Columbia University: CU Later Team Reference Document

Kevin Yang, Innokentiy Kaurov, Eric Yuang Shao

May 21th 2024

Contents

Templates

Ken's template

```cpp
#include <bits/stdc++.h>
using namespace std;
#define all(v) (v).begin(), (v).end()
typedef long long ll;
typedef long double ld;
typedef vector<int> vi;
typedef vector<ll> vll;
typedef pair<int, int> pii;
typedef pair<ll, ll> pll;
#define pb push_back
#define sz(x) (int)(x).size()
#define fi first
#define se second
#define forn(i, n) for (int i = 0; i < int(n); i++)
#define endl '\n'
```

Kevin's template

```cpp
// paste Ken's Template, minus last line
const char nl = '\n';
ll k, n, m, u, v, w, x, y, z;
string s;

bool multiTest = 1;
void solve(int tt){
}

int main(){
  ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);
  cout<<fixed<< setprecision(14);

  int t = 1;
  if (multiTest) cin >> t;
  forn(ii, t) solve(ii);
}
```

Kevin's Template Extended

- to type after the start of the contest

```cpp
typedef pair<double, double> pdd;
const ld PI = acosl(-1);
const ll mod7 = 1e9 + 7;
const ll mod9 = 998244353;
const ll INF = 2*1024*1024*1023;
#pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template<class T> using ordered_set = tree<T, null_type,
  ↪ less<T>, rb_tree_tag, tree_order_statistics_node_update>;
vi d4x = {1, 0, -1, 0};
vi d4y = {0, 1, 0, -1};
vi d8x = {1, 0, -1, 0, 1, 1, -1, -1};
vi d8y = {0, 1, 0, -1, 1, -1, 1, -1};
mt19937
  ↪ rng(chrono::steady_clock::now().time_since_epoch().count());
```

Geometry

Point and vector basics

```cpp
const ld EPS = 1e-9;

struct pt{
  ld x, y;
  pt() : x(0), y(0) {}
  pt(ld x_, ld y_) : x(x_), y(y_) {}

  pt operator+ (pt rhs) const{
    return pt(x + rhs.x, y + rhs.y); }
  pt operator- (pt rhs) const{
    return pt(x - rhs.x, y - rhs.y); }
  pt operator* (ld rhs) const{
    return pt(x * rhs, y * rhs); }
  pt operator/ (ld rhs) const{
    return pt(x / rhs, y / rhs); }
  pt ort() const{
    return pt(-y, x); }
  ld abs2() const{
    return x * x + y * y; }
  ld len() const{
    return sqrtl(abs2()); }
  pt unit() const{
    return pt(x, y) / len(); }
  pt rotate(ld a) const{
    return pt(x * cosl(a) - y * sinl(a), x * sinl(a) + y *
  ↪  cosl(a));
  }
  friend ostream& operator<<(ostream& os, pt p){
    return os << "(" << p.x << "," << p.y << ")";
  }

  bool operator< (pt rhs) const{
    return make_pair(x, y) < make_pair(rhs.x, rhs.y);
  }
  bool operator== (pt rhs) const{
    return abs(x - rhs.x) < EPS && abs(y - rhs.y) < EPS;
  }
};

ld sq(ld a){
  return a * a; }
ld dot(pt a, pt b){
  return a.x * b.x + a.y * b.y; }
ld cross(pt a, pt b){
  return a.x * b.y - a.y * b.x; }
ld dist(pt a, pt b){
  return (a - b).len(); }
bool acw(pt a, pt b){
  return cross(a, b) > -EPS; }
bool cw(pt a, pt b){
  return cross(a, b) < EPS; }
int sgn(ld x){
  return (x > EPS) - (x < EPS); } // for integer: EPS = 0
int half(pt p) { return p.y != 0 ? sgn(p.y) : sgn(p.x); } //
  ↪ +1: [0, pi), -1: [pi, 2*pi)
bool angle_comp(pt a, pt b) { int A = half(a), B = half(b);
  return A == B ? cross(a, b) > 0 : A > B; }
```

Line basics

```cpp
struct line{
  ld a, b, c;
  line() : a(0), b(0), c(0) {}
  line(ld a_, ld b_, ld c_) : a(a_), b(b_), c(c_) {}
  line(pt p1, pt p2){
    a = p1.y - p2.y;
    b = p2.x - p1.x;
    c = -a * p1.x - b * p1.y;
  }
};

ld det(ld a11, ld a12, ld a21, ld a22){
  return a11 * a22 - a12 * a21;
}
bool parallel(line l1, line l2){
  return abs(cross(pt(l1.a, l1.b), pt(l2.a, l2.b))) < EPS;
}
bool operator==(line l1, line l2){
  return parallel(l1, l2) &&
  abs(det(l1.b, l1.c, l2.b, l2.c)) < EPS &&
  abs(det(l1.a, l1.c, l2.a, l2.c)) < EPS;
}
```

Line and segment intersections

```
1   // {p, 0} - unique intersection, {p, 1} - infinite, {p, 2} -
↪     none
2   pair<pt, int> line_inter(line l1, line l2){
3     if (parallel(l1, l2)){
4       return {pt(), l1 == l2? 1 : 2};
5     }
6     return {pt(
7       det(-l1.c, l1.b, -l2.c, l2.b) / det(l1.a, l1.b, l2.a,
↪       l2.b),
8       det(l1.a, -l1.c, l2.a, -l2.c) / det(l1.a, l1.b, l2.a,
↪       l2.b)
9     ), 0};
10  }
11
12
13  // Checks if p lies on ab
14  bool is_on_seg(pt p, pt a, pt b){
15    return abs(cross(p - a, p - b)) < EPS && dot(p - a, p - b) <
↪     EPS;
16  }
17
18  /*
19  If a unique intersection point between the line segments going
↪    from a to b and from c to d exists then it is returned.
20  If no intersection point exists an empty vector is returned.
21  If infinitely many exist a vector with 2 elements is returned,
↪    containing the endpoints of the common line segment.
22  */
23  vector<pt> segment_inter(pt a, pt b, pt c, pt d) {
24    auto oa = cross(d - c, a - c), ob = cross(d - c, b - c), oc
↪     = cross(b - a, c - a), od = cross(b - a, d - a);
25    if (sgn(oa) * sgn(ob) < 0 && sgn(oc) * sgn(od) < 0) return
↪     {(a * ob - b * oa) / (ob - oa)};
26    set<pt> s;
27    if (is_on_seg(a, c, d)) s.insert(a);
28    if (is_on_seg(b, c, d)) s.insert(b);
29    if (is_on_seg(c, a, b)) s.insert(c);
30    if (is_on_seg(d, a, b)) s.insert(d);
31    return {all(s)};
32  }
```

Distances from a point to line and segment

```
1   // Distance from p to line ab
2   ld line_dist(pt p, pt a, pt b){
3     return cross(b - a, p - a) / (b - a).len();
4   }
5
6   // Distance from p to segment ab
7   ld segment_dist(pt p, pt a, pt b){
8     if (a == b) return (p - a).len();
9     auto d = (a - b).abs2(), t = min(d, max((ld)0, dot(p - a, b
↪     - a)));
10    return ((p - a) * d - (b - a) * t).len() / d;
11  }
```

Polygon area and Centroid

```
1   pair<pt,ld> cenArea(const vector<pt>& v) { assert(sz(v) >= 3);
2     pt cen(0, 0); ld area = 0;
3     forn(i,sz(v)) {
4       int j = (i+1)%sz(v); ld a = cross(v[i],v[j]);
5       cen = cen + a*(v[i]+v[j]); area += a; }
6     return {cen/area/(ld)3,area/2}; // area is SIGNED
7   }
```

Convex hull

- Complexity: $O(n \log n)$.

```
1   vector<pt> convex_hull(vector<pt> pts){
2     sort(all(pts));
3     pts.erase(unique(all(pts)), pts.end());
```

```
4     vector<pt> up, down;
5     for (auto p : pts){
6       while (sz(up) > 1 && acw(up.end()[-1] - up.end()[-2], p -
↪       up.end()[-2])) up.pop_back();
7       while (sz(down) > 1 && cw(down.end()[-1] - down.end()[-2],
↪       p - down.end()[-2])) down.pop_back();
8       up.pb(p), down.pb(p);
9     }
10    for (int i = sz(up) - 2; i >= 1; i--) down.pb(up[i]);
11    return down;
12  }
```

Point location in a convex polygon

- Complexity: $O(n)$ precalculation and $O(\log n)$ query.

```
1   void prep_convex_poly(vector<pt>& pts){
2     rotate(pts.begin(), min_element(all(pts)), pts.end());
3   }
4
5   // 0 - Outside, 1 - Exclusively Inside, 2 - On the Border
6   int in_convex_poly(pt p, vector<pt>& pts){
7     int n = sz(pts);
8     if (!n) return 0;
9     if (n <= 2) return is_on_seg(p, pts[0], pts.back());
10    int l = 1, r = n - 1;
11    while (r - l > 1){
12      int mid = (l + r) / 2;
13      if (acw(pts[mid] - pts[0], p - pts[0])) l = mid;
14      else r = mid;
15    }
16    if (!in_triangle(p, pts[0], pts[l], pts[l + 1])) return 0;
17    if (is_on_seg(p, pts[l], pts[l + 1]) ||
18      is_on_seg(p, pts[0], pts.back()) ||
19      is_on_seg(p, pts[0], pts[1])
20    ) return 2;
21    return 1;
22  }
```

Point location in a simple polygon

- Complexity: $O(n)$.

```
1   // 0 - Outside, 1 - Exclusively Inside, 2 - On the Border
2   int in_simple_poly(pt p, vector<pt>& pts){
3     int n = sz(pts);
4     bool res = 0;
5     for (int i = 0; i < n; i++){
6       auto a = pts[i], b = pts[(i + 1) % n];
7       if (is_on_seg(p, a, b)) return 2;
8       if (((a.y > p.y) - (b.y > p.y)) * cross(b - p, a - p) >
↪       EPS){
9         res ^= 1;
10      }
11    }
12    return res;
13  }
```

Minkowski Sum

- For two convex polygons P and Q, returns the set of points $(p + q)$, where $p \in P, q \in Q$.
- This set is also a convex polygon.
- Complexity: $O(n)$.

```
1   void minkowski_rotate(vector<pt>& P){
2     int pos = 0;
3     for (int i = 1; i < sz(P); i++){
4       if (abs(P[i].y - P[pos].y) <= EPS){
5         if (P[i].x < P[pos].x) pos = i;
6       }
7       else if (P[i].y < P[pos].y) pos = i;
8     }
9     rotate(P.begin(), P.begin() + pos, P.end());
10  }
```

```
11  // P and Q are strictly convex, points given in
    ↪  counterclockwise order.
12  vector<pt> minkowski_sum(vector<pt> P, vector<pt> Q){
13    minkowski_rotate(P);
14    minkowski_rotate(Q);
15    P.pb(P[0]);
16    Q.pb(Q[0]);
17    vector<pt> ans;
18    int i = 0, j = 0;
19    while (i < sz(P) - 1 || j < sz(Q) - 1){
20      ans.pb(P[i] + Q[j]);
21      ld curmul;
22      if (i == sz(P) - 1) curmul = -1;
23      else if (j == sz(Q) - 1) curmul = +1;
24      else curmul = cross(P[i + 1] - P[i], Q[j + 1] - Q[j]);
25      if (abs(curmul) < EPS || curmul > 0) i++;
26      if (abs(curmul) < EPS || curmul < 0) j++;
27    }
28    return ans;
29  }
```

