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;
#define pb push_back
#define sz(x) (int)(x).size()
#define fi first
#define se second
#define endl '\n'
```

Kevin's template

```cpp
// paste Kaurov's Template, minus last line
typedef vector<int> vi;
typedef vector<ll> vll;
typedef pair<int, int> pii;
typedef pair<ll, ll> pll;
const char nl = '\n';
#define forn(i, n) for (int i = 0; i < int(n); i++)
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 basics

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

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

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

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

ld sq(ld a){
  return a * a;
}
ld smul(point a, point b){
  return a.x * b.x + a.y * b.y;
}
ld vmul(point a, point b){
  return a.x * b.y - a.y * b.x;
}
ld dist(point a, point b){
  return (a - b).len();
}
bool acw(point a, point b){
  return vmul(a, b) > -EPS;
}
bool cw(point a, point b){
  return vmul(a, b) < EPS;
}
int sgn(ld x){
  return (x > EPS) - (x < EPS);
}
```

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(point p1, point 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(vmul(point(l1.a, l1.b), point(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

```cpp
// {p, 0} - unique intersection, {p, 1} - infinite, {p,
  ↪  2} - none
pair<point, int> line_inter(line l1, line l2){
  if (parallel(l1, l2)){
```

```
4      return {point(), l1 == l2? 1 : 2};
5    }
6    return {point(
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(point p, point a, point b){
15    return abs(vmul(p - a, p - b)) < EPS && smul(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<point> segment_inter(point a, point b, point c,
↪    point d) {
24    auto oa = vmul(d - c, a - c), ob = vmul(d - c, b - c),
↪    oc = vmul(b - a, c - a), od = vmul(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<point> 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(point p, point a, point b){
3    return vmul(b - a, p - a) / (b - a).len();
4  }
5
6  // Distance from p to segment ab
7  ld segment_dist(point p, point a, point b){
8    if (a == b) return (p - a).len();
9    auto d = (a - b).abs2(), t = min(d, max((ld)0, smul(p
↪    - a, b - a)));
10    return ((p - a) * d - (b - a) * t).len() / d;
11  }
```

Polygon area

```
1  ld area(vector<point> pts){
2    int n = sz(pts);
3    ld ans = 0;
4    for (int i = 0; i < n; i++){
5      ans += vmul(pts[i], pts[(i + 1) % n]);
6    }
7    return abs(ans) / 2;
8  }
```

Convex hull

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

```
1  vector<point> convex_hull(vector<point> pts){
2    sort(all(pts));
3    pts.erase(unique(all(pts)), pts.end());
4    vector<point> 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<point>& 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(point p, vector<point>& 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(point p, vector<point>& 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)) * vmul(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<point>& 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.
    vector<point> minkowski_sum(vector<point> P,
↪    vector<point> Q){
      minkowski_rotate(P);
      minkowski_rotate(Q);
      P.pb(P[0]);
      Q.pb(Q[0]);
```

```
17   vector<point> 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 = vmul(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, smul, vmul
2  const ld EPS = 1e-9;
3
4  int sgn(ld a){
5    return (a > EPS) - (a < -EPS);
6  }
7  int half(point p){
8    return p.y != 0? sgn(p.y) : -sgn(p.x);
9  }
10 bool angle_comp(point a, point b){
11   int A = half(a), B = half(b);
12   return A == B? vmul(a, b) > 0 : A < B;
13 }
14 struct ray{
15   point p, dp; // origin, direction
16   ray(point p_, point dp_){
17     p = p_, dp = dp_;
18   }
19   point isect(ray l){
20     return p + dp * (vmul(l.dp, l.p - p) / vmul(l.dp,
   ↪ dp));
21   }
22   bool operator<(ray l){
23     return angle_comp(dp, l.dp);
24   }
25 };
26 vector<point> 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({point(0, 0), point(1, 0)});
29   rays.pb({point(DX, 0), point(0, 1)});
30   rays.pb({point(DX, DY), point(-1, 0)});
31   rays.pb({point(0, DY), point(0, -1)});
32   sort(all(rays));
33   {
34     vector<ray> nrays;
35     for (auto t : rays){
36       if (nrays.empty() || vmul(nrays.back().dp, t.dp) >
   ↪ EPS){
37         nrays.pb(t);
38         continue;
39       }
40       if (vmul(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     point p1 = a.isect(b), p2 = b.isect(c);
46     if (smul(p2 - p1, b.dp) <= EPS){
47       if (vmul(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<point> 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 }
```

Strings

```
1  vector<int> prefix_function(string s){
2    int n = sz(s);
3    vector<int> 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 vector<int> kmp(string s, string k){
14   string st = k + "#" + s;
15   vector<int> res;
16   auto pi = prefix_function(st);
17   for (int i = 0; i < sz(st); i++){
18     if (pi[i] == sz(k)){
19       res.pb(i - 2 * sz(k));
20     }
21   }
22   return res;
23 }
24 vector<int> z_function(string s){
25   int n = sz(s);
26   vector<int> z(n);
27   int l = 0, r = 0;
28   for (int i = 1; i < n; i++){
29     if (r >= i) z[i] = min(z[i - l], r - i + 1);
30     while (i + z[i] < n && s[z[i]] == s[i + z[i]]){
31       z[i]++;
32     }
33     if (i + z[i] - 1 > r){
34       l = i, r = i + z[i] - 1;
35     }
36   }
37   return z;
38 }
```

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<vector<int>, vector<int>> 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   vector<int> 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   vector<int> even(sz(s)), odd(sz(s));
18   for (int i = 0; i < sz(s); i++){
19     even[i] = p[2 * i + 1] / 2, odd[i] = p[2 * i + 2] /
   ↪ 2;
```

```
20        }
21        return {even, odd};
22    }
```

