md5sum \| cut -c-6
```

Voronoi.cpp

5f5301, 105 lines

```
// Source: http://web.mit.edu/~ecprice/acm/acm08/notebook.
    html#file7
#define MAXN 1024
#define INF 1000000

//Voronoi diagrams: O(N^2*LogN)
//Convex hull: O(N*LogN)
typedef struct {
  int id;
  double x;
  double y;
  double ang;
} chp;

int n;
double x[MAXN], y[MAXN]; // Input points
chp inv[2*MAXN]; // Points after inversion (to be given to
    Convex Hull)
int vors;
int vor[MAXN]; // Set of points in convex hull;
               //starts at lefmost; last same as first!!
```

```
PT ans[MAXN][2];

int chpcmp(const void *aa, const void *bb) {
  double a = ((chp *)aa)->ang;
  double b = ((chp *)bb)->ang;
  if (a<b) return -1;
  else if (a>b) return 1;
  else return 0; // Might be better to include a
                 // tie-breaker on distance, instead of the
                    cheap hack below
}

int orient(chp *a, chp *b, chp *c) {
  double s = a->x*(b->y-c->y) + b->x*(c->y-a->y) + c->x*(a->
     y-b->y);
  if (s>0) return 1;
  else if (s<0) return -1;
  else if (a->ang==b->ang && a->ang==c->ang) return -1; //
       Cheap hack
         //for points with same angles
  else return 0;
}

//the pt argument must have the points with precomputed
     angles (atan2()'s)
//with respect to a point on the inside (e.g. the center of
     mass)
int convexHull(int n, chp *pt, int *ans) {
  int i, j, st, anses=0;

  qsort(pt, n, sizeof(chp), chpcmp);
  for (i=0; i<n; i++) pt[n+i] = pt[i];
  st = 0;
  for (i=1; i<n; i++) { // Pick leftmost (bottommost)
                        //point to make sure it's on the
                           convex hull
    if (pt[i].x<pt[st].x || (pt[i].x==pt[st].x && pt[i].y<pt
        [st].y)) st = i;
  }
  ans[anses++] = st;
  for (i=st+1; i<=st+n; i++) {
    for (j=anses-1; j; j--) {
      if (orient(pt+ans[j-1], pt+ans[j], pt+i)>=0) break;
      // Should change the above to strictly greater,
      // if you don't want points that lie on the side (not
         on a vertex) of the hull
      // If you really want them, you might also put an
         epsilon in orient
    }
    ans[j+1] = i;
    anses = j+2;
  }
  for (i=0; i<anses; i++) ans[i] = pt[ans[i]].id;
  return anses;
}

int main(void) {
  int i, j, jj;
  double tmp;

  scanf("%d", &n);
  for (i=0; i<n; i++) scanf("%lf %lf", &x[i], &y[i]);
  for (i=0; i<n; i++) {
    x[n] = 2*(-INF)-x[i]; y[n] = y[i];
    x[n+1] = x[i]; y[n+1] = 2*INF-y[i];
    x[n+2] = 2*INF-x[i]; y[n+2] = y[i];
    x[n+3] = x[i]; y[n+3] = 2*(-INF)-y[i];
    for (j=0; j<n+4; j++) if (j!=i) {
      jj = j - (j>i);
      inv[jj].id = j;
      tmp = (x[j]-x[i])*(x[j]-x[i]) + (y[j]-y[i])*(y[j]-y[i
         ]);
      inv[jj].x = (x[j]-x[i])/tmp;
      inv[jj].y = (y[j]-y[i])/tmp;
      inv[jj].ang = atan2(inv[jj].y, inv[jj].x);
    }
    vors = convexHull(n+3, inv, vor);
    // Build bisectors
    for (j=0; j<vors; j++) {
      ans[j][0].x = (x[i]+x[vor[j]])/2;
      ans[j][0].y = (y[i]+y[vor[j]])/2;
      ans[j][1].x = ans[j][0].x - (y[vor[j]]-y[i]);
      ans[j][1].y = ans[j][0].y + (x[vor[j]]-x[i]);
    }
    printf("Around (%lf, %lf)\n", x[i], y[i]);
    // List all intersections of the bisectors
    for (j=1; j<vors; j++) {
      PT vv;
      vv = ComputeLineIntersection(ans[j-1][0], ans[j-1][1],
         ans[j][0], ans[j][1]);
      printf("%lf, %lf\n", vv.x, vv.y);
    }
    printf("\n");
  }
  return 0;
}
```

Z-algorithm.cpp

9dc088, 13 lines

```
vector<int> z_function(string s) {
    int n = sz(s);
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r)
            z[i] = min (r - i + 1, z[i - l]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > r)
            l = i, r = i + z[i] - 1;
    }
    return z;
}
```

hash.sh

3 lines