Half-plane intersection

- Given N half-plane conditions in the form of a ray, computes the vertices of their intersection polygon.
- Complexity: $O(N \log N)$.
- A ray is defined by a point p and direction vector dp. The half-plane is to the **left** of the direction vector.

```
1   // Extra functions needed: point operations, dot, cross
2   const ld EPS = 1e-9;
3
4   int sgn(ld a){
5     return (a > EPS) - (a < -EPS);
6   }
7   int half(pt p){
8     return p.y != 0? sgn(p.y) : -sgn(p.x);
9   }
10  bool angle_comp(pt a, pt b){
11    int A = half(a), B = half(b);
12    return A == B? cross(a, b) > 0 : A < B;
13  }
14  struct ray{
15    pt p, dp; // origin, direction
16    ray(pt p_, pt dp_){
17      p = p_, dp = dp_;
18    }
19    pt isect(ray l){
20      return p + dp * (cross(l.dp, l.p - p) / cross(l.dp, dp));
21    }
22    bool operator<(ray l){
23      return angle_comp(dp, l.dp);
24    }
25  };
26  vector<pt> half_plane_isect(vector<ray> rays, ld DX = 1e9, ld
    ↪  DY = 1e9){
27    // constrain the area to [0, DX] x [0, DY]
28    rays.pb({pt(0, 0), pt(1, 0)});
29    rays.pb({pt(DX, 0), pt(0, 1)});
30    rays.pb({pt(DX, DY), pt(-1, 0)});
31    rays.pb({pt(0, DY), pt(0, -1)});
32    sort(all(rays));
33    {
34      vector<ray> nrays;
35      for (auto t : rays){
36        if (nrays.empty() || cross(nrays.back().dp, t.dp) >
    ↪  EPS){
37          nrays.pb(t);
38          continue;
39        }
40        if (cross(t.dp, t.p - nrays.back().p) > 0) nrays.back()
    ↪  = t;
41      }
42      swap(rays, nrays);
43    }
44    auto bad = [&] (ray a, ray b, ray c){
```

```
45      pt p1 = a.isect(b), p2 = b.isect(c);
46      if (dot(p2 - p1, b.dp) <= EPS){
47        if (cross(a.dp, c.dp) <= 0) return 2;
48        return 1;
49      }
50      return 0;
51    };
52    #define reduce(t) \
53      while (sz(poly) > 1){ \
54        int b = bad(poly[sz(poly) - 2], poly.back(), t); \
55        if (b == 2) return {}; \
56        if (b == 1) poly.pop_back(); \
57        else break; \
58      }
59    deque<ray> poly;
60    for (auto t : rays){
61      reduce(t);
62      poly.pb(t);
63    }
64    for (;; poly.pop_front()){
65      reduce(poly[0]);
66      if (!bad(poly.back(), poly[0], poly[1])) break;
67    }
68    assert(sz(poly) >= 3); // expect nonzero area
69    vector<pt> poly_points;
70    for (int i = 0; i < sz(poly); i++){
71      poly_points.pb(poly[i].isect(poly[(i + 1) % sz(poly)]));
72    }
73    return poly_points;
74  }
```

Circles

- Finds minimum enclosing circle of vector of points in expected $O(N)$

```
1   // necessary point functions
2   ld sq(ld a) { return a*a; }
3   pt operator+(const pt& l, const pt& r) {
4     return pt(l.x+r.x,l.y+r.y); }
5   pt operator*(const pt& l, const ld& r) {
6     return pt(l.x*r,l.y*r); }
7   pt operator*(const ld& l, const pt& r) { return r*l; }
8   ld abs2(const pt& p) { return sq(p.x)+sq(p.y); }
9   ld abs(const pt& p) { return sqrt(abs2(p)); }
10  pt conj(const pt& p) { return pt(p.x,-p.y); }
11  pt operator-(const pt& l, const pt& r) {
12    return pt(l.x-r.x,l.y-r.y); }
13  pt operator*(const pt& l, const pt& r) {
14    return pt(l.x*r.x-l.y*r.y,l.y*r.x+l.x*r.y); }
15  pt operator/(const pt& l, const ld& r) {
16    return pt(l.x/r,l.y/r); }
17  pt operator/(const pt& l, const pt& r) {
18    return l*conj(r)/abs2(r); }
19
20  // circle code
21  using circ = pair<pt,ld>;
22
23  circ ccCenter(pt a, pt b, pt c) {
24    b = b-a; c = c-a;
25    pt res = b*c*(conj(c)-conj(b))/(b*conj(c)-conj(b)*c);
26    return {a+res,abs(res)};
27  }
28
29  circ mec(vector<pt> ps) {
30    // expected O(N)
31    shuffle(all(ps), rng);
32    pt o = ps[0]; ld r = 0, EPS = 1+1e-8;
33    forn(i,sz(ps)) if (abs(o-ps[i]) > r*EPS) {
34      o = ps[i], r = 0; // point is on MEC
35      forn(j,i) if (abs(o-ps[j]) > r*EPS) {
36        o = (ps[i]+ps[j])/2, r = abs(o-ps[i]);
37        forn(k,j) if (abs(o-ps[k]) > r*EPS)
38          tie(o,r) = ccCenter(ps[i],ps[j],ps[k]);
39      }
40    }
```

```
41    return {o,r};
42  }
```

- Circle tangents, external and internal

```
1  pt unit(const pt& p) { return p * (1/abs(p)); }
2
3  pt tangent(pt p, circ c, int t = 0) {
4    c.se = abs(c.se); // abs needed because internal calls y.s <
       ↪ 0
5    if (c.se == 0) return c.fi;
6    ld d = abs(p-c.fi);
7    pt a = pow(c.se/d,2)*(p-c.fi)+c.fi;
8    pt b = sqrt(d*d-c.se*c.se)/d*c.se*unit(p-c.fi)*pt(0,1);
9    return t == 0 ? a+b : a-b;
10  }
11  vector<pair<pt,pt>> external(circ a, circ b) {
12    vector<pair<pt,pt>> v;
13    if (a.se == b.se) {
14      pt tmp = unit(a.fi-b.fi)*a.se*pt(0, 1);
15      v.emplace_back(a.fi+tmp,b.fi+tmp);
16      v.emplace_back(a.fi-tmp,b.fi-tmp);
17    } else {
18      pt p = (b.se*a.fi-a.se*b.fi)/(b.se-a.se);
19      forn(i,2) v.emplace_back(tangent(p,a,i),tangent(p,b,i));
20    }
21    return v;
22  }
23  vector<pair<pt,pt>> internal(circ a, circ b) {
24    return external({a.fi,-a.se},b); }
```

Strings

```
1  vi prefix_function(string s){
2    int n = sz(s);
3    vi pi(n);
4    for (int i = 1; i < n; i++){
5      int k = pi[i - 1];
6      while (k > 0 && s[i] != s[k]){
7        k = pi[k - 1];
8      }
9      pi[i] = k + (s[i] == s[k]);
10    }
11    return pi;
12  }
13  // Returns the positions of the first character
14  vi kmp(string s, string k){
15    string st = k + "#" + s;
16    vi res;
17    auto pi = prefix_function(st);
18    forn(i, sz(st)){
19      if (pi[i] == sz(k)){
20        res.pb(i - 2 * sz(k));
21      }
22    }
23    return res;
24  }
25  vi z_function(string s){
26    int n = sz(s);
27    vi z(n);
28    int l = 0, r = 0;
29    for (int i = 1; i < n; i++){
30      if (r >= i) z[i] = min(z[i - l], r - i + 1);
31      while (i + z[i] < n && s[z[i]] == s[i + z[i]]){
32        z[i]++;
33      }
34      if (i + z[i] - 1 > r){
35        l = i, r = i + z[i] - 1;
36      }
37    }
38    return z;
39  }
```

Manacher's algorithm

```
1  /*
2  Finds longest palindromes centered at each index
3  even[i] = d --> [i - d, i + d - 1] is a max-palindrome
4  odd[i] = d  --> [i - d, i + d] is a max-palindrome
5  */
6  pair<vi, vi> manacher(string s) {
7    vector<char> t{'^', '#'};
8    for (char c : s) t.push_back(c), t.push_back('#');
9    t.push_back('$');
10    int n = t.size(), r = 0, c = 0;
11    vi p(n, 0);
12    for (int i = 1; i < n - 1; i++) {
13      if (i < r + c) p[i] = min(p[2 * c - i], r + c - i);
14      while (t[i + p[i] + 1] == t[i - p[i] - 1]) p[i]++;
15      if (i + p[i] > r + c) r = p[i], c = i;
16    }
17    vi even(sz(s)), odd(sz(s));
18    forn(i, sz(s)){
19      even[i] = p[2 * i + 1] / 2, odd[i] = p[2 * i + 2] / 2;
20    }
21    return {even, odd};
22  }
```

Aho-Corasick Trie

- Given a set of strings, constructs a trie with suffix links.
- For a particular node, *link* points to the longest proper suffix of this node that's contained in the trie.
- *nxt* encodes suffix links in a compressed format:
  - If vertex v has a child by letter x, then $trie[v].nxt[x]$ points to that child.
  - If vertex v doesn't have such child, then $trie[v].nxt[x]$ points to the suffix link of that child if we would actually have it.
- **Facts:** suffix link graph can be seen as a tree; terminal link tree has height $O(\sqrt{N})$, where N is the sum of strings' lengths.
- **Usage:** add all strings, then call *add_links*().

```
1  const int S = 26;
2
3  // Function converting char to int.
4  int ctoi(char c){
5    return c - 'a';
6  }
7
8  // To add terminal links, use DFS
9  struct Node{
10    vi nxt;
11    int link;
12    bool terminal;
13
14    Node() {
15      nxt.assign(S, -1), link = 0, terminal = 0;
16    }
17  };
18
19  vector<Node> trie(1);
20
21  // add_string returns the terminal vertex.
22  int add_string(string& s){
23    int v = 0;
24    for (auto c : s){
25      int cur = ctoi(c);
26      if (trie[v].nxt[cur] == -1){
27        trie[v].nxt[cur] = sz(trie);
28        trie.emplace_back();
29      }
30      v = trie[v].nxt[cur];
31    }
32    trie[v].terminal = 1;
33    return v;
```

```
34   }
35
36   void add_links(){
37     queue<int> q;
38     q.push(0);
39     while (!q.empty()){
40       auto v = q.front();
41       int u = trie[v].link;
42       q.pop();
43       forn(i, S){
44         int& ch = trie[v].nxt[i];
45         if (ch == -1){
46           ch = v? trie[u].nxt[i] : 0;
47         }
48         else{
49           trie[ch].link = v? trie[u].nxt[i] : 0;
50           q.push(ch);
51         }
52       }
53     }
54   }
55
56   bool is_terminal(int v){
57     return trie[v].terminal;
58   }
59
60   int get_link(int v){
61     return trie[v].link;
62   }
63
64   int go(int v, char c){
65     return trie[v].nxt[ctoi(c)];
66   }
```

Suffix Automaton

- Given a string S, constructs a DAG that is an automaton of all suffixes of S.
- The automaton has $\leq 2n$ nodes and $\leq 3n$ edges.
- Properties (let all paths start at node 0):
  - Every path represents a unique substring of S.
  - A path ends at a terminal node iff it represents a suffix of S.
  - All paths ending at a fixed node v have the same set of right endpoints of their occurences in S.
  - Let $endpos(v)$ represent this set. Then, $link(v) := u$ such that $endpos(v) \subset endpos(u)$ and $|endpos(u)|$ is smallest possible. $link(0) := -1$. Links form a tree.
  - Let $len(v)$ be the longest path ending at v. All paths ending at v have distinct lengths: every length from interval $[len(link(v)) + 1, len(v)]$.
- One of the main applications is dealing with **distinct** substrings. Such problems can be solved with DFS and DP.
- Complexity: $O(|S| \cdot \log |\Sigma|)$. Perhaps replace map with vector if $|\Sigma|$ is small.

```
1   const int MAXLEN = 1e5 + 20;
2
3   struct suffix_automaton{
4     struct state {
5       int len, link;
6       bool terminal = 0, used = 0;
7       map<char, int> next;
8     };
9
10    state st[MAXLEN * 2];
11    int sz = 0, last;
12
13    suffix_automaton(){
14      st[0].len = 0;
15      st[0].link = -1;
16      sz++;
17      last = 0;
18    };
19
20    void extend(char c) {
21      int cur = sz++;
22      st[cur].len = st[last].len + 1;
23      int p = last;
24      while (p != -1 && !st[p].next.count(c)) {
25        st[p].next[c] = cur;
26        p = st[p].link;
27      }
28      if (p == -1) {
29        st[cur].link = 0;
30      } else {
31        int q = st[p].next[c];
32        if (st[p].len + 1 == st[q].len) {
33          st[cur].link = q;
34        } else {
35          int clone = sz++;
36          st[clone].len = st[p].len + 1;
37          st[clone].next = st[q].next;
38          st[clone].link = st[q].link;
39          while (p != -1 && st[p].next[c] == q) {
40            st[p].next[c] = clone;
41            p = st[p].link;
42          }
43          st[q].link = st[cur].link = clone;
44        }
45      }
46      last = cur;
47    }
48
49    void mark_terminal(){
50      int cur = last;
51      while (cur) st[cur].terminal = 1, cur = st[cur].link;
52    }
53  };
54  /*
55  Usage:
56  suffix_automaton sa;
57  for (int i = 0; i < sz(str); i++) sa.extend(str[i]);
58  sa.mark_terminal();
59  */
```