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<vector<int>> adj;
10     int n, m = 0;
11     int s, t;
12     vector<int> 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 <ext/pb_ds/priority_queue.hpp>
2  template <typename T, typename C>
```

```
3  class MCMF {
4  public:
5      static constexpr T eps = (T) 1e-9;
6
7      struct edge {
8          int from;
9          int to;
10         T c;
11         T f;
12         C cost;
13     };
14
15     int n;
16     vector<vector<int>> g;
17     vector<edge> edges;
18     vector<C> d;
19     vector<C> pot;
20     __gnu_pbds::priority_queue<pair<C, int>> q;
21     vector<typename decltype(q)::point_iterator> its;
22     vector<int> pe;
23     const C INF_C = numeric_limits<C>::max() / 2;
24
25     explicit MCMF(int n_) : n(n_), g(n), d(n), pot(n, 0),
↪  its(n), pe(n) {}
26
27     int add(int from, int to, T forward_cap, C edge_cost,
↪  T backward_cap = 0) {
28         assert(0 <= from && from < n && 0 <= to && to < n);
29         assert(forward_cap >= 0 && backward_cap >= 0);
30         int id = static_cast<int>(edges.size());
31         g[from].push_back(id);
32         edges.push_back({from, to, forward_cap, 0,
↪  edge_cost});
33         g[to].push_back(id + 1);
34         edges.push_back({to, from, backward_cap, 0,
↪  -edge_cost});
35         return id;
36     }
37
38     void expath(int st) {
39         fill(d.begin(), d.end(), INF_C);
40         q.clear();
41         fill(its.begin(), its.end(), q.end());
42         its[st] = q.push({pot[st], st});
43         d[st] = 0;
44         while (!q.empty()) {
45             int i = q.top().second;
46             q.pop();
47             its[i] = q.end();
48             for (int id : g[i]) {
49                 const edge &e = edges[id];
50                 int j = e.to;
51                 if (e.c - e.f > eps && d[i] + e.cost < d[j]) {
52                     d[j] = d[i] + e.cost;
53                     pe[j] = id;
54                     if (its[j] == q.end()) {
55                         its[j] = q.push({pot[j] - d[j], j});
```

```cpp
      } else {
        q.modify(its[j], {pot[j] - d[j], j});
      }
    }
  }
}
swap(d, pot);
}

pair<T, C> max_flow(int st, int fin) {
  T flow = 0;
  C cost = 0;
  bool ok = true;
  for (auto& e : edges) {
    if (e.c - e.f > eps && e.cost + pot[e.from] -
↪ pot[e.to] < 0) {
      ok = false;
      break;
    }
  }
  if (ok) {
    expath(st);
  } else {
    vector<int> deg(n, 0);
    for (int i = 0; i < n; i++) {
      for (int eid : g[i]) {
        auto& e = edges[eid];
        if (e.c - e.f > eps) {
          deg[e.to] += 1;
        }
      }
    }
    vector<int> que;
    for (int i = 0; i < n; i++) {
      if (deg[i] == 0) {
        que.push_back(i);
      }
    }
    for (int b = 0; b < (int) que.size(); b++) {
      for (int eid : g[que[b]]) {
        auto& e = edges[eid];
        if (e.c - e.f > eps) {
          deg[e.to] -= 1;
          if (deg[e.to] == 0) {
            que.push_back(e.to);
          }
        }
      }
    }
    fill(pot.begin(), pot.end(), INF_C);
    pot[st] = 0;
    if (static_cast<int>(que.size()) == n) {
      for (int v : que) {
        if (pot[v] < INF_C) {
          for (int eid : g[v]) {
            auto& e = edges[eid];
            if (e.c - e.f > eps) {
              if (pot[v] + e.cost < pot[e.to]) {
                pot[e.to] = pot[v] + e.cost;
                pe[e.to] = eid;
              }
            }
          }
        }
      }
    } else {
      que.assign(1, st);
      vector<bool> in_queue(n, false);
      in_queue[st] = true;
      for (int b = 0; b < (int) que.size(); b++) {
        int i = que[b];
        in_queue[i] = false;
        for (int id : g[i]) {
          const edge &e = edges[id];
          if (e.c - e.f > eps && pot[i] + e.cost <
↪ pot[e.to]) {
            pot[e.to] = pot[i] + e.cost;
            pe[e.to] = id;
            if (!in_queue[e.to]) {
              que.push_back(e.to);
              in_queue[e.to] = true;
            }
          }
        }
      }
    }
  }
  while (pot[fin] < INF_C) {
    T push = numeric_limits<T>::max();
    int v = fin;
    while (v != st) {
      const edge &e = edges[pe[v]];
      push = min(push, e.c - e.f);
      v = e.from;
    }
    v = fin;
    while (v != st) {
      edge &e = edges[pe[v]];
      e.f += push;
      edge &back = edges[pe[v] ^ 1];
      back.f -= push;
      v = e.from;
    }
    flow += push;
    cost += push * pot[fin];
    expath(st);
  }
  return {flow, cost};
}
};

// Examples: MCMF<int, int> g(n); g.add(u,v,c,w,0);
↪ g.max_flow(s,t).
// To recover flow through original edges: iterate over
↪ even indices in edges.
```

Graphs

Kuhn's algorithm for bipartite matching