```
# Hashes a file, ignoring all whitespace and comments. Use
    for
# verifying that code was correctly typed.
cpp -dD -P -fpreprocessed | tr -d '[:space:]'| md5sum |cut -
    c-6
```

Math Hints (2)

Newton's Method: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.

Lagrange Multiplier: Let $f : \mathbb{R}^n \to \mathbb{R}$ be the objective function, $g : \mathbb{R}^n \to \mathbb{R}^c$ be the constraints function, let x^* be an optimal solution to the optimization problem such that $Dg(x^*) = c$: maximize $f(x)$ subject to $g(x) = 0$. There there exists λ such that $Df(x^*) = \lambda^* \top Dg(x^*)$.

Burnside's Lemma: Let G be a finite group acting on set X. Let X^g denote the set of elements in X that are fixed by g. Then the number of orbits is given by $|X/G| = \frac{1}{|G|}\sum_{g\in G}|X^g|$.

Linear Time Inverses Modulo p: For $i \geq 2$, $i^{-1} = -\lfloor \frac{p}{i} \rfloor (p \bmod i)^{-1}$.

Quadratic Residue: $\left(\frac{a}{p}\right) = a^{(p-1)/2}$. If $x^2 \equiv a (\mathrm{mod} p)$ has a solution, then $\left(\frac{a}{p}\right) = 1$.

LGV Lemma: Assume $G = (V, E)$ is a DAG. Let $\omega(P)$ be the product of edge weights on path P. Let $e(u, v) := \sum_{P:u\to v}\omega(P)$ be the sum of $\omega(P)$ for all paths from u to v. The set of sources $A \subseteq V$ and set of sinks $B \subseteq V$. A collection of disjoint paths $A \to B$ consists of n paths S_i such that S_i is a path from A_i to $B_{\sigma(S)_i}$ such that for any $i \neq j$, S_i and S_j does not share a common vertex. Then if we let

$$M = \begin{bmatrix} e(A_1, B_1) & e(A_1, B_2) & \cdots e(A_1, B_n) & \\ \vdots & \vdots & \ddots & \vdots \\ e(A_n, B_1) & e(A_n, B_2) & \cdots & e(A_n, B_n) \end{bmatrix},$$

then $\det M = \sum_{S:A\to B} \operatorname{sgn} \sigma(S) \prod_{i=1}^n \omega(S_i)$ where $S : A \to B$ denotes a set of disjoint paths S from A to B.

Network Flow with Lower/Upper Bounds: Suppose the flow must satisfy $b(u, v) \leq f(u, v) \leq c(u, v)$, and have conservation of flows over vertices.

Variant 1: No source/sink (i.e. flow at all vertices must be balanced), check if there is a feasible flow: create a graph G'. For any edge $u \to v$, add an edge with capacity $c(u, v) - b(u, v)$. Now assume initially, at vertex u, the sum of capacities of edges into u minus the sum of capacities out of u is M. If $M > 0$, add an edge from super source S to u with capacity M. If $M < 0$, add an edge from u to the super sink T with capacity $-M$.

Variant 2: Feasible flow with sources and sinks: Add an edge from original sink t to source s with capacity $+\infty$.

Variant 3: Maximum flow with sources and sinks: first, check if there exists a feasible flow. Then augment using source s and sink t (i.e. run Dinic again with source s and sink t).

Variant 4: Minimum flow with sources and sinks: find the feasible flow first, remove the edge from sink to source (the current flow on the edge is the size of the original flow), and then run the maximum flow from the sink t to the source s to see how much we may get rid off from the original flow.

Pick's Theorem: Suppose a polygon has integer coordinates for all of its vertices; let i be the number of integer points that are interior to the polyon, b be the number of integer points on its boundary, the area of the polygon is $A = i + \frac{b}{2} - 1$.

Mobius Transformation/Circle Inversion: Mobius transformations $f : \hat{\mathbb{C}} \to \hat{\mathbb{C}}$ are specified by $f(z) = \frac{az+b}{cz+d}$.

Dilworth's Theorem: For a partially ordered set S, the maximum size of an antichain is equal to the minimum number of chains (i.e. any two elements are comparable) required to cover S.

LP Duality: Suppose the primal linear program is given by maximize $\mathbf{c}^\top \mathbf{x}$ subject to $A\mathbf{x} \le \mathbf{b}, \mathbf{x} \ge 0$, the dual progarm is given by minimize $\mathbf{b}^\top \mathbf{y}$ subject to $A^\top \mathbf{y} \ge \mathbf{c}, \mathbf{y} \ge 0$.

Mobius Inversion: If g, f are arithmetic functions satisfying $g(n) = \sum_{d|n} f(d)$ for $n \ge 1$, then $f(n) = \sum_{d|n} \mu(d) g(\frac{n}{d})$.

Number of Points on Lattice Convex Polygon: A convex lattice polygon with coordinates in $[0, N]$ has at most $O(N^{2/3})$ points.

Green's Theorem: Let C be a positively oriented, smooth, simple closed curve and let D be the region bounded by C. If L and M are functions of (x, y) defined on an open region containing D and having continuous partial derivatives, then

$$\int_C (Ldx + Mdy) = \int_D (\frac{\partial M}{\partial x} - \frac{\partial L}{\partial y}) dxdy.$$

Polynomial Division: Suppose we are given polynomials $f(x), g(x)$, and we want to write $f(x)$ as $f(x) = Q(x)g(x) + R(x)$. Let $f^R(x) = x^{\deg f} f(\frac{1}{x})$ (i.e. reverse the coefficients of the polynomial). Let $n = \deg f, m = \deg g$. Then

$$f^R(x) \equiv Q^R(x) g^R(x) \mod x^{n-m+1}/$$