Flows

$O(N^2M)$, on unit networks $O(N^{1/2}M)$

```
1   struct FlowEdge {
2     int from, to;
3     ll cap, flow = 0;
4     FlowEdge(int u, int v, ll cap) : from(u), to(v), cap(cap) {}
5   };
6   struct Dinic {
7     const ll flow_inf = 1e18;
8     vector<FlowEdge> edges;
9     vector<vi> adj;
10    int n, m = 0;
11    int s, t;
12    vi level, ptr;
13    vector<bool> used;
14    queue<int> q;
15    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
16      adj.resize(n);
17      level.resize(n);
18      ptr.resize(n);
19    }
20    void add_edge(int u, int v, ll cap) {
21      edges.emplace_back(u, v, cap);
22      edges.emplace_back(v, u, 0);
23      adj[u].push_back(m);
24      adj[v].push_back(m + 1);
```

```
25        m += 2;
26      }
27    bool bfs() {
28      while (!q.empty()) {
29        int v = q.front();
30        q.pop();
31        for (int id : adj[v]) {
32          if (edges[id].cap - edges[id].flow < 1)
33            continue;
34          if (level[edges[id].to] != -1)
35            continue;
36          level[edges[id].to] = level[v] + 1;
37          q.push(edges[id].to);
38        }
39      }
40      return level[t] != -1;
41    }
42    ll dfs(int v, ll pushed) {
43      if (pushed == 0)
44        return 0;
45      if (v == t)
46        return pushed;
47      for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
48        int id = adj[v][cid];
49        int u = edges[id].to;
50        if (level[v] + 1 != level[u] || edges[id].cap -
     edges[id].flow < 1)
51          continue;
52        ll tr = dfs(u, min(pushed, edges[id].cap -
     edges[id].flow));
53        if (tr == 0)
54          continue;
55        edges[id].flow += tr;
56        edges[id ^ 1].flow -= tr;
57        return tr;
58      }
59      return 0;
60    }
61    ll flow() {
62      ll f = 0;
63      while (true) {
64        fill(level.begin(), level.end(), -1);
65        level[s] = 0;
66        q.push(s);
67        if (!bfs())
68          break;
69        fill(ptr.begin(), ptr.end(), 0);
70        while (ll pushed = dfs(s, flow_inf)) {
71          f += pushed;
72        }
73      }
74      return f;
75    }
76
77    void cut_dfs(int v){
78      used[v] = 1;
79      for (auto i : adj[v]){
80        if (edges[i].flow < edges[i].cap && !used[edges[i].to]){
81          cut_dfs(edges[i].to);
82        }
83      }
84    }
85
86    // Assumes that max flow is already calculated
87    // true -> vertex is in S, false -> vertex is in T
88    vector<bool> min_cut(){
89      used = vector<bool>(n);
90      cut_dfs(s);
91      return used;
92    }
93  };
94  // To recover flow through original edges: iterate over even
     indices in edges.
```

MCMF – maximize flow, then minimize its cost. $O(mn + Fm\log n)$.

```
1   #include <bits/extc++.h> /// include-line, keep-include
2
3   const ll INF = LLONG_MAX / 4;
4
5   struct MCMF {
6     struct edge {
7       int from, to, rev;
8       ll cap, cost, flow;
9     };
10    int N;
11    vector<vector<edge>> ed;
12    vi seen;
13    vll dist, pi;
14    vector<edge*> par;
15
16    MCMF(int N) : N(N), ed(N), seen(N), dist(N), pi(N), par(N)
     {}
17
18    void add_edge(int from, int to, ll cap, ll cost) {
19      if (from == to) return;
20      ed[from].push_back(edge{ from,to,sz(ed[to]),cap,cost,0 });
21      ed[to].push_back(edge{ to,from,sz(ed[from])-1,0,-cost,0
     });
22    }
23
24    void path(int s) {
25      fill(all(seen), 0);
26      fill(all(dist), INF);
27      dist[s] = 0; ll di;
28
29      __gnu_pbds::priority_queue<pair<ll, int>> q;
30      vector<decltype(q)::point_iterator> its(N);
31      q.push({ 0, s });
32
33      while (!q.empty()) {
34        s = q.top().second; q.pop();
35        seen[s] = 1; di = dist[s] + pi[s];
36        for (edge& e : ed[s]) if (!seen[e.to]) {
37          ll val = di - pi[e.to] + e.cost;
38          if (e.cap - e.flow > 0 && val < dist[e.to]) {
39            dist[e.to] = val;
40            par[e.to] = &e;
41            if (its[e.to] == q.end())
42              its[e.to] = q.push({ -dist[e.to], e.to });
43            else
44              q.modify(its[e.to], { -dist[e.to], e.to });
45          }
46        }
47      }
48      forn(i, N) pi[i] = min(pi[i] + dist[i], INF);
49    }
50
51    pair<ll, ll> max_flow(int s, int t) {
52      ll totflow = 0, totcost = 0;
53      while (path(s), seen[t]) {
54        ll fl = INF;
55        for (edge* x = par[t]; x; x = par[x->from])
56          fl = min(fl, x->cap - x->flow);
57
58        totflow += fl;
59        for (edge* x = par[t]; x; x = par[x->from]) {
60          x->flow += fl;
61          ed[x->to][x->rev].flow -= fl;
62        }
63      }
64      forn(i, N) for(edge& e : ed[i]) totcost += e.cost *
     e.flow;
65      return {totflow, totcost/2};
66    }
67
68    // If some costs can be negative, call this before maxflow:
69    void setpi(int s) { // (otherwise, leave this out)
70      fill(all(pi), INF); pi[s] = 0;
71      int it = N, ch = 1; ll v;
```

```
72        while (ch-- && it--)
73          forn(i, N) if (pi[i] != INF)
74            for (edge& e : ed[i]) if (e.cap)
75              if ((v = pi[i] + e.cost) < pi[e.to])
76                pi[e.to] = v, ch = 1;
77        assert(it >= 0); // negative cost cycle
78      }
79  };
80  // Usage: MCMF g(n); g.add_edge(u,v,c,w); g.max_flow(s,t).
81  // To recover flow through original edges: iterate over even
   ↪   indices in edges.
```

Graphs

Kuhn's algorithm for bipartite matching

```
1  /*
2  The graph is split into 2 halves of n1 and n2 vertices.
3  Complexity: O(n1 * m). Usually runs much faster. MUCH
   ↪   FASTER!!!
4  */
5  const int N = 305;
6
7  vi g[N]; // Stores edges from left half to right.
8  bool used[N]; // Stores if vertex from left half is used.
9  int mt[N]; // For every vertex in right half, stores to which
   ↪   vertex in left half it's matched (-1 if not matched).
10
11 bool try_dfs(int v){
12   if (used[v]) return false;
13   used[v] = 1;
14   for (auto u : g[v]){
15     if (mt[u] == -1 || try_dfs(mt[u])){
16       mt[u] = v;
17       return true;
18     }
19   }
20   return false;
21 }
22
23 int main(){
24 // ......
25   for (int i = 1; i <= n2; i++) mt[i] = -1;
26   for (int i = 1; i <= n1; i++) used[i] = 0;
27   for (int i = 1; i <= n1; i++){
28     if (try_dfs(i)){
29       for (int j = 1; j <= n1; j++) used[j] = 0;
30     }
31   }
32   vector<pair<int, int>> ans;
33   for (int i = 1; i <= n2; i++){
34     if (mt[i] != -1) ans.pb({mt[i], i});
35   }
36 }
37
38 // Finding maximal independent set: size = # of nodes - # of
   ↪   edges in matching.
39 // To construct: launch Kuhn-like DFS from unmatched nodes in
   ↪   the left half.
40 // Independent set = visited nodes in left half + unvisited in
   ↪   right half.
41 // Finding minimal vertex cover: complement of maximal
   ↪   independent set.
```

Hungarian algorithm for Assignment Problem

- Given a 1-indexed $(n \times m)$ matrix A, select a number in each row such that each column has at most 1 number selected, and the sum of the selected numbers is minimized.

```
1  int INF = 1e9; // constant greater than any number in the
   ↪   matrix
```

```
2  vi u(n+1), v(m+1), p(m+1), way(m+1);
3  for (int i=1; i<=n; ++i) {
4    p[0] = i;
5    int j0 = 0;
6    vi minv (m+1, INF);
7    vector<bool> used (m+1, false);
8    do {
9      used[j0] = true;
10     int i0 = p[j0],  delta = INF,  j1;
11     for (int j=1; j<=m; ++j)
12       if (!used[j]) {
13         int cur = A[i0][j]-u[i0]-v[j];
14         if (cur < minv[j])
15           minv[j] = cur,  way[j] = j0;
16         if (minv[j] < delta)
17           delta = minv[j],  j1 = j;
18       }
19     for (int j=0; j<=m; ++j)
20       if (used[j])
21         u[p[j]] += delta,  v[j] -= delta;
22       else
23         minv[j] -= delta;
24     j0 = j1;
25   } while (p[j0] != 0);
26   do {
27     int j1 = way[j0];
28     p[j0] = p[j1];
29     j0 = j1;
30   } while (j0);
31 }
32 vi ans (n+1); // ans[i] stores the column selected for row i
33 for (int j=1; j<=m; ++j)
34   ans[p[j]] = j;
35 int cost = -v[0]; // the total cost of the matching
```

Dijkstra's Algorithm

```
1  priority_queue<pair<ll, ll>, vector<pair<ll, ll>>,
   ↪   greater<pair<ll, ll>>> q;
2  dist[start] = 0;
3  q.push({0, start});
4  while (!q.empty()){
5    auto [d, v] = q.top();
6    q.pop();
7    if (d != dist[v]) continue;
8    for (auto [u, w] : g[v]){
9      if (dist[u] > dist[v] + w){
10       dist[u] = dist[v] + w;
11       q.push({dist[u], u});
12     }
13   }
14 }
```

Bellman-Ford Algorithm

- Finds single-source shortest paths with negative edge weights.
- Returns the vector of distances to 0-indexed vertices, or empty vector if a negative cycle is reachable from source.

```
1  const ll bf_inf = 1e18;
2
3  struct edge {
4      ll a, b, w;
5  };
6
7  vector<ll> bellman_ford(int n, vector<edge> edges, int src)
8  {
9      vector<ll> d(n, bf_inf);
10     d[src] = 0;
11     vector<ll> p(n, -1);
12     int x;
13     forn(i, n) {
14         x = -1;
15         for (edge e : edges)
```

```
16        if (d[e.a] < bf_inf)
17            if (d[e.b] > d[e.a] + e.w) {
18                d[e.b] = max(-bf_inf, d[e.a] + e.w);
19                p[e.b] = e.a;
20                x = e.b;
21            }
22    }
23
24    if (x != -1){
25        // negative cycle reachable from src
26        return {};
27    }
28    return d;
29 }
```

Eulerian Cycle DFS

```
1 void dfs(int v){
2   while (!g[v].empty()){
3     int u = g[v].back();
4     g[v].pop_back();
5     dfs(u);
6     ans.pb(v);
7   }
8 }
```

SCC and 2-SAT

```
1 void scc(vector<vi>& g, int* idx) {
2   int n = g.size(), ct = 0;
3   int out[n];
4   vi ginv[n];
5   memset(out, -1, sizeof out);
6   memset(idx, -1, n * sizeof(int));
7   function<void(int)> dfs = [&](int cur) {
8     out[cur] = INT_MAX;
9     for(int v : g[cur]) {
10       ginv[v].push_back(cur);
11       if(out[v] == -1) dfs(v);
12     }
13     ct++; out[cur] = ct;
14   };
15   vi order;
16   for(int i = 0; i < n; i++) {
17     order.push_back(i);
18     if(out[i] == -1) dfs(i);
19   }
20   sort(order.begin(), order.end(), [&](int& u, int& v) {
21     return out[u] > out[v];
22   });
23   ct = 0;
24   stack<int> s;
25   auto dfs2 = [&](int start) {
26     s.push(start);
27     while(!s.empty()) {
28       int cur = s.top();
29       s.pop();
30       idx[cur] = ct;
31       for(int v : ginv[cur])
32         if(idx[v] == -1) s.push(v);
33     }
34   };
35   for(int v : order) {
36     if(idx[v] == -1) {
37       dfs2(v);
38       ct++;
39     }
40   }
41 }
42
43 // 0 => impossible, 1 => possible
44 pair<int,vi> sat2(int n, vector<pii>& clauses) {
45   vi ans(n);
46   vector<vi> g(2*n + 1);
47   for(auto [x, y] : clauses) {
48     x = x < 0 ? -x + n : x;
```

```
49     y = y < 0 ? -y + n : y;
50     int nx = x <= n ? x + n : x - n;
51     int ny = y <= n ? y + n : y - n;
52     g[nx].push_back(y);
53     g[ny].push_back(x);
54   }
55   int idx[2*n + 1];
56   scc(g, idx);
57   for(int i = 1; i <= n; i++) {
58     if(idx[i] == idx[i + n]) return {0, {}};
59     ans[i - 1] = idx[i + n] < idx[i];
60   }
61   return {1, ans};
62 }
```