```cpp
/*
The graph is split into 2 halves of n1 and n2 vertices.
Complexity: O(n1 * m). Usually runs much faster. MUCH
↪ FASTER!!!
*/
const int N = 305;

vector<int> g[N]; // Stores edges from left half to
↪ right.
bool used[N]; // Stores if vertex from left half is
↪ used.
int mt[N]; // For every vertex in right half, stores to
↪ which vertex in left half it's matched (-1 if not
↪ matched).

bool try_dfs(int v){
  if (used[v]) return false;
  used[v] = 1;
  for (auto u : g[v]){
    if (mt[u] == -1 || try_dfs(mt[u])){
      mt[u] = v;
      return true;
    }
  }
  return false;
}

int main(){
// ......
  for (int i = 1; i <= n2; i++) mt[i] = -1;
  for (int i = 1; i <= n1; i++) used[i] = 0;
  for (int i = 1; i <= n1; i++){
    if (try_dfs(i)){
      for (int j = 1; j <= n1; j++) used[j] = 0;
    }
  }
  vector<pair<int, int>> ans;
  for (int i = 1; i <= n2; i++){
    if (mt[i] != -1) ans.pb({mt[i], i});
  }
}

// Finding maximal independent set: size = # of nodes -
↪ # of edges in matching.
// To construct: launch Kuhn-like DFS from unmatched
↪ nodes in the left half.
// Independent set = visited nodes in left half +
↪ unvisited in right half.
// 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   vector<int> 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       vector<int> 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  vector<int> 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  }
```

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<vector<int>>& g, int* idx) {
2       int n = g.size(), ct = 0;
3       int out[n];
4       vector<int> 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      vector<int> 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,vector<int>> sat2(int n, vector<pair<int,int>>&
    ↪    clauses) {
45      vector<int> ans(n);
46      vector<vector<int>> 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   vector<int> 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

```cpp
// order stores the nodes in the queried set
sort(all(order), [&] (int u, int v){return tin[u] <
↪ tin[v];});
int m = sz(order);
for (int i = 1; i < m; i++){
    order.pb(lca(order[i], order[i - 1]));
}
sort(all(order), [&] (int u, int v){return tin[u] <
↪ tin[v];});
order.erase(unique(all(order)), order.end());
vector<int> stk{order[0]};
for (int i = 1; i < sz(order); i++){
    int v = order[i];
    while (tout[stk.back()] < tout[v]) stk.pop_back();
    int u = stk.back();
    vg[u].pb({v, dep[v] - dep[u]});
    stk.pb(v);
}
```

HLD on Edges DFS

```cpp
void dfs1(int v, int p, int d){
  par[v] = p;
  for (auto e : g[v]){
    if (e.fi == p){
      g[v].erase(find(all(g[v]), e));
      break;
    }
  }
  dep[v] = d;
  sz[v] = 1;
  for (auto [u, c] : g[v]){
    dfs1(u, v, d + 1);
    sz[v] += sz[u];
  }
  if (!g[v].empty()) iter_swap(g[v].begin(),
↪ max_element(all(g[v]), comp));
}
void dfs2(int v, int rt, int c){
  pos[v] = sz(a);
  a.pb(c);
  root[v] = rt;
  for (int i = 0; i < sz(g[v]); i++){
    auto [u, c] = g[v][i];
    if (!i) dfs2(u, rt, c);
    else dfs2(u, u, c);
  }
}
int getans(int u, int v){
  int res = 0;
  for (; root[u] != root[v]; v = par[root[v]]){
    if (dep[root[u]] > dep[root[v]]) swap(u, v);
    res = max(res, rmq(0, 0, n - 1, pos[root[v]],
↪ pos[v]));
  }
  if (pos[u] > pos[v]) swap(u, v);
  return max(res, rmq(0, 0, n - 1, pos[u] + 1, pos[v]));
}
```

Centroid Decomposition

```cpp
vector<char> res(n), seen(n), sz(n);
function<int(int, int)> get_size = [&](int node, int fa)
↪ {
  sz[node] = 1;
  for (auto& ne : g[node]) {
    if (ne == fa || seen[ne]) continue;
    sz[node] += get_size(ne, node);
  }
  return sz[node];
};
function<int(int, int, int)> find_centroid = [&](int
↪ node, int fa, int t) {
  for (auto& ne : g[node])
    if (ne != fa && !seen[ne] && sz[ne] > t / 2) return
↪ find_centroid(ne, node, t);
  return node;
};
function<void(int, char)> solve = [&](int node, char
↪ cur) {
  get_size(node, -1); auto c = find_centroid(node, -1,
↪ sz[node]);
  seen[c] = 1, res[c] = cur;
  for (auto& ne : g[c]) {
    if (seen[ne]) continue;
    solve(ne, char(cur + 1)); // we can pass c here to
↪ build tree
  }
};
```

Math

Binary exponentiation

```cpp
ll power(ll a, ll b){
  ll res = 1;
  for (; b; a = a * a % MOD, b >>= 1){
    if (b & 1) res = res * a % MOD;
  }
  return res;
}
```

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