Finding Bridges

```
1 /*
2 Bridges.
3 Results are stored in a map "is_bridge".
4 For each connected component, call "dfs(starting vertex,
   ↪    starting vertex)".
5 */
6 const int N = 2e5 + 10; // Careful with the constant!
7
8 vi g[N];
9 int tin[N], fup[N], timer;
10 map<pair<int, int>, bool> is_bridge;
11
12 void dfs(int v, int p){
13   tin[v] = ++timer;
14   fup[v] = tin[v];
15   for (auto u : g[v]){
16     if (!tin[u]){
17       dfs(u, v);
18       if (fup[u] > tin[v]){
19         is_bridge[{u, v}] = is_bridge[{v, u}] = true;
20       }
21       fup[v] = min(fup[v], fup[u]);
22     }
23     else{
24       if (u != p) fup[v] = min(fup[v], tin[u]);
25     }
26   }
27 }
```

Virtual Tree

```
1 // order stores the nodes in the queried set
2 sort(all(order), [&] (int u, int v){return tin[u] < tin[v];});
3 int m = sz(order);
4 for (int i = 1; i < m; i++){
5   order.pb(lca(order[i], order[i - 1]));
6 }
7 sort(all(order), [&] (int u, int v){return tin[u] < tin[v];});
8 order.erase(unique(all(order)), order.end());
9 vi stk{order[0]};
10 for (int i = 1; i < sz(order); i++){
11   int v = order[i];
12   while (tout[stk.back()] < tout[v]) stk.pop_back();
13   int u = stk.back();
14   vg[u].pb({v, dep[v] - dep[u]});
15   stk.pb(v);
16 }
```

HLD on Edges DFS

```
1 void dfs1(int v, int p, int d){
2   par[v] = p;
3   for (auto e : g[v]){
4     if (e.fi == p){
5       g[v].erase(find(all(g[v]), e));
6       break;
7     }
8   }
```

```
9      dep[v] = d;
10     sz[v] = 1;
11     for (auto [u, c] : g[v]){
12       dfs1(u, v, d + 1);
13       sz[v] += sz[u];
14     }
15     if (!g[v].empty()) iter_swap(g[v].begin(),
   ↪ max_element(all(g[v]), comp));
16   }
17   void dfs2(int v, int rt, int c){
18     pos[v] = sz(a);
19     a.pb(c);
20     root[v] = rt;
21     forn(i, sz(g[v])){
22       auto [u, c] = g[v][i];
23       if (!i) dfs2(u, rt, c);
24       else dfs2(u, u, c);
25     }
26   }
27   int getans(int u, int v){
28     int res = 0;
29     for (; root[u] != root[v]; v = par[root[v]]){
30       if (dep[root[u]] > dep[root[v]]) swap(u, v);
31       res = max(res, rmq(0, 0, n - 1, pos[root[v]], pos[v]));
32     }
33     if (pos[u] > pos[v]) swap(u, v);
34     return max(res, rmq(0, 0, n - 1, pos[u] + 1, pos[v]));
35   }
```

Centroid Decomposition

```
1   vector<char> res(n), seen(n), sz(n);
2   function<int(int, int)> get_size = [&](int node, int fa) {
3     sz[node] = 1;
4     for (auto& ne : g[node]) {
5       if (ne == fa || seen[ne]) continue;
6       sz[node] += get_size(ne, node);
7     }
8     return sz[node];
9   };
10  function<int(int, int, int)> find_centroid = [&](int node, int
    ↪ fa, int t) {
11    for (auto& ne : g[node])
12      if (ne != fa && !seen[ne] && sz[ne] > t / 2) return
        ↪ find_centroid(ne, node, t);
13    return node;
14  };
15  function<void(int, char)> solve = [&](int node, char cur) {
16    get_size(node, -1); auto c = find_centroid(node, -1,
      ↪ sz[node]);
17    seen[c] = 1, res[c] = cur;
18    for (auto& ne : g[c]) {
19      if (seen[ne]) continue;
20      solve(ne, char(cur + 1)); // we can pass c here to build
        ↪ tree
21    }
22  };
```

Biconnected Components and Block-Cut Tree

- Biconnected components are the ones that have no artic-ulation points.
- They are defined by edge sets that are "bounded" by articulation points in the original graph.
- The corresponding vertex sets are stored in *comps*.
- Block-Cut tree is constructed by creating a fictive node for each component, and attaching edges to its members.
- Articulation points in the original graph are the non-leaf non-fictive nodes in the BC tree.
- Complexity: $O(n)$.

```
1   // Usage: pass in adjacency list in 0-based indexation.
```

```
2   // Return: adjacency list of block-cut tree (nodes 0...n-1
    ↪ represent original nodes, the rest are component nodes).
3   vector<vi> biconnected_components(vector<vi> g) {
4     int n = sz(g);
5     vector<vi> comps;
6     vi stk, num(n), low(n);
7     int timer = 0;
8     // Finds the biconnected components
9     function<void(int, int)> dfs = [&](int v, int p) {
10      num[v] = low[v] = ++timer;
11      stk.pb(v);
12      for (int son : g[v]) {
13        if (son == p) continue;
14        if (num[son]) low[v] = min(low[v], num[son]);
15        else{
16          dfs(son, v);
17          low[v] = min(low[v], low[son]);
18          if (low[son] >= num[v]){
19            comps.pb({v});
20            while (comps.back().back() != son){
21              comps.back().pb(stk.back());
22              stk.pop_back();
23            }
24          }
25        }
26      }
27    };
28    dfs(0, -1);
29    // Build the block-cut tree
30    auto build_tree = [&]() {
31      vector<vi> t(n);
32      for (auto &comp : comps){
33        t.push_back({});
34        for (int u : comp){
35          t.back().pb(u);
36          t[u].pb(sz(t) - 1);
37        }
38      }
39      return t;
40    };
41    return build_tree();
42  }
```

Math

Binary exponentiation

```
1   ll power(ll a, ll b){
2     ll res = 1;
3     for (; b; a = a * a % MOD, b >>= 1){
4       if (b & 1) res = res * a % MOD;
5     }
6     return res;
7   }
```

Matrix Exponentiation: $O(n^3 \log b)$

```
1   const int N = 100, MOD = 1e9 + 7;
2
3   struct matrix{
4     ll m[N][N];
5     int n;
6     matrix(){
7       n = N;
8       memset(m, 0, sizeof(m));
9     };
10    matrix(int n_){
11      n = n_;
12      memset(m, 0, sizeof(m));
13    };
14    matrix(int n_, ll val){
15      n = n_;
16      memset(m, 0, sizeof(m));
17      forn(i, n) m[i][i] = val;
18    };
```

```cpp
19    matrix operator* (matrix oth){
20      matrix res(n);
21      forn(i, n){
22        forn(j, n){
23          forn(k, n){
24            res.m[i][j] = (res.m[i][j] + m[i][k] * oth.m[k][j])
↪ % MOD;
25          }
26        }
27      }
28      return res;
29    }
30  };
31
32  matrix power(matrix a, ll b){
33    matrix res(a.n, 1);
34    for (; b; a = a * a, b >>= 1){
35      if (b & 1) res = res * a;
36    }
37    return res;
38  }
```

Extended Euclidean Algorithm

- $O(\max(\log a, \log b))$
- Finds solution (x, y) to $ax + by = \gcd(a, b)$
- Can find all solutions given $(x_0, y_0) : \forall k, a(x_0 + kb/g) + b(y_0 - ka/g) = \gcd(a, b)$.

```cpp
1  ll euclid(ll a, ll b, ll &x, ll &y) {
2    if (!b) return x = 1, y = 0, a;
3    ll d = euclid(b, a % b, y, x);
4    return y -= a/b * x, d;
5  }
```

CRT

- $crt(a, m, b, n)$ computes x such that $x \equiv a \pmod{m}, x \equiv b \pmod{n}$
- If $|a| < m$ and $|b| < n$, x will obey $0 \le x < \text{lcm}(m, n)$.
- Assumes $mn < 2^{62}$.
- $O(\max(\log m, \log n))$

```cpp
1  ll crt(ll a, ll m, ll b, ll n) {
2    if (n > m) swap(a, b), swap(m, n);
3    ll x, y, g = euclid(m, n, x, y);
4    assert((a - b) % g == 0); // else no solution
5    // can replace assert with whatever needed
6    x = (b - a) % n * x % n / g * m + a;
7    return x < 0 ? x + m*n/g : x;
8  }
```

Linear Sieve

- Mobius Function

```cpp
1  vi prime;
2  bool is_composite[MAX_N];
3  int mu[MAX_N];
4
5  void sieve(int n){
6    fill(is_composite, is_composite + n, 0);
7    mu[1] = 1;
8    for (int i = 2; i < n; i++){
9      if (!is_composite[i]){
10       prime.push_back(i);
11       mu[i] = -1; //i is prime
12     }
13     for (int j = 0; j < prime.size() && i * prime[j] < n; j++){
14       is_composite[i * prime[j]] = true;
15       if (i % prime[j] == 0){
16         mu[i * prime[j]] = 0; //prime[j] divides i
17         break;
```

```cpp
18       } else {
19         mu[i * prime[j]] = -mu[i]; //prime[j] does not divide i
20       }
21     }
22   }
23 }
```

- Euler's Totient Function

```cpp
1  vi prime;
2  bool is_composite[MAX_N];
3  int phi[MAX_N];
4
5  void sieve(int n){
6    fill(is_composite, is_composite + n, 0);
7    phi[1] = 1;
8    for (int i = 2; i < n; i++){
9      if (!is_composite[i]){
10       prime.push_back (i);
11       phi[i] = i - 1; //i is prime
12     }
13     for (int j = 0; j < prime.size () && i * prime[j] < n; j++){
14       is_composite[i * prime[j]] = true;
15       if (i % prime[j] == 0){
16         phi[i * prime[j]] = phi[i] * prime[j]; //prime[j]
↪ divides i
17         break;
18       } else {
19         phi[i * prime[j]] = phi[i] * phi[prime[j]]; //prime[j]
↪ does not divide i
20       }
21     }
22   }
23 }
```

Mod Class

- For Gaussian Elimination

```cpp
1  constexpr ll norm(ll x) { return (x % MOD + MOD) % MOD; }
2  template <typename T>
3  constexpr T power(T a, ll b, T res = 1) {
4    for (; b; b /= 2, (a *= a) %= MOD)
5      if (b & 1) (res *= a) %= MOD;
6    return res;
7  }
8  struct Z {
9    ll x;
10   constexpr Z(ll _x = 0) : x(norm(_x)) {}
11   // auto operator<=>(const Z &) const = default; // cpp20
↪ only
12   Z operator-() const { return Z(norm(MOD - x)); }
13   Z inv() const { return power(*this, MOD - 2); }
14   Z &operator*=(const Z &rhs) { return x = x * rhs.x % MOD,
↪ *this; }
15   Z &operator+=(const Z &rhs) { return x = norm(x + rhs.x),
↪ *this; }
16   Z &operator-=(const Z &rhs) { return x = norm(x - rhs.x),
↪ *this; }
17   Z &operator/=(const Z &rhs) { return *this *= rhs.inv(); }
18   Z &operator%=(const ll &rhs) { return x %= rhs, *this; }
19   friend Z operator*(Z lhs, const Z &rhs) { return lhs *= rhs;
↪ }
20   friend Z operator+(Z lhs, const Z &rhs) { return lhs += rhs;
↪ }
21   friend Z operator-(Z lhs, const Z &rhs) { return lhs -= rhs;
↪ }
22   friend Z operator/(Z lhs, const Z &rhs) { return lhs /= rhs;
↪ }
23   friend Z operator%(Z lhs, const ll &rhs) { return lhs %=
↪ rhs; }
24   friend auto &operator>>(istream &i, Z &z) { return i >> z.x;
↪ }
25   friend auto &operator<<(ostream &o, const Z &z) { return o
↪ << z.x; }
26 };
```

- Fastest mod class! be careful with overflow, only use when the time limit is tight

```cpp
constexpr int norm(int x) {
  if (x < 0) x += MOD;
  if (x >= MOD) x -= MOD;
  return x;
}
```

Gaussian Elimination

```cpp
bool is_0(Z v) { return v.x == 0; }
int abs(Z v) { return v.x; }
bool is_0(double v) { return abs(v) < 1e-9; }

// 1 => unique solution, 0 => no solution, -1 => multiple
↪  solutions
template <typename T>
int gaussian_elimination(vector<vector<T>> &a, int limit) {
  if (a.empty() || a[0].empty()) return -1;
  int h = (int)a.size(), w = (int)a[0].size(), r = 0;
  for (int c = 0; c < limit; c++) {
    int id = -1;
    for (int i = r; i < h; i++) {
      if (!is_0(a[i][c]) && (id == -1 || abs(a[id][c]) <
↪  abs(a[i][c]))) {
        id = i;
      }
    }
    if (id == -1) continue;
    if (id > r) {
      swap(a[r], a[id]);
      for (int j = c; j < w; j++) a[id][j] = -a[id][j];
    }
    vi nonzero;
    for (int j = c; j < w; j++) {
      if (!is_0(a[r][j])) nonzero.push_back(j);
    }
    T inv_a = 1 / a[r][c];
    for (int i = r + 1; i < h; i++) {
      if (is_0(a[i][c])) continue;
      T coeff = -a[i][c] * inv_a;
      for (int j : nonzero) a[i][j] += coeff * a[r][j];
    }
    ++r;
  }
  for (int row = h - 1; row >= 0; row--) {
    for (int c = 0; c < limit; c++) {
      if (!is_0(a[row][c])) {
        T inv_a = 1 / a[row][c];
        for (int i = row - 1; i >= 0; i--) {
          if (is_0(a[i][c])) continue;
          T coeff = -a[i][c] * inv_a;
          for (int j = c; j < w; j++) a[i][j] += coeff *
↪  a[row][j];
        }
        break;
      }
    }
  } // not-free variables: only it on its line
  for(int i = r; i < h; i++) if(!is_0(a[i][limit])) return 0;
  return (r == limit) ? 1 : -1;
}

template <typename T>
pair<int,vector<T>> solve_linear(vector<vector<T>> a, const
↪  vector<T> &b, int w) {
  int h = (int)a.size();
  forn(i, h) a[i].push_back(b[i]);
  int sol = gaussian_elimination(a, w);
  if(!sol) return {0, vector<T>()};
  vector<T> x(w, 0);
  forn(i, h) {
    forn(j, w) {
      if (!is_0(a[i][j])) {
        x[j] = a[i][w] / a[i][j];
        break;
```

```cpp
      }
    }
  }
  return {sol, x};
}
```

Pollard-Rho Factorization

- Uses Miller–Rabin primality test
- $O(n^{1/4})$ (heuristic estimation)

```cpp
typedef __int128_t i128;

i128 power(i128 a, i128 b, i128 MOD = 1, i128 res = 1) {
  for (; b; b /= 2, (a *= a) %= MOD)
    if (b & 1) (res *= a) %= MOD;
  return res;
}

bool is_prime(ll n) {
  if (n < 2) return false;
  static constexpr int A[] = {2, 3, 5, 7, 11, 13, 17, 19, 23};
  int s = __builtin_ctzll(n - 1);
  ll d = (n - 1) >> s;
  for (auto a : A) {
    if (a == n) return true;
    ll x = (ll)power(a, d, n);
    if (x == 1 || x == n - 1) continue;
    bool ok = false;
    for (int i = 0; i < s - 1; ++i) {
      x = ll((i128)x * x % n);  // potential overflow!
      if (x == n - 1) {
        ok = true;
        break;
      }
    }
    if (!ok) return false;
  }
  return true;
}

ll pollard_rho(ll x) {
  ll s = 0, t = 0, c = rng() % (x - 1) + 1;
  ll stp = 0, goal = 1, val = 1;
  for (goal = 1;; goal *= 2, s = t, val = 1) {
    for (stp = 1; stp <= goal; ++stp) {
      t = ll(((i128)t * t + c) % x);
      val = ll((i128)val * abs(t - s) % x);
      if ((stp % 127) == 0) {
        ll d = gcd(val, x);
        if (d > 1) return d;
      }
    }
    ll d = gcd(val, x);
    if (d > 1) return d;
  }
}

ll get_max_factor(ll _x) {
  ll max_factor = 0;
  function<void(ll)> fac = [&](ll x) {
    if (x <= max_factor || x < 2) return;
    if (is_prime(x)) {
      max_factor = max_factor > x ? max_factor : x;
      return;
    }
    ll p = x;
    while (p >= x) p = pollard_rho(x);
    while ((x % p) == 0) x /= p;
    fac(x), fac(p);
  };
  fac(_x);
  return max_factor;
}
```

Modular Square Root

- $O(\log^2 p)$ in worst case, typically $O(\log p)$ for most p

```
1  ll sqrt(ll a, ll p) {
2    a %= p; if (a < 0) a += p;
3    if (a == 0) return 0;
4    assert(pow(a, (p-1)/2, p) == 1); // else no solution
5    if (p % 4 == 3) return pow(a, (p+1)/4, p);
6    // a^(n+3)/8 or 2^(n+3)/8 * 2^(n-1)/4 works if p % 8 == 5
7    ll s = p - 1, n = 2;
8    int r = 0, m;
9    while (s % 2 == 0)
10     ++r, s /= 2;
11   /// find a non-square mod p
12   while (pow(n, (p - 1) / 2, p) != p - 1) ++n;
13   ll x = pow(a, (s + 1) / 2, p);
14   ll b = pow(a, s, p), g = pow(n, s, p);
15   for (;; r = m) {
16     ll t = b;
17     for (m = 0; m < r && t != 1; ++m)
18       t = t * t % p;
19     if (m == 0) return x;
20     ll gs = pow(g, 1LL << (r - m - 1), p);
21     g = gs * gs % p;
22     x = x * gs % p;
23     b = b * g % p;
24   }
25 }
```

Berlekamp-Massey

- Recovers any n-order linear recurrence relation from the first $2n$ terms of the sequence.
- Input s is the sequence to be analyzed.
- Output c is the shortest sequence $c_1, ..., c_n$, such that

$$s_m = \sum_{i=1}^{n} c_i \cdot s_{m-i}, \text{ for all } m \geq n.$$

- Be careful since c is returned in 0-based indexation.
- Complexity: $O(N^2)$

```
1  vll berlekamp_massey(vll s) {
2    int n = sz(s), l = 0, m = 1;
3    vll b(n), c(n);
4    ll ldd = b[0] = c[0] = 1;
5    for (int i = 0; i < n; i++, m++) {
6      ll d = s[i];
7      for (int j = 1; j <= l; j++) d = (d + c[j] * s[i - j]) % MOD;
8      if (d == 0) continue;
9      vll temp = c;
10     ll coef = d * power(ldd, MOD - 2) % MOD;
11     for (int j = m; j < n; j++){
12       c[j] = (c[j] + MOD - coef * b[j - m]) % MOD;
13       if (c[j] < 0) c[j] += MOD;
14     }
15     if (2 * l <= i) {
16       l = i + 1 - l;
17       b = temp;
18       ldd = d;
19       m = 0;
20     }
21   }
22   c.resize(l + 1);
23   c.erase(c.begin());
24   for (ll &x : c)
25     x = (MOD - x) % MOD;
26   return c;
27 }
```

Calculating k-th term of a linear recurrence

- Given the first n terms $s_0, s_1, ..., s_{n-1}$ and the sequence $c_1, c_2, ..., c_n$ such that

$$s_m = \sum_{i=1}^{n} c_i \cdot s_{m-i}, \text{ for all } m \geq n,$$

the function calc_kth computes s_k.
- Complexity: $O(n^2 \log k)$

```
1  vll poly_mult_mod(vll p, vll q, vll& c){
2    vll ans(sz(p) + sz(q) - 1);
3    forn(i, sz(p)){
4      forn(j, sz(q)){
5        ans[i + j] = (ans[i + j] + p[i] * q[j]) % MOD;
6      }
7    }
8    int n = sz(ans), m = sz(c);
9    for (int i = n - 1; i >= m; i--){
10     forn(j, m){
11       ans[i - 1 - j] = (ans[i - 1 - j] + c[j] * ans[i]) % MOD;
12     }
13   }
14   ans.resize(m);
15   return ans;
16 }
17
18 ll calc_kth(vll s, vll c, ll k){
19   assert(sz(s) >= sz(c)); // size of s can be greater than c,
   ↪ but not less
20   if (k < sz(s)) return s[k];
21   vll res{1};
22   for (vll poly = {0, 1}; k; poly = poly_mult_mod(poly, poly,
   ↪ c), k >>= 1){
23     if (k & 1) res = poly_mult_mod(res, poly, c);
24   }
25   ll ans = 0;
26   forn(i, min(sz(res), sz(c))) ans = (ans + s[i] * res[i]) %
   ↪ MOD;
27   return ans;
28 }
```

Partition Function

- Returns number of partitions of n in $O(n^{1.5})$

```
1  int partition(int n) {
2    int dp[n + 1];
3    dp[0] = 1;
4    for (int i = 1; i <= n; i++) {
5      dp[i] = 0;
6      for (int j = 1, r = 1; i - (3 * j * j - j) / 2 >= 0; ++j,
   ↪ r *= -1) {
7        dp[i] += dp[i - (3 * j * j - j) / 2] * r;
8        if (i - (3 * j * j + j) / 2 >= 0) dp[i] += dp[i - (3 * j
   ↪ * j + j) / 2] * r;
9      }
10   }
11   return dp[n];
12 }
```

NTT

- large mod (for NTT to do FFT in ll range without modulo)

```
1  constexpr i128 MOD = 9223372036737335297;
```

- Otherwise, use below

```
1  const int MOD = 998244353;
2  void ntt(vll& a, int f) {
3    int n = int(a.size());
4    vll w(n);
```

```
5     vi rev(n);
6     forn(i, n) rev[i] = (rev[i / 2] / 2) | ((i & 1) * (n / 2));
7     forn(i, n) {
8       if (i < rev[i]) swap(a[i], a[rev[i]]);
9     }
10    ll wn = power(f ? (MOD + 1) / 3 : 3, (MOD - 1) / n);
11    w[0] = 1;
12    for (int i = 1; i < n; i++) w[i] = w[i - 1] * wn % MOD;
13    for (int mid = 1; mid < n; mid *= 2) {
14      for (int i = 0; i < n; i += 2 * mid) {
15        forn(j, mid) {
16          ll x = a[i + j], y = a[i + j + mid] * w[n / (2 * mid)
      ↪ * j] % MOD;
17          a[i + j] = (x + y) % MOD, a[i + j + mid] = (x + MOD -
      ↪ y) % MOD;
18        }
19      }
20    }
21    if (f) {
22      ll iv = power(n, MOD - 2);
23      for (auto& x : a) x = x * iv % MOD;
24    }
25  }
26  vll mul(vll a, vll b) {
27    int n = 1, m = (int)a.size() + (int)b.size() - 1;
28    while (n < m) n *= 2;
29    a.resize(n), b.resize(n);
30    ntt(a, 0), ntt(b, 0); // if squaring, you can save one NTT
      ↪   here
31    forn(i, n) a[i] = a[i] * b[i] % MOD;
32    ntt(a, 1);
33    a.resize(m);
34    return a;
35  }
```

FFT

```
1   const ld PI = acosl(-1);
2   auto mul = [&](const vector<ld>& aa, const vector<ld>& bb) {
3     int n = (int)aa.size(), m = (int)bb.size(), bit = 1;
4     while ((1 << bit) < n + m - 1) bit++;
5     int len = 1 << bit;
6     vector<complex<ld>> a(len), b(len);
7     vi rev(len);
8     forn(i, n) a[i].real(aa[i]);
9     forn(i, m) b[i].real(bb[i]);
10    forn(i, len) rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (bit
      ↪  - 1));
11    auto fft = [&](vector<complex<ld>>& p, int inv) {
12      forn(i, len)
13        if (i < rev[i]) swap(p[i], p[rev[i]]);
14      for (int mid = 1; mid < len; mid *= 2) {
15        auto w1 = complex<ld>(cos(PI / mid), (inv ? -1 : 1) *
      ↪ sin(PI / mid));
16        for (int i = 0; i < len; i += mid * 2) {
17          auto wk = complex<ld>(1, 0);
18          for (int j = 0; j < mid; j++, wk = wk * w1) {
19            auto x = p[i + j], y = wk * p[i + j + mid];
20            p[i + j] = x + y, p[i + j + mid] = x - y;
21          }
22        }
23      }
24      if (inv == 1) {
25        forn(i, len) p[i].real(p[i].real() / len);
26      }
27    };
28    fft(a, 0), fft(b, 0);
29    forn(i, len) a[i] = a[i] * b[i];
30    fft(a, 1);
31    a.resize(n + m - 1);
32    vector<ld> res(n + m - 1);
33    forn(i, n + m - 1) res[i] = a[i].real();
34    return res;
35  };
```