```cpp
const int N = 100, MOD = 1e9 + 7;

struct matrix{
  ll m[N][N];
  int n;
  matrix(){
    n = N;
    memset(m, 0, sizeof(m));
  };
  matrix(int n_){
    n = n_;
    memset(m, 0, sizeof(m));
  };
  matrix(int n_, ll val){
    n = n_;
    memset(m, 0, sizeof(m));
    for (int i = 0; i < n; i++) m[i][i] = val;
  };

  matrix operator* (matrix oth){
    matrix res(n);
    for (int i = 0; i < n; i++){
      for (int j = 0; j < n; j++){
        for (int k = 0; k < n; k++){
          res.m[i][j] = (res.m[i][j] + m[i][k] *
↪ oth.m[k][j]) % MOD;
        }
      }
    }
    return res;
  }
};

matrix power(matrix a, ll b){
  matrix res(a.n, 1);
  for (; b; a = a * a, b >>= 1){
    if (b & 1) res = res * a;
  }
  return res;
}
```

Extended Euclidean Algorithm

```cpp
// gives (x, y) for ax + by = g
// solutions given (x0, y0): a(x0 + kb/g) + b(y0 - ka/g)
↪ = g
int gcd(int a, int b, int& x, int& y) {
  x = 1, y = 0; int sum1 = a;
  int x2 = 0, y2 = 1, sum2 = b;
  while (sum2) {
    int q = sum1 / sum2;
    tie(x, x2) = make_tuple(x2, x - q * x2);
    tie(y, y2) = make_tuple(y2, y - q * y2);
    tie(sum1, sum2) = make_tuple(sum2, sum1 - q * sum2);
  }
  return sum1;
}
```

Linear Sieve

- Mobius Function

```cpp
vector<int> prime;
bool is_composite[MAX_N];
int mu[MAX_N];

void sieve(int n){
  fill(is_composite, is_composite + n, 0);
  mu[1] = 1;
  for (int i = 2; i < n; i++){
    if (!is_composite[i]){
      prime.push_back(i);
      mu[i] = -1; //i is prime
    }
    for (int j = 0; j < prime.size() && i * prime[j] < n; j++){
      is_composite[i * prime[j]] = true;
      if (i % prime[j] == 0){
        mu[i * prime[j]] = 0; //prime[j] divides i
        break;
      } else {
        mu[i * prime[j]] = -mu[i]; //prime[j] does not divide i
      }
    }
  }
}
```

- Euler's Totient Function

```cpp
vector<int> prime;
bool is_composite[MAX_N];
int phi[MAX_N];

void sieve(int n){
  fill(is_composite, is_composite + n, 0);
  phi[1] = 1;
  for (int i = 2; i < n; i++){
    if (!is_composite[i]){
      prime.push_back (i);
      phi[i] = i - 1; //i is prime
    }
    for (int j = 0; j < prime.size () && i * prime[j] < n; j++){
      is_composite[i * prime[j]] = true;
      if (i % prime[j] == 0){
        phi[i * prime[j]] = phi[i] * prime[j]; //prime[j] divides i
        break;
      } else {
        phi[i * prime[j]] = phi[i] * phi[prime[j]];
        //prime[j] does not divide i
      }
    }
  }
}
```

Gaussian Elimination

```cpp
bool is_0(Z v) { return v.x == 0; }
Z abs(Z v) { return v; }
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];
    }
    vector<int> 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();
  for (int i = 0; i < h; i++) a[i].push_back(b[i]);
  int sol = gaussian_elimination(a, w);
  if(!sol) return {0, vector<T>()};
  vector<T> x(w, 0);
  for (int i = 0; i < h; i++) {
    for (int j = 0; j < w; j++) {
      if (!is_0(a[i][j])) {
        x[j] = a[i][w] / a[i][j];
        break;
      }
    }
  }
  return {sol, x};
}
```

is_prime

- (Miller–Rabin primality test)

```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;
}

typedef __int128_t i128;

ll pollard_rho(ll x) {
  ll s = 0, t = 0, c = rng() % (x - 1) + 1;
  ll stp = 0, goal = 1, val = 1;
```

```
6      for (goal = 1;; goal *= 2, s = t, val = 1) {
7        for (stp = 1; stp <= goal; ++stp) {
8          t = ll(((i128)t * t + c) % x);
9          val = ll((i128)val * abs(t - s) % x);
10         if ((stp % 127) == 0) {
11           ll d = gcd(val, x);
12           if (d > 1) return d;
13         }
14       }
15       ll d = gcd(val, x);
16       if (d > 1) return d;
17     }
18 }
19
20 ll get_max_factor(ll _x) {
21   ll max_factor = 0;
22   function<void(ll)> fac = [&](ll x) {
23     if (x <= max_factor || x < 2) return;
24     if (is_prime(x)) {
25       max_factor = max_factor > x ? max_factor : x;
26       return;
27     }
28     ll p = x;
29     while (p >= x) p = pollard_rho(x);
30     while ((x % p) == 0) x /= p;
31     fac(x), fac(p);
32   };
33   fac(_x);
34   return max_factor;
35 }
```

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

```
23   for (vector<ll> poly = {0, 1}; k; poly =
↪  poly_mult_mod(poly, poly, c), k >>= 1){
24     if (k & 1) res = poly_mult_mod(res, poly, c);
25   }
26   ll ans = 0;
27   for (int i = 0; i < min(sz(res), sz(c)); i++) ans =
↪  (ans + s[i] * res[i]) % MOD;
28   return ans;
29 }
```

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

```
1 void ntt(vector<ll>& a, int f) {
2   int n = int(a.size());
3   vector<ll> w(n);
4   vector<int> rev(n);
5   for (int i = 0; i < n; i++) rev[i] = (rev[i / 2] / 2)
↪  | ((i & 1) * (n / 2));
6   for (int i = 0; i < n; i++) {
7     if (i < rev[i]) swap(a[i], a[rev[i]]);
8   }
9   ll wn = power(f ? (MOD + 1) / 3 : 3, (MOD - 1) / n);
10  w[0] = 1;
11  for (int i = 1; i < n; i++) w[i] = w[i - 1] * wn %
↪  MOD;
12  for (int mid = 1; mid < n; mid *= 2) {
13    for (int i = 0; i < n; i += 2 * mid) {
14      for (int j = 0; j < mid; j++) {
15        ll x = a[i + j], y = a[i + j + mid] * w[n / (2 *
↪  mid) * j] % MOD;
16        a[i + j] = (x + y) % MOD, a[i + j + mid] = (x +
↪  MOD - y) % MOD;
17      }
18    }
19  }
20  if (f) {
21    ll iv = power(n, MOD - 2);
```

```
22        for (auto& x : a) x = x * iv % MOD;
23    }
24  }
25  vector<ll> mul(vector<ll> a, vector<ll> b) {
26    int n = 1, m = (int)a.size() + (int)b.size() - 1;
27    while (n < m) n *= 2;
28    a.resize(n), b.resize(n);
29    ntt(a, 0), ntt(b, 0); // if squaring, you can save one
   ↪ NTT here
30    for (int i = 0; i < n; i++) a[i] = a[i] * b[i] % MOD;
31    ntt(a, 1);
32    a.resize(m);
33    return a;
34  }
```

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     vector<int> rev(len);
8     for (int i = 0; i < n; i++) a[i].real(aa[i]);
9     for (int i = 0; i < m; i++) b[i].real(bb[i]);
10    for (int i = 0; i < len; i++) rev[i] = (rev[i >> 1] >>
   ↪ 1) | ((i & 1) << (bit - 1));
11    auto fft = [&](vector<complex<ld>>& p, int inv) {
12      for (int i = 0; i < len; i++)
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        for (int i = 0; i < len; i++)
   ↪ p[i].real(p[i].real() / len);
26      }
27    };
28    fft(a, 0), fft(b, 0);
29    for (int i = 0; i < len; i++) a[i] = a[i] * b[i];
30    fft(a, 1);
31    a.resize(n + m - 1);
32    vector<ld> res(n + m - 1);
33    for (int i = 0; i < n + m - 1; i++) res[i] =
   ↪ a[i].real();
34    return res;
35  };
```

MIT's FFT/NTT, Polynomial mod/log/exp Template

- For integers rounding works if $(|a| + |b|) \max(a, b) < \sim 10^9$, or in theory maybe 10^6
- $\frac{1}{P(x)}$ 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   // use #define FFT 1 to use FFT instead of NTT (default)
2   // Examples:
3   // poly a(n+1); // constructs degree n poly
4   // a[0].v = 10; // assigns constant term a_0 = 10
5   // poly b = exp(a);
6   // poly is vector<num>
7   // for NTT, num stores just one int named v
8   // for FFT, num stores two doubles named x (real), y
   ↪ (imag)

10  #define sz(x) ((int)x.size())
11  #define rep(i, j, k) for (int i = int(j); i < int(k);
   ↪ i++)
12  #define trav(a, x) for (auto &a : x)
13  #define per(i, a, b) for (int i = (b)-1; i >= (a); --i)
14  using ll = long long;
15  using vi = vector<int>;

17  namespace fft {
18  #if FFT
19  // FFT
20  using dbl = double;
21  struct num {
22    dbl x, y;
23    num(dbl x_ = 0, dbl y_ = 0): x(x_), y(y_) {}
24  };
25  inline num operator+(num a, num b) {
26    return num(a.x + b.x, a.y + b.y);
27  }
28  inline num operator-(num a, num b) {
29    return num(a.x - b.x, a.y - b.y);
30  }
31  inline num operator*(num a, num b) {
32    return num(a.x * b.x - a.y * b.y, a.x * b.y + a.y *
   ↪ b.x);
33  }
34  inline num conj(num a) { return num(a.x, -a.y); }
35  inline num inv(num a) {
36    dbl n = (a.x * a.x + a.y * a.y);
37    return num(a.x / n, -a.y / n);
38  }

40  #else
41  // NTT
42  const int mod = 998244353, g = 3;
```

```
    // For p < 2^30 there is also (5 << 25, 3), (7 << 26,
   ↪ 3),
44  // (479 << 21, 3) and (483 << 21, 5). Last two are >
   ↪ 10^9.
    struct num {
      int v;
      num(ll v_ = 0): v(int(v_ % mod)) {
        if (v < 0) v += mod;
      }
      explicit operator int() const { return v; }
    };
    inline num operator+(num a, num b) { return num(a.v +
   ↪ b.v); }
    inline num operator-(num a, num b) {
      return num(a.v + mod - b.v);
    }
    inline num operator*(num a, num b) {
      return num(1ll * a.v * b.v);
    }
    inline num pow(num a, int b) {
      num r = 1;
      do {
        if (b & 1) r = r * a;
        a = a * a;
      } while (b >>= 1);
      return r;
    }
    inline num inv(num a) { return pow(a, mod - 2); }