Poly mod, log, exp, multipoint, interpolation

- $\frac{1}{P(x)}$ in $O(n \log n)$, $e^{P(x)}$ in $O(n \log n)$, $\ln(P(x))$ in $O(n \log n)$, $P(x)^k$ in $O(n \log n)$, Evaluates $P(x_1), \cdots, P(x_n)$ in $O(n \log^2 n)$, Lagrange Interpolation in $O(n \log^2 n)$

```
1   // Examples:
2   // poly a(n+1); // constructs degree n poly
3   // a[0].v = 10; // assigns constant term a_0 = 10
4   // poly b = exp(a);
5   // poly is vector<num>
6   // for NTT, num stores just one int named v
7
8   #define sz(x) ((int)x.size())
9   #define rep(i, j, k) for (int i = int(j); i < int(k); i++)
10  #define per(i, a, b) for (int i = (b)-1; i >= (a); --i)
11  using vi = vi;
12
13  const int MOD = 998244353, g = 3;
14
15  // NTT
16  // For p < 2^30 there is also (5 << 25, 3), (7 << 26, 3),
17  // (479 << 21, 3) and (483 << 21, 5). Last two are > 10^9.
18  struct num {
19    int v;
20    num(ll v_ = 0): v(int(v_ % MOD)) {
21      if (v < 0) v += MOD;
22    }
23    explicit operator int() const { return v; }
24  };
25  inline num operator+(num a, num b) { return num(a.v + b.v); }
26  inline num operator-(num a, num b) { return num(a.v + MOD -
      ↪ b.v); }
27  inline num operator*(num a, num b) { return num(1ll * a.v *
      ↪ b.v); }
28  inline num pow(num a, int b) {
29    num r = 1;
30    do {
31      if (b & 1) r = r * a;
32      a = a * a;
33    } while (b >>= 1);
34    return r;
35  }
36  inline num inv(num a) { return pow(a, MOD - 2); }
37  using vn = vector<num>;
38  vi rev({0, 1});
39  vn rt(2, num(1)), fa, fb;
40  inline void init(int n) {
41    if (n <= sz(rt)) return;
42    rev.resize(n);
43    rep(i, 0, n) rev[i] = (rev[i >> 1] | ((i & 1) * n)) >> 1;
44    rt.reserve(n);
45    for (int k = sz(rt); k < n; k *= 2) {
46      rt.resize(2 * k);
47      num z = pow(num(g), (MOD - 1) / (2 * k)); // NTT
48      rep(i, k / 2, k) rt[2 * i] = rt[i], rt[2 * i + 1] = rt[i]
      ↪ * z;
49    }
50  }
51  inline void fft(vector<num>& a, int n) {
52    init(n);
53    int s = __builtin_ctz(sz(rev) / n);
54    rep(i, 0, n) if (i < rev[i] >> s) swap(a[i], a[rev[i] >>
      ↪ s]);
55    for (int k = 1; k < n; k *= 2)
56      for (int i = 0; i < n; i += 2 * k) rep(j, 0, k) {
57        num t = rt[j + k] * a[i + j + k];
58        a[i + j + k] = a[i + j] - t;
59        a[i + j] = a[i + j] + t;
60      }
61  }
62  // NTT
63  vn multiply(vn a, vn b) {
64    int s = sz(a) + sz(b) - 1;
65    if (s <= 0) return {};
```

```
 66     int L = s > 1 ? 32 - __builtin_clz(s - 1) : 0, n = 1 << L;
 67     a.resize(n), b.resize(n);
 68     fft(a, n);
 69     fft(b, n);
 70     num d = inv(num(n));
 71     rep(i, 0, n) a[i] = a[i] * b[i] * d;
 72     reverse(a.begin() + 1, a.end());
 73     fft(a, n);
 74     a.resize(s);
 75     return a;
 76   }
 77   // NTT power-series inverse
 78   // Doubles b as b[:n] = (2 - a[:n] * b[:n/2]) * b[:n/2]
 79   vn inverse(const vn& a) {
 80     if (a.empty()) return {};
 81     vn b({inv(a[0])});
 82     b.reserve(2 * a.size());
 83     while (sz(b) < sz(a)) {
 84       int n = 2 * sz(b);
 85       b.resize(2 * n, 0);
 86       if (sz(fa) < 2 * n) fa.resize(2 * n);
 87       fill(fa.begin(), fa.begin() + 2 * n, 0);
 88       copy(a.begin(), a.begin() + min(n, sz(a)), fa.begin());
 89       fft(b, 2 * n);
 90       fft(fa, 2 * n);
 91       num d = inv(num(2 * n));
 92       rep(i, 0, 2 * n) b[i] = b[i] * (2 - fa[i] * b[i]) * d;
 93       reverse(b.begin() + 1, b.end());
 94       fft(b, 2 * n);
 95       b.resize(n);
 96     }
 97     b.resize(a.size());
 98     return b;
 99   }
100
101   using poly = vn;
102
103   poly operator+(const poly& a, const poly& b) {
104     poly r = a;
105     if (sz(r) < sz(b)) r.resize(b.size());
106     rep(i, 0, sz(b)) r[i] = r[i] + b[i];
107     return r;
108   }
109   poly operator-(const poly& a, const poly& b) {
110     poly r = a;
111     if (sz(r) < sz(b)) r.resize(b.size());
112     rep(i, 0, sz(b)) r[i] = r[i] - b[i];
113     return r;
114   }
115   poly operator*(const poly& a, const poly& b) {
116     return multiply(a, b);
117   }
118   // Polynomial floor division; no leading 0's please
119   poly operator/(poly a, poly b) {
120     if (sz(a) < sz(b)) return {};
121     int s = sz(a) - sz(b) + 1;
122     reverse(a.begin(), a.end());
123     reverse(b.begin(), b.end());
124     a.resize(s);
125     b.resize(s);
126     a = a * inverse(move(b));
127     a.resize(s);
128     reverse(a.begin(), a.end());
129     return a;
130   }
131   poly operator%(const poly& a, const poly& b) {
132     poly r = a;
133     if (sz(r) >= sz(b)) {
134       poly c = (r / b) * b;
135       r.resize(sz(b) - 1);
136       rep(i, 0, sz(r)) r[i] = r[i] - c[i];
137     }
138     return r;
139   }
140
141   // Log/exp/pow
142   poly deriv(const poly& a) {
```

```
143     if (a.empty()) return {};
144     poly b(sz(a) - 1);
145     rep(i, 1, sz(a)) b[i - 1] = a[i] * i;
146     return b;
147   }
148   poly integ(const poly& a) {
149     poly b(sz(a) + 1);
150     b[1] = 1; // mod p
151     rep(i, 2, sz(b)) b[i] =
152       b[MOD % i] * (-MOD / i); // mod p
153     rep(i, 1, sz(b)) b[i] = a[i - 1] * b[i]; // mod p
154     //rep(i,1,sz(b)) b[i]=a[i-1]*inv(num(i)); // else
155     return b;
156   }
157   poly log(const poly& a) { // MUST have a[0] == 1
158     poly b = integ(deriv(a) * inverse(a));
159     b.resize(a.size());
160     return b;
161   }
162   poly exp(const poly& a) { // MUST have a[0] == 0
163     poly b(1, num(1));
164     if (a.empty()) return b;
165     while (sz(b) < sz(a)) {
166       int n = min(sz(b) * 2, sz(a));
167       b.resize(n);
168       poly v = poly(a.begin(), a.begin() + n) - log(b);
169       v[0] = v[0] + num(1);
170       b = b * v;
171       b.resize(n);
172     }
173     return b;
174   }
175   poly pow(const poly& a, int m) { // m >= 0
176     poly b(a.size());
177     if (!m) {
178       b[0] = 1;
179       return b;
180     }
181     int p = 0;
182     while (p < sz(a) && a[p].v == 0) ++p;
183     if (1ll * m * p >= sz(a)) return b;
184     num mu = pow(a[p], m), di = inv(a[p]);
185     poly c(sz(a) - m * p);
186     rep(i, 0, sz(c)) c[i] = a[i + p] * di;
187     c = log(c);
188     for(auto &v : c) v = v * m;
189     c = exp(c);
190     rep(i, 0, sz(c)) b[i + m * p] = c[i] * mu;
191     return b;
192   }
193
194   // Multipoint evaluation/interpolation
195
196   vector<num> eval(const poly& a, const vector<num>& x) {
197     int n = sz(x);
198     if (!n) return {};
199     vector<poly> up(2 * n);
200     rep(i, 0, n) up[i + n] = poly({0 - x[i], 1});
201     per(i, 1, n) up[i] = up[2 * i] * up[2 * i + 1];
202     vector<poly> down(2 * n);
203     down[1] = a % up[1];
204     rep(i, 2, 2 * n) down[i] = down[i / 2] % up[i];
205     vector<num> y(n);
206     rep(i, 0, n) y[i] = down[i + n][0];
207     return y;
208   }
209
210   poly interp(const vector<num>& x, const vector<num>& y) {
211     int n = sz(x);
212     assert(n);
213     vector<poly> up(n * 2);
214     rep(i, 0, n) up[i + n] = poly({0 - x[i], 1});
215     per(i, 1, n) up[i] = up[2 * i] * up[2 * i + 1];
216     vector<num> a = eval(deriv(up[1]), x);
217     vector<poly> down(2 * n);
218     rep(i, 0, n) down[i + n] = poly({y[i] * inv(a[i])});
219     per(i, 1, n) down[i] =
```

```
220        down[i * 2] * up[i * 2 + 1] + down[i * 2 + 1] * up[i * 2];
221    return down[1];
222  }
```

Simplex method for linear programs

- Maximize $c^T x$ subject to $Ax \le b$, $x \ge 0$.
- Returns $-\infty$ if there is no solution, $+\infty$ if there are arbitrarily good solutions, or the maximum value of $c^T x$ otherwise. The (arbitrary) 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.
- Complexity: $O(NM \cdot pivots)$. $O(2^n)$ in general (very hard to achieve).

```
1  typedef double T; // might be much slower with long doubles
2  typedef vector<T> vd;
3  typedef vector<vd> vvd;
4  const T eps = 1e-8, inf = 1/.0;
5  #define MP make_pair
6  #define ltj(X) if(s == -1 || MP(X[j],N[j]) < MP(X[s],N[s]))
   ↪  s=j
7  #define rep(i, a, b) for(int i = a; i < (b); ++i)
8
9  struct LPSolver {
10   int m, n;
11   vi N,B;
12   vvd D;
13   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)){
14     rep(i,0,m) rep(j,0,n) D[i][j] = A[i][j];
15     rep(i,0,m) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i];}
   ↪  rep(j,0,n) { N[j] = j; D[m][j] = -c[j]; }
16     N[n] = -1; D[m+1][n] = 1;
17   };
18   void pivot(int r, int s){
19     T *a = D[r].data(), inv = 1 / a[s];
20     rep(i,0,m+2) if (i != r && abs(D[i][s]) > eps) {
21       T *b = D[i].data(), inv2 = b[s] * inv;
22       rep(j,0,n+2) b[j] -= a[j] * inv2;
23       b[s] = a[s] * inv2;
24     }
25     rep(j,0,n+2) if (j != s) D[r][j] *= inv;
26     rep(i,0,m+2) if (i != r) D[i][s] *= -inv;
27     D[r][s] = inv;
28     swap(B[r], N[s]);
29   }
30   bool simplex(int phase){
31     int x = m + phase - 1;
32     for (;;) {
33       int s = -1;
34       rep(j,0,n+1) if (N[j] != -phase) ltj(D[x]); if (D[x][s]
   ↪  >= -eps) return true;
35       int r = -1;
36       rep(i,0,m) {
37         if (D[i][s] <= eps) continue;
38         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;
39       }
40       if (r == -1) return false;
41       pivot(r, s);
42     }
43   }
44   T solve(vd &x){
45     int r = 0;
46     rep(i,1,m) if (D[i][n+1] < D[r][n+1]) r = i;
47     if (D[r][n+1] < -eps) {
48       pivot(r, n);
49       if (!simplex(2) || D[m+1][n+1] < -eps) return -inf;
50       rep(i,0,m) if (B[i] == -1) {
51         int s = 0;
52         rep(j,1,n+1) ltj(D[i]);
53         pivot(i, s);
54       }
55     }
56     bool ok = simplex(1); x = vd(n);
57     rep(i,0,m) if (B[i] < n) x[B[i]] = D[i][n+1];
58     return ok ? D[m][n+1] : inf;
59   }
60 };
```