    #endif
    using vn = vector<num>;
    vi rev({0, 1});
    vn rt(2, num(1)), fa, fb;
    inline void init(int n) {
      if (n <= sz(rt)) return;
      rev.resize(n);
      rep(i, 0, n) rev[i] = (rev[i >> 1] | ((i & 1) * n)) >>
   ↪ 1;
      rt.reserve(n);
      for (int k = sz(rt); k < n; k *= 2) {
        rt.resize(2 * k);
    #if FFT
        double a = M_PI / k;
        num z(cos(a), sin(a)); // FFT
    #else
        num z = pow(num(g), (mod - 1) / (2 * k)); // NTT
    #endif
        rep(i, k / 2, k) rt[2 * i] = rt[i],
                         rt[2 * i + 1] = rt[i] * z;
      }
    }
    inline void fft(vector<num>& a, int n) {
      init(n);
      int s = __builtin_ctz(sz(rev) / n);
      rep(i, 0, n) if (i < rev[i] >> s) swap(a[i], a[rev[i]
   ↪ >> s]);
      for (int k = 1; k < n; k *= 2)
```

```cpp
 95      for (int i = 0; i < n; i += 2 * k) rep(j, 0, k) {
 96        num t = rt[j + k] * a[i + j + k];
 97        a[i + j + k] = a[i + j] - t;
 98        a[i + j] = a[i + j] + t;
 99      }
100  }
101  // Complex/NTT
102  vn multiply(vn a, vn b) {
103    int s = sz(a) + sz(b) - 1;
104    if (s <= 0) return {};
105    int L = s > 1 ? 32 - __builtin_clz(s - 1) : 0, n = 1 << L;
106    a.resize(n), b.resize(n);
107    fft(a, n);
108    fft(b, n);
109    num d = inv(num(n));
110    rep(i, 0, n) a[i] = a[i] * b[i] * d;
111    reverse(a.begin() + 1, a.end());
112    fft(a, n);
113    a.resize(s);
114    return a;
115  }
116  // Complex/NTT power-series inverse
117  // Doubles b as b[:n] = (2 - a[:n] * b[:n/2]) * b[:n/2]
118  vn inverse(const vn& a) {
119    if (a.empty()) return {};
120    vn b({inv(a[0])});
121    b.reserve(2 * a.size());
122    while (sz(b) < sz(a)) {
123      int n = 2 * sz(b);
124      b.resize(2 * n, 0);
125      if (sz(fa) < 2 * n) fa.resize(2 * n);
126      fill(fa.begin(), fa.begin() + 2 * n, 0);
127      copy(a.begin(), a.begin() + min(n, sz(a)),
           fa.begin());
128      fft(b, 2 * n);
129      fft(fa, 2 * n);
130      num d = inv(num(2 * n));
131      rep(i, 0, 2 * n) b[i] = b[i] * (2 - fa[i] * b[i]) *
           d;
132      reverse(b.begin() + 1, b.end());
133      fft(b, 2 * n);
134      b.resize(n);
135    }
136    b.resize(a.size());
137    return b;
138  }
139  #if FFT
140  // Double multiply (num = complex)
141  using vd = vector<double>;
142  vd multiply(const vd& a, const vd& b) {
143    int s = sz(a) + sz(b) - 1;
144    if (s <= 0) return {};
145    int L = s > 1 ? 32 - __builtin_clz(s - 1) : 0, n = 1 << L;
146    if (sz(fa) < n) fa.resize(n);
147    if (sz(fb) < n) fb.resize(n);
148    fill(fa.begin(), fa.begin() + n, 0);
149    rep(i, 0, sz(a)) fa[i].x = a[i];
150    rep(i, 0, sz(b)) fa[i].y = b[i];
151    fft(fa, n);
152    trav(x, fa) x = x * x;
153    rep(i, 0, n) fb[i] = fa[(n - i) & (n - 1)] -
         conj(fa[i]);
154    fft(fb, n);
155    vd r(s);
156    rep(i, 0, s) r[i] = fb[i].y / (4 * n);
157    return r;
158  }
159  // Integer multiply mod m (num = complex)
160  vi multiply_mod(const vi& a, const vi& b, int m) {
161    int s = sz(a) + sz(b) - 1;
162    if (s <= 0) return {};
163    int L = s > 1 ? 32 - __builtin_clz(s - 1) : 0, n = 1 << L;
164    if (sz(fa) < n) fa.resize(n);
165    if (sz(fb) < n) fb.resize(n);
166    rep(i, 0, sz(a)) fa[i] =
167      num(a[i] & ((1 << 15) - 1), a[i] >> 15);
168    fill(fa.begin() + sz(a), fa.begin() + n, 0);
169    rep(i, 0, sz(b)) fb[i] =
170      num(b[i] & ((1 << 15) - 1), b[i] >> 15);
171    fill(fb.begin() + sz(b), fb.begin() + n, 0);
172    fft(fa, n);
173    fft(fb, n);
174    double r0 = 0.5 / n; // 1/2n
175    rep(i, 0, n / 2 + 1) {
176      int j = (n - i) & (n - 1);
177      num g0 = (fb[i] + conj(fb[j])) * r0;
178      num g1 = (fb[i] - conj(fb[j])) * r0;
179      swap(g1.x, g1.y);
180      g1.y *= -1;
181      if (j != i) {
182        swap(fa[j], fa[i]);
183        fb[j] = fa[j] * g1;
184        fa[j] = fa[j] * g0;
185      }
186      fb[i] = fa[i] * conj(g1);
187      fa[i] = fa[i] * conj(g0);
188    }
189    fft(fa, n);
190    fft(fb, n);
191    vi r(s);
192    rep(i, 0, s) r[i] =
193      int((ll(fa[i].x + 0.5) + (ll(fa[i].y + 0.5) % m <<
         15) +
194          (ll(fb[i].x + 0.5) % m << 15) +
195          (ll(fb[i].y + 0.5) % m << 30)) %
196        m);
197    return r;
198  }
199  #endif
200  } // namespace fft
201  // For multiply_mod, use num = modnum, poly =
        vector<num>
202  using fft::num;
203  using poly = fft::vn;
204  using fft::multiply;
205  using fft::inverse;
206
207  poly& operator+=(poly& a, const poly& b) {
208    if (sz(a) < sz(b)) a.resize(b.size());
209    rep(i, 0, sz(b)) a[i] = a[i] + b[i];
210    return a;
211  }
212  poly operator+(const poly& a, const poly& b) {
213    poly r = a;
214    r += b;
215    return r;
216  }
217  poly& operator-=(poly& a, const poly& b) {
218    if (sz(a) < sz(b)) a.resize(b.size());
219    rep(i, 0, sz(b)) a[i] = a[i] - b[i];
220    return a;
221  }
222  poly operator-(const poly& a, const poly& b) {
223    poly r = a;
224    r -= b;
225    return r;
226  }
227  poly operator*(const poly& a, const poly& b) {
228    return multiply(a, b);
229  }
230  poly& operator*=(poly& a, const poly& b) { return a = a
         * b; }
231
232  poly& operator*=(poly& a, const num& b) { // Optional
233    trav(x, a) x = x * b;
234    return a;
235  }
236  poly operator*(const poly& a, const num& b) {
237    poly r = a;
238    r *= b;
239    return r;
240  }
241  // Polynomial floor division; no leading 0's please
242  poly operator/(poly a, poly b) {
243    if (sz(a) < sz(b)) return {};
244    int s = sz(a) - sz(b) + 1;
245    reverse(a.begin(), a.end());
246    reverse(b.begin(), b.end());
247    a.resize(s);
248    b.resize(s);
249    a = a * inverse(move(b));
250    a.resize(s);
251    reverse(a.begin(), a.end());
252    return a;
253  }
254  poly& operator/=(poly& a, const poly& b) { return a = a
         / b; }
255  poly& operator%=(poly& a, const poly& b) {
256    if (sz(a) >= sz(b)) {
257      poly c = (a / b) * b;
```

```
258      a.resize(sz(b) - 1);
259      rep(i, 0, sz(a)) a[i] = a[i] - c[i];
260    }
261    return a;
262  }
263  poly operator%(const poly& a, const poly& b) {
264    poly r = a;
265    r %= b;
266    return r;
267  }
268  // Log/exp/pow
269  poly deriv(const poly& a) {
270    if (a.empty()) return {};
271    poly b(sz(a) - 1);
272    rep(i, 1, sz(a)) b[i - 1] = a[i] * i;
273    return b;
274  }
275  poly integ(const poly& a) {
276    poly b(sz(a) + 1);
277    b[1] = 1; // mod p
278    rep(i, 2, sz(b)) b[i] =
279      b[fft::mod % i] * (-fft::mod / i); // mod p
280    rep(i, 1, sz(b)) b[i] = a[i - 1] * b[i]; // mod p
281    //rep(i,1,sz(b)) b[i]=a[i-1]*inv(num(i)); // else
282    return b;
283  }
284  poly log(const poly& a) { // MUST have a[0] == 1
285    poly b = integ(deriv(a) * inverse(a));
286    b.resize(a.size());
287    return b;
288  }
289  poly exp(const poly& a) { // MUST have a[0] == 0
290    poly b(1, num(1));
291    if (a.empty()) return b;
292    while (sz(b) < sz(a)) {
293      int n = min(sz(b) * 2, sz(a));
294      b.resize(n);
295      poly v = poly(a.begin(), a.begin() + n) - log(b);
296      v[0] = v[0] + num(1);
297      b *= v;
298      b.resize(n);
299    }
300    return b;
301  }
302  poly pow(const poly& a, int m) { // m >= 0
303    poly b(a.size());
304    if (!m) {
305      b[0] = 1;
306      return b;
307    }
308    int p = 0;
309    while (p < sz(a) && a[p].v == 0) ++p;
310    if (1ll * m * p >= sz(a)) return b;
311    num mu = pow(a[p], m), di = inv(a[p]);
312    poly c(sz(a) - m * p);
313    rep(i, 0, sz(c)) c[i] = a[i + p] * di;
314    c = log(c);
315    trav(v, c) v = v * m;
```

```
316    c = exp(c);
317    rep(i, 0, sz(c)) b[i + m * p] = c[i] * mu;
318    return b;
319  }
320  // Multipoint evaluation/interpolation
321
322  vector<num> eval(const poly& a, const vector<num>& x) {
323    int n = sz(x);
324    if (!n) return {};
325    vector<poly> up(2 * n);
326    rep(i, 0, n) up[i + n] = poly({0 - x[i], 1});
327    per(i, 1, n) up[i] = up[2 * i] * up[2 * i + 1];
328    vector<poly> down(2 * n);
329    down[1] = a % up[1];
330    rep(i, 2, 2 * n) down[i] = down[i / 2] % up[i];
331    vector<num> y(n);
332    rep(i, 0, n) y[i] = down[i + n][0];
333    return y;
334  }
335
336  poly interp(const vector<num>& x, const vector<num>& y)
   {
337    int n = sz(x);
338    assert(n);
339    vector<poly> up(n * 2);
340    rep(i, 0, n) up[i + n] = poly({0 - x[i], 1});
341    per(i, 1, n) up[i] = up[2 * i] * up[2 * i + 1];
342    vector<num> a = eval(deriv(up[1]), x);
343    vector<poly> down(2 * n);
344    rep(i, 0, n) down[i + n] = poly({y[i] * inv(a[i])});
345    per(i, 1, n) down[i] =
346      down[i * 2] * up[i * 2 + 1] + down[i * 2 + 1] * up[i
        * 2];
347    return down[1];
348  }
```

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    vector<int> 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      }
```