Matroid Intersection

- Matroid is a pair $< X, I >$, where X is a finite set and I is a family of subsets of X satisfying:
  1. $\emptyset \in I$.
  2. If $A \in I$ and $B \subseteq A$, then $B \in I$.
  3. If $A, B \in I$ and $|A| > |B|$, then there exists $x \in A \backslash B$ such that $B \cup \{x\} \in I$.
- Set S is called **independent** if $S \in I$.
- **Common matroids:** uniform (sets of bounded size); colorful (sets of colored elements where each color only appears once); graphic (acyclic sets of edges in a graph); linear-algebraic (sets of linearly independent vectors).
- **Matroid Intersection Problem:** Given two matroids, find the largest common independent set.
- A matroid has 3 functions:
  - *check(int x)*: returns if current matroid can add x without becoming dependent.
  - *add(int x)*: adds an element to the matroid (guaranteed to never make it dependent).
  - *clear()*: sets the matroid to the empty matroid.
- The matroid is given an *int* representing the element, and is expected to convert it (e.g: color or edge endpoints)
- Pass the matroid with more expensive add/clear operations to M1.
- **Complexity:** $R^2 \cdot N \cdot (M2.add + M1.check + M2.check) + R^3 \cdot (M1.add) + R^2 \cdot (M1.clear) + R \cdot N \cdot (M2.clear)$, where $R = answer$.

```
1
2  // Example matroid
3  struct GraphicMatroid{
4    vector<pair<int, int>> e;
5    int n;
6    DSU dsu;
7
8    GraphicMatroid(vector<pair<int, int>> edges, int vertices){
9      e = edges, n = vertices;
10     dsu = DSU(n);
11   };
12   bool check(int idx){
13     return !dsu.same(e[idx].fi, e[idx].se);
14   }
15   void add(int idx){
16     dsu.unite(e[idx].fi, e[idx].se);
17   }
18   void clear(){
19     dsu = DSU(n);
20   }
21 };
22
23 template <class M1, class M2> struct MatroidIsect {
24   int n;
25   vector<char> iset;
26   M1 m1; M2 m2;
27   MatroidIsect(M1 m1, M2 m2, int n) : n(n), iset(n + 1),
   ↪  m1(m1), m2(m2) {}
28   vi solve() {
29     forn(i, n) if (m1.check(i) && m2.check(i))
30       iset[i] = true, m1.add(i), m2.add(i);
31     while (augment());
```

```
32          vi ans;
33          forn(i, n) if (iset[i]) ans.push_back(i);
34          return ans;
35      }
36      bool augment() {
37          vi frm(n, -1);
38          queue<int> q({n}); // starts at dummy node
39          auto fwdE = [&](int a) {
40              vi ans;
41              m1.clear();
42              for (int v = 0; v < n; v++) if (iset[v] && v != a)
  ↪ m1.add(v);
43              for (int b = 0; b < n; b++) if (!iset[b] && frm[b]
  ↪ == -1 && m1.check(b))
44                  ans.push_back(b), frm[b] = a;
45              return ans;
46          };
47          auto backE = [&](int b) {
48              m2.clear();
49              for (int cas = 0; cas < 2; cas++) for (int v = 0;
  ↪ v < n; v++){
50                  if ((v == b || iset[v]) && (frm[v] == -1) ==
  ↪ cas) {
51                      if (!m2.check(v))
52                          return cas ? q.push(v), frm[v] = b, v
  ↪ : -1;
53                      m2.add(v);
54                  }
55              }
56              return n;
57          };
58          while (!q.empty()) {
59              int a = q.front(), c; q.pop();
60              for (int b : fwdE(a))
61                  while((c = backE(b)) >= 0) if (c == n) {
62                      while (b != n) iset[b] ^= 1, b = frm[b];
63                      return true;
64                  }
65          }
66          return false;
67      }
68  };
69
70  /*
71  Usage:
72  MatroidIsect<GraphicMatroid, ColorfulMatroid> solver(matroid1,
  ↪  matroid2, n);
73  vi answer = solver.solve();
74  */
```

Data Structures

Fenwick Tree

```
1  ll sum(int r) {
2    ll ret = 0;
3    for (; r >= 0; r = (r & r + 1) - 1) ret += bit[r];
4    return ret;
5  }
6  void add(int idx, ll delta) {
7    for (; idx < n; idx |= idx + 1) bit[idx] += delta;
8  }
```

Lazy Propagation SegTree

```
1  // Clear: clear() or build()
2  const int N = 2e5 + 10; // Change the constant!
3  template<typename T>
4  struct LazySegTree{
5    T t[4 * N];
6    T lazy[4 * N];
7    int n;
8
9    // Change these functions, default return, and lazy mark.
10   T default_return = 0, lazy_mark = numeric_limits<T>::min();
```

```
11   // Lazy mark is how the algorithm will identify that no
  ↪  propagation is needed.
12   function<T(T, T)> f = [&] (T a, T b){
13     return a + b;
14   };
15   // f_on_seg calculates the function f, knowing the lazy
  ↪  value on segment,
16   // segment's size and the previous value.
17   // The default is segment modification for RSQ. For
  ↪  increments change to:
18   //     return cur_seg_val + seg_size * lazy_val;
19   // For RMQ.  Modification: return lazy_val;  Increments:
  ↪  return cur_seg_val + lazy_val;
20   function<T(T, int, T)> f_on_seg = [&] (T cur_seg_val, int
  ↪  seg_size, T lazy_val){
21     return seg_size * lazy_val;
22   };
23   // upd_lazy updates the value to be propagated to child
  ↪  segments.
24   // Default: modification. For increments change to:
25   //     lazy[v] = (lazy[v] == lazy_mark? val : lazy[v] +
  ↪  val);
26   function<void(int, T)> upd_lazy = [&] (int v, T val){
27     lazy[v] = val;
28   };
29   // Tip: for "get element on single index" queries, use max()
  ↪  on segment: no overflows.
30
31   LazySegTree(int n_) : n(n_) {
32     clear(n);
33   }
34
35   void build(int v, int tl, int tr, vector<T>& a){
36     if (tl == tr) {
37       t[v] = a[tl];
38       return;
39     }
40     int tm = (tl + tr) / 2;
41     // left child: [tl, tm]
42     // right child: [tm + 1, tr]
43     build(2 * v + 1, tl, tm, a);
44     build(2 * v + 2, tm + 1, tr, a);
45     t[v] = f(t[2 * v + 1], t[2 * v + 2]);
46   }
47
48   LazySegTree(vector<T>& a){
49     build(a);
50   }
51
52   void push(int v, int tl, int tr){
53     if (lazy[v] == lazy_mark) return;
54     int tm = (tl + tr) / 2;
55     t[2 * v + 1] = f_on_seg(t[2 * v + 1], tm - tl + 1,
  ↪  lazy[v]);
56     t[2 * v + 2] = f_on_seg(t[2 * v + 2], tr - tm, lazy[v]);
57     upd_lazy(2 * v + 1, lazy[v]), upd_lazy(2 * v + 2,
  ↪  lazy[v]);
58     lazy[v] = lazy_mark;
59   }
60
61   void modify(int v, int tl, int tr, int l, int r, T val){
62     if (l > r) return;
63     if (tl == l && tr == r){
64       t[v] = f_on_seg(t[v], tr - tl + 1, val);
65       upd_lazy(v, val);
66       return;
67     }
68     push(v, tl, tr);
69     int tm = (tl + tr) / 2;
70     modify(2 * v + 1, tl, tm, l, min(r, tm), val);
71     modify(2 * v + 2, tm + 1, tr, max(l, tm + 1), r, val);
72     t[v] = f(t[2 * v + 1], t[2 * v + 2]);
73   }
74
75   T query(int v, int tl, int tr, int l, int r) {
76     if (l > r) return default_return;
77     if (tl == l && tr == r) return t[v];
```

```
78        push(v, tl, tr);
79        int tm = (tl + tr) / 2;
80        return f(
81          query(2 * v + 1, tl, tm, l, min(r, tm)),
82          query(2 * v + 2, tm + 1, tr, max(l, tm + 1), r)
83        );
84      }
85
86      void modify(int l, int r, T val){
87        modify(0, 0, n - 1, l, r, val);
88      }
89
90      T query(int l, int r){
91        return query(0, 0, n - 1, l, r);
92      }
93
94      T get(int pos){
95        return query(pos, pos);
96      }
97
98      // Change clear() function to t.clear() if using
   ↪  unordered_map for SegTree!!!
99      void clear(int n_){
100       n = n_;
101       forn(i, 4 * n) t[i] = 0, lazy[i] = lazy_mark;
102     }
103
104     void build(vector<T>& a){
105       n = sz(a);
106       clear(n);
107       build(0, 0, n - 1, a);
108     }
109   };
```

Sparse Table

```
1   const int N = 2e5 + 10, LOG = 20; // Change the constant!
2   template<typename T>
3   struct SparseTable{
4   int lg[N];
5   T st[N][LOG];
6   int n;
7
8   // Change this function
9   function<T(T, T)> f = [&] (T a, T b){
10    return min(a, b);
11  };
12
13  void build(vector<T>& a){
14    n = sz(a);
15    lg[1] = 0;
16    for (int i = 2; i <= n; i++) lg[i] = lg[i / 2] + 1;
17
18    for (int k = 0; k < LOG; k++){
19      forn(i, n){
20        if (!k) st[i][k] = a[i];
21        else st[i][k] = f(st[i][k - 1], st[min(n - 1, i + (1 <<
   ↪  (k - 1)))][k - 1]);
22      }
23    }
24  }
25
26  T query(int l, int r){
27    int sz = r - l + 1;
28    return f(st[l][lg[sz]], st[r - (1 << lg[sz]) + 1][lg[sz]]);
29  }
30  };
```

Suffix Array and LCP array

- (uses SparseTable above)

```
1   struct SuffixArray{
2     vi p, c, h;
3     SparseTable<int> st;
4     /*
```

```
5    In the end, array c gives the position of each suffix in p
6    using 1-based indexation!
7    */
8
9    SuffixArray() {}
10
11   SuffixArray(string s){
12     buildArray(s);
13     buildLCP(s);
14     buildSparse();
15   }
16
17   void buildArray(string s){
18     int n = sz(s) + 1;
19     p.resize(n), c.resize(n);
20     forn(i, n) p[i] = i;
21     sort(all(p), [&] (int a, int b){return s[a] < s[b];});
22     c[p[0]] = 0;
23     for (int i = 1; i < n; i++){
24       c[p[i]] = c[p[i - 1]] + (s[p[i]] != s[p[i - 1]]);
25     }
26     vi p2(n), c2(n);
27     // w is half-length of each string.
28     for (int w = 1; w < n; w <<= 1){
29       forn(i, n){
30         p2[i] = (p[i] - w + n) % n;
31       }
32       vi cnt(n);
33       for (auto i : c) cnt[i]++;
34       for (int i = 1; i < n; i++) cnt[i] += cnt[i - 1];
35       for (int i = n - 1; i >= 0; i--){
36         p[--cnt[c[p2[i]]]] = p2[i];
37       }
38       c2[p[0]] = 0;
39       for (int i = 1; i < n; i++){
40         c2[p[i]] = c2[p[i - 1]] +
41         (c[p[i]] != c[p[i - 1]] ||
42         c[(p[i] + w) % n] != c[(p[i - 1] + w) % n]);
43       }
44       c.swap(c2);
45     }
46     p.erase(p.begin());
47   }
48
49   void buildLCP(string s){
50     // The algorithm assumes that suffix array is already
   ↪  built on the same string.
51     int n = sz(s);
52     h.resize(n - 1);
53     int k = 0;
54     forn(i, n){
55       if (c[i] == n){
56         k = 0;
57         continue;
58       }
59       int j = p[c[i]];
60       while (i + k < n && j + k < n && s[i + k] == s[j + k])
   ↪  k++;
61       h[c[i] - 1] = k;
62       if (k) k--;
63     }
64     /*
65     Then an RMQ Sparse Table can be built on array h
66     to calculate LCP of 2 non-consecutive suffixes.
67     */
68   }
69
70   void buildSparse(){
71     st.build(h);
72   }
73
74   // l and r must be in 0-BASED INDEXATION
75   int lcp(int l, int r){
76     l = c[l] - 1, r = c[r] - 1;
77     if (l > r) swap(l, r);
78     return st.query(l, r - 1);
79   }
```

```
80      };
```

Aho Corasick Trie

- For each node in the trie, the suffix link points to the longest proper suffix of the represented string. The terminal-link tree has square-root height (can be constructed by DFS).

```
1   const int S = 26;
2
3   // Function converting char to int.
4   int ctoi(char c){
5     return c - 'a';
6   }
7
8   // To add terminal links, use DFS
9   struct Node{
10    vi nxt;
11    int link;
12    bool terminal;
13
14    Node() {
15      nxt.assign(S, -1), link = 0, terminal = 0;
16    }
17  };
18
19  vector<Node> trie(1);
20
21  // add_string returns the terminal vertex.
22  int add_string(string& s){
23    int v = 0;
24    for (auto c : s){
25      int cur = ctoi(c);
26      if (trie[v].nxt[cur] == -1){
27        trie[v].nxt[cur] = sz(trie);
28        trie.emplace_back();
29      }
30      v = trie[v].nxt[cur];
31    }
32    trie[v].terminal = 1;
33    return v;
34  }
35
36  /*
37  Suffix links are compressed.
38  This means that:
39    If vertex v has a child by letter x, then:
40      trie[v].nxt[x] points to that child.
41    If vertex v doesn't have such child, then:
42      trie[v].nxt[x] points to the suffix link of that child
43      if we would actually have it.
44  */
45  void add_links(){
46    queue<int> q;
47    q.push(0);
48    while (!q.empty()){
49      auto v = q.front();
50      int u = trie[v].link;
51      q.pop();
52      forn(i, S){
53        int& ch = trie[v].nxt[i];
54        if (ch == -1){
55          ch = v? trie[u].nxt[i] : 0;
56        }
57        else{
58          trie[ch].link = v? trie[u].nxt[i] : 0;
59          q.push(ch);
60        }
61      }
62    }
63  }
64
65  bool is_terminal(int v){
66    return trie[v].terminal;
67  }
```

```
68
69  int get_link(int v){
70    return trie[v].link;
71  }
72
73  int go(int v, char c){
74    return trie[v].nxt[ctoi(c)];
75  }
```