13

```
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  };
```

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    for (int i = 0; i < 4 * n; i++) 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     for (int i = 0; i < n; i++){
20       if (!k) st[i][k] = a[i];
```

14

```
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    vector<int> 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     for (int i = 0; i < n; i++) 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     vector<int> p2(n), c2(n);
27     // w is half-length of each string.
28     for (int w = 1; w < n; w <<= 1){
29       for (int i = 0; i < n; i++){
30         p2[i] = (p[i] - w + n) % n;
31       }
32       vector<int> 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     for (int i = 0; i < n; i++){
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 // To add terminal links, use DFS
8 struct Node{
9   vector<int> nxt;
10  int link;
11  bool terminal;
12
13  Node() {
14    nxt.assign(S, -1), link = 0, terminal = 0;
15  }
16 };
17
18 vector<Node> trie(1);
19
20 // add_string returns the terminal vertex.
21 int add_string(string& s){
22   int v = 0;
23   for (auto c : s){
24     int cur = ctoi(c);
25     if (trie[v].nxt[cur] == -1){
26       trie[v].nxt[cur] = sz(trie);
27       trie.emplace_back();
28     }
29     v = trie[v].nxt[cur];
30   }
31   trie[v].terminal = 1;
32   return v;
33 }
34
35 /*
36 Suffix links are compressed.
37 This means that:
38   If vertex v has a child by letter x, then:
39     trie[v].nxt[x] points to that child.
40   If vertex v doesn't have such child, then:
41     trie[v].nxt[x] points to the suffix link of that
   ↪  child
42     if we would actually have it.
43 */
44 void add_links(){
45   queue<int> q;
46   q.push(0);
47   while (!q.empty()){
48     auto v = q.front();
49     int u = trie[v].link;
50     q.pop();
51     for (int i = 0; i < S; i++){
52       int& ch = trie[v].nxt[i];
53       if (ch == -1){
54         ch = v? trie[u].nxt[i] : 0;
55       }
56       else{
57         trie[ch].link = v? trie[u].nxt[i] : 0;
58         q.push(ch);
59       }
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: THE DEFAULT VERSION SURELY WORKS. IF MODIFIED VERSIONS DON'T WORK, TRY TRANSFORMING THEM TO THE DEFAULT ONE BY CHANGING SIGNS.

```
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); // Default:
↪   minimum. For maximum change "min" to "max".
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(); // Default:
↪   decreasing gradient k. For increasing k change the
↪   sign to <=.
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      // Default: minimum. For maximum change the sign to
↪   <=.
29      else r = mid;
30    }
31    return hull[l].f(x);
32  }
```

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()

```
const ll INF = 1e18; // Change the constant!
struct LiChaoTree{
  struct line{
    ll k, b;
    line(){
      k = b = 0;
    };
    line(ll k_, ll b_){
      k = k_, b = b_;
    };
    ll f(ll x){
      return k * x + b;
    };
  };
  int n;
  bool minimum, on_points;
  vector<ll> pts;
  vector<line> t;

  void clear(){
    for (auto& l : t) l.k = 0, l.b = minimum? INF :
↪   -INF;
  }

  LiChaoTree(int n_, bool min_){ // This is a default
↪   constructor for numbers in range [0, n - 1].
    n = n_, minimum = min_, on_points = false;
    t.resize(4 * n);
    clear();
  };

  LiChaoTree(vector<ll> 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.
    pts = pts_, minimum = min_;
    sort(all(pts));
    pts.erase(unique(all(pts)), pts.end());
    on_points = true;
```

```
    n = sz(pts);
    t.resize(4 * n);
    clear();
  };

  void add_line(int v, int l, int r, line nl){
    // Adding on segment [l, r)
    int m = (l + r) / 2;
    ll lval = on_points? pts[l] : l, mval = on_points?
↪   pts[m] : m;
    if ((minimum && nl.f(mval) < t[v].f(mval)) ||
↪   (!minimum && nl.f(mval) > t[v].f(mval))) swap(t[v],
↪   nl);
    if (r - l == 1) return;
    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);
    else add_line(2 * v + 2, m, r, nl);
  }

  ll get(int v, int l, int r, int x){
    int m = (l + r) / 2;
    if (r - l == 1) return t[v].f(on_points? pts[x] :
↪   x);
    else{
      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));
      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));
    }
  }

  void add_line(ll k, ll b){
    add_line(0, 0, n, line(k, b));
  }

  ll get(ll x){
    return get(0, 0, n, on_points? lower_bound(all(pts),
↪   x) - pts.begin() : x);
  }; // Always pass the actual value of x, even if LCT
↪   is on points.
};
```

Persistent Segment Tree

- for RSQ

```
struct Node {
  ll val;
  Node *l, *r;

  Node(ll x) : val(x), l(nullptr), r(nullptr) {}
  Node(Node *ll, Node *rr) {
    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 }
```

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.
```

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!

Dynamic Programming

Sum over Subset DP

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

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

Divide and Conquer DP

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

```
1 vector<ll> 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 }
```