Convex Hull Trick

- Allows to insert a linear function to the hull in (1) and get the minimum/maximum value of the stored function at a point in O(log n).
- NOTE: The lines must be added in the order of decreasing/increasing gradients. CAREFULLY CHECK THE SETUP BEFORE USING!
- IMPORTANT: DO NOT MODIFY TO QUERY MAX, IT WILL BREAK

```
1   struct line{
2     ll k, b;
3     ll f(ll x){
4       return k * x + b;
5     };
6   };
7
8   vector<line> hull;
9
10  void add_line(line nl){
11    if (!hull.empty() && hull.back().k == nl.k){
12      nl.b = min(nl.b, hull.back().b);
13      hull.pop_back();
14    }
15    while (sz(hull) > 1){
16      auto& l1 = hull.end()[-2], l2 = hull.back();
17      if ((nl.b - l1.b) * (l2.k - nl.k) >= (nl.b - l2.b) * (l1.k
    ↪ - nl.k)) hull.pop_back();
18      else break;
19    }
20    hull.pb(nl);
21  }
22
23  ll get(ll x){
24    int l = 0, r = sz(hull);
25    while (r - l > 1){
26      int mid = (l + r) / 2;
27      if (hull[mid - 1].f(x) >= hull[mid].f(x)) l = mid;
28      else r = mid;
29    }
30    return hull[l].f(x);
31  }
```

Li-Chao Segment Tree

- allows to add linear functions in any order and query minimum/maximum value of those at a point, all in O(log n).
- Clear: clear()

```
1   const ll INF = 1e18; // Change the constant!
2   struct LiChaoTree{
3     struct line{
4       ll k, b;
5       line(){
6         k = b = 0;
7       };
8       line(ll k_, ll b_){
9         k = k_, b = b_;
10      };
11      ll f(ll x){
12        return k * x + b;
13      };
14    };
```

```
15   int n;
16   bool minimum, on_points;
17   vll pts;
18   vector<line> t;
19
20   void clear(){
21     for (auto& l : t) l.k = 0, l.b = minimum? INF : -INF;
22   }
23
24   LiChaoTree(int n_, bool min_){ // This is a default
  ↪  constructor for numbers in range [0, n - 1].
25     n = n_, minimum = min_, on_points = false;
26     t.resize(4 * n);
27     clear();
28   };
29
30   LiChaoTree(vll pts_, bool min_){ // This constructor will
  ↪  build LCT on the set of points you pass. The points may be
  ↪  in any order and contain duplicates.
31     pts = pts_, minimum = min_;
32     sort(all(pts));
33     pts.erase(unique(all(pts)), pts.end());
34     on_points = true;
35     n = sz(pts);
36     t.resize(4 * n);
37     clear();
38   };
39
40   void add_line(int v, int l, int r, line nl){
41     // Adding on segment [l, r]
42     int m = (l + r) / 2;
43     ll lval = on_points? pts[l] : l, mval = on_points? pts[m]
  ↪  : m;
44     if ((minimum && nl.f(mval) < t[v].f(mval)) || (!minimum &&
  ↪  nl.f(mval) > t[v].f(mval))) swap(t[v], nl);
45     if (r - l == 1) return;
46     if ((minimum && nl.f(lval) < t[v].f(lval)) || (!minimum &&
  ↪  nl.f(lval) > t[v].f(lval))) add_line(2 * v + 1, l, m, nl);
47     else add_line(2 * v + 2, m, r, nl);
48   }
49
50   ll get(int v, int l, int r, int x){
51     int m = (l + r) / 2;
52     if (r - l == 1) return t[v].f(on_points? pts[x] : x);
53     else{
54       if (minimum) return min(t[v].f(on_points? pts[x] : x), x
  ↪  < m? get(2 * v + 1, l, m, x) : get(2 * v + 2, m, r, x));
55       else return max(t[v].f(on_points? pts[x] : x), x < m?
  ↪  get(2 * v + 1, l, m, x) : get(2 * v + 2, m, r, x));
56     }
57   }
58
59   void add_line(ll k, ll b){
60     add_line(0, 0, n, line(k, b));
61   }
62
63   ll get(ll x){
64     return get(0, 0, n, on_points? lower_bound(all(pts), x) -
  ↪  pts.begin() : x);
65   }; // Always pass the actual value of x, even if LCT is on
  ↪  points.
66 };
```

Persistent Segment Tree

- for RSQ

```
1  struct Node {
2    ll val;
3    Node *l, *r;
4
5    Node(ll x) : val(x), l(nullptr), r(nullptr) {}
6    Node(Node *ll, Node *rr) {
7      l = ll, r = rr;
8      val = 0;
9      if (l) val += l->val;
10     if (r) val += r->val;
```

```
11   }
12   Node(Node *cp) : val(cp->val), l(cp->l), r(cp->r) {}
13 };
14 const int N = 2e5 + 20;
15 ll a[N];
16 Node *roots[N];
17 int n, cnt = 1;
18 Node *build(int l = 1, int r = n) {
19   if (l == r) return new Node(a[l]);
20   int mid = (l + r) / 2;
21   return new Node(build(l, mid), build(mid + 1, r));
22 }
23 Node *update(Node *node, int val, int pos, int l = 1, int r =
  ↪  n) {
24   if (l == r) return new Node(val);
25   int mid = (l + r) / 2;
26   if (pos > mid)
27     return new Node(node->l, update(node->r, val, pos, mid +
  ↪  1, r));
28   else return new Node(update(node->l, val, pos, l, mid),
  ↪  node->r);
29 }
30 ll query(Node *node, int a, int b, int l = 1, int r = n) {
31   if (l > b || r < a) return 0;
32   if (l >= a && r <= b) return node->val;
33   int mid = (l + r) / 2;
34   return query(node->l, a, b, l, mid) + query(node->r, a, b,
  ↪  mid + 1, r);
35 }
```

Dynamic Programming

Sum over Subset DP

- Computes $f[A] = \sum_{B \subseteq A} a[B]$.
- Complexity: $O(2^n \cdot n)$.

```
1  forn(i, (1 << n)) f[i] = a[i];
2  forn(i, n) for (int mask = 0; mask < (1 << n); mask++) if
  ↪  ((mask >> i) & 1){
3    f[mask] += f[mask ^ (1 << i)];
4  }
```

Divide and Conquer DP

- Helps to compute 2D DP of the form:
- $dp[i][j] = \min_{0 \le k \le j-1} (dp[i-1][k] + cost(k+1, j))$
- **Necessary condition:** let $opt(i, j)$ be the optimal k for the state (i, j). Then, $opt(i, j) \le opt(i, j+1)$.
- **Sufficient condition:** $cost(a, d) + cost(b, c) \ge cost(a, c) + cost(b, d)$ where $a < b < c < d$.
- Complexity: $O(M \cdot N \cdot \log N)$ for computing $dp[M][N]$.

```
1  vll dp_old(N), dp_new(N);
2
3  void rec(int l, int r, int optl, int optr){
4    if (l > r) return;
5    int mid = (l + r) / 2;
6    pair<ll, int> best = {INF, optl};
7    for (int i = optl; i <= min(mid - 1, optr); i++){ // If k
  ↪  can be j, change to "i <= min(mid, optr)".
8      ll cur = dp_old[i] + cost(i + 1, mid);
9      if (cur < best.fi) best = {cur, i};
10   }
11   dp_new[mid] = best.fi;
12
13   rec(l, mid - 1, optl, best.se);
14   rec(mid + 1, r, best.se, optr);
15 }
16
17 // Computes the DP "by layers"
18 fill(all(dp_old), INF);
19 dp_old[0] = 0;
```

```
20    while (layers--){
21      rec(0, n, 0, n);
22      dp_old = dp_new;
23    }
```

Knuth's DP Optimization

- Computes DP of the form
- $dp[i][j] = \min_{i \leq k \leq j-1} (dp[i][k] + dp[k+1][j] + cost(i, j))$
- **Necessary Condition:** $opt(i, j-1) \leq opt(i, j) \leq opt(i+1, j)$
- **Sufficient Condition:** For $a \leq b \leq c \leq d$, $cost(b, c) \leq cost(a, d)$ AND $cost(a, d) + cost(b, c) \geq cost(a, c) + cost(b, d)$
- Complexity: $O(n^2)$

```
1   int N;
2   int dp[N][N], opt[N][N];
3   auto C = [&](int i, int j) {
4     // Implement cost function C.
5   };
6   forn(i, N) {
7     opt[i][i] = i;
8     // Initialize dp[i][i] according to the problem
9   }
10  for (int i = N-2; i >= 0; i--) {
11    for (int j = i+1; j < N; j++) {
12      int mn = INT_MAX;
13      int cost = C(i, j);
14      for (int k = opt[i][j-1]; k <= min(j-1, opt[i+1][j]); k++)
    ↪   {
15        if (mn >= dp[i][k] + dp[k+1][j] + cost) {
16          opt[i][j] = k;
17          mn = dp[i][k] + dp[k+1][j] + cost;
18        }
19      }
20      dp[i][j] = mn;
21    }
22  }
```

Miscellaneous

Ordered Set

```
1   #include <ext/pb_ds/assoc_container.hpp>
2   #include <ext/pb_ds/tree_policy.hpp>
3   using namespace __gnu_pbds;
4   typedef tree<int, null_type, less<int>, rb_tree_tag,
    ↪   tree_order_statistics_node_update> ordered_set;
```

Measuring Execution Time

```
1   ld tic = clock();
2   // execute algo…
3   ld tac = clock();
4   // Time in milliseconds
5   cerr << (tac - tic) / CLOCKS_PER_SEC * 1000 << endl;
6   // No need to comment out the print because it's done to cerr.
```

Setting Fixed D.P. Precision

```
1   cout << setprecision(d) << fixed;
2   // Each number is rounded to d digits after the decimal point,
    ↪   and truncated.
```

Debugging Flags

- Converts segfaults into Wrong Answers. Similarly one can catch SIGABRT (assertion failures) and SIGFPE (zero divisions)

```
1   signal(SIGSEGV, [](int) { Exit(0); });
```

Common Bugs and General Advice

- Check overflow, array bounds
- Check variable overloading
- Check special cases (n=1?)
- Do something instead of nothing, stay organized
- Write stuff down!
- Don't get stuck on one approach!

MIT troubleshoot.txt

- General:
- Write down most of your thoughts, even if you're not sure whether they're useful.
- Stay organized and don't leave papers all over the place!

- Pre-submit:
- Write a few simple test cases if sample is not enough.
- Are time limits close? If so, generate max cases.
- Is the memory usage fine?
- Could anything overflow?
- Remove debug output.
- Make sure to submit the right file.

- Wrong answer:
- Print your solution! Print debug output as well.
- Read the full problem statement again.
- Have you understood the problem correctly?
- Are you sure your algorithm works?
- Try writing a slow (but correct) solution.
- Can your algorithm handle the whole range of input?
- Did you consider corner cases (ex. n=1)?
- Is your output format correct? (including whitespace)
- Are you clearing all data structures between test cases?
- Any uninitialized variables?
- Any undefined behavior (array out of bounds)?
- Any overflows or NaNs (or shifting ll by >=64 bits)?
- Confusing N and M, i and j, etc.?
- Confusing ++i and i++?
- Return vs continue vs break?
- Are you sure the STL functions you use work as you think?
- Add some assertions, maybe resubmit.
- Create some test cases to run your algorithm on.
- Go through the algorithm for a simple case.

- Go through this list again.

- Explain your algorithm to a teammate.

- Ask a teammate to look at your code.

- Go for a small walk, e.g. to the toilet.

- Rewrite your solution from the start or let a teammate do it.

- Geometry:

- Work with ints if possible.

- Correctly account for numbers close to (but not) zero.

- Related: for functions like acos make sure absolute val of input is not (slightly) greater than one.

- Correctly deal with vertices that are collinear, concyclic, coplanar (in 3D), etc.

- Subtracting a point from every other (but not itself)?

- Runtime error:

- Have you tested all corner cases locally?

- Any uninitialized variables?

- Are you reading or writing outside the range of any vector?

- Any assertions that might fail?

- Any possible division by 0? (mod 0 for example)

- Any possible infinite recursion?

- Invalidated pointers or iterators?

- Are you using too much memory?

- Debug with resubmits (e.g. remapped signals, see Various).

- Time limit exceeded:

- Do you have any possible infinite loops?

- What's your complexity? Large TL does not mean that something simple (like NlogN) isn't intended.

- Are you copying a lot of unnecessary data? (References)

- Avoid vector, map. (use arrays/unordered_map)

- How big is the input and output? (consider FastIO)

- What do your teammates think about your algorithm?

- Calling count() on multiset?

- Memory limit exceeded:

- What is the max amount of memory your algorithm should need?

- Are you clearing all data structures between test cases?