# Columbia University: CU Later Team Reference Document

Kevin Yang, Innokentiy Kaurov, Eric Yuang Shao

May 21th 2024

# Contents

# Templates

## Ken's template

```
#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

```
// 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 form(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

```
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

```
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

```
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

```
// {p, 0} - unique intersection, {p, 1} - infinite, {p, 2} - none
pair<point, int> line_inter(line l1, line l2){
  if (parallel(l1, l2)){
    return {point(), l1 == l2? 1 : 2};
  }
  return {point(
    det(-l1.c, l1.b, -l2.c, l2.b) / det(l1.a, l1.b, l2.a, l2.b),
    det(l1.a, -l1.c, l2.a, -l2.c) / det(l1.a, l1.b, l2.a, l2.b)
  ), 0};
}


// Checks if p lies on ab
bool is_on_seg(point p, point a, point b){
  return abs(vmul(p - a, p - b)) < EPS && smul(p - a, p - b) < EPS;
}

/*
If a unique intersection point between the line segments going from a to b and from c to d exists then it is returned.
If no intersection point exists an empty vector is returned.
If infinitely many exist a vector with 2 elements is returned, containing the endpoints of the common line segment.
*/
vector<point> segment_inter(point a, point b, point c, point d) {
  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);
  if (sgn(oa) * sgn(ob) < 0 && sgn(oc) * sgn(od) < 0) return {(a * ob - b * oa) / (ob - oa)};
  set<point> s;
  if (is_on_seg(a, c, d)) s.insert(a);
  if (is_on_seg(b, c, d)) s.insert(b);
  if (is_on_seg(c, a, b)) s.insert(c);
  if (is_on_seg(d, a, b)) s.insert(d);
  return {all(s)};
}
```

## Distances from a point to line and segment

```
// Distance from p to line ab
ld line_dist(point p, point a, point b){
  return vmul(b - a, p - a) / (b - a).len();
}

// Distance from p to segment ab
ld segment_dist(point p, point a, point b){
  if (a == b) return (p - a).len();
  auto d = (a - b).abs2(), t = min(d, max((ld)0, smul(p - a, b - a)));
  return ((p - a) * d - (b - a) * t).len() / d;
}
```

## Polygon area

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

## Convex hull

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

```
vector<point> convex_hull(vector<point> pts){
  sort(all(pts));
  pts.erase(unique(all(pts)), pts.end());
  vector<point> up, down;
  for (auto p : pts){
    while (sz(up) > 1 && acw(up.end()[-1] - up.end()[-2], p - up.end()[-2])) up.pop_back();
    while (sz(down) > 1 && cw(down.end()[-1] - down.end()[-2], p - down.end()[-2])) down.pop_back();
    up.pb(p), down.pb(p);
  }
  for (int i = sz(up) - 2; i >= 1; i--) down.pb(up[i]);
  return down;
}
```

## Point location in a convex polygon

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

```
void prep_convex_poly(vector<point>& pts){
  rotate(pts.begin(), min_element(all(pts)), pts.end());
}

// 0 - Outside, 1 - Exclusively Inside, 2 - On the Border
int in_convex_poly(point p, vector<point>& pts){
  int n = sz(pts);
  if (!n) return 0;
  if (n <= 2) return is_on_seg(p, pts[0], pts.back());
  int l = 1, r = n - 1;
  while (r - l > 1){
    int mid = (l + r) / 2;
    if (acw(pts[mid] - pts[0], p - pts[0])) l = mid;
    else r = mid;
  }
  if (!in_triangle(p, pts[0], pts[l], pts[l + 1])) return 0;
  if (is_on_seg(p, pts[l], pts[l + 1]) ||
    is_on_seg(p, pts[0], pts.back()) ||
    is_on_seg(p, pts[0], pts[1])
  ) return 2;
  return 1;
}
```

## Point location in a simple polygon

- Complexity: $O(n)$.

```
// 0 - Outside, 1 - Exclusively Inside, 2 - On the Border
int in_simple_poly(point p, vector<point>& pts){
  int n = sz(pts);
  bool res = 0;
  for (int i = 0; i < n; i++){
    auto a = pts[i], b = pts[(i + 1) % n];
    if (is_on_seg(p, a, b)) return 2;
    if (((a.y > p.y) - (b.y > p.y)) * vmul(b - p, a - p) > EPS){
      res ^= 1;
    }
  }
  return res;
}
```

## 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: $O(n)$.

```cpp
void minkowski_rotate(vector<point>& P){
  int pos = 0;
  for (int i = 1; i < sz(P); i++){
    if (abs(P[i].y - P[pos].y) <= EPS){
      if (P[i].x < P[pos].x) pos = i;
    }
    else if (P[i].y < P[pos].y) pos = i;
  }
  rotate(P.begin(), P.begin() + pos, P.end());
}
// 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]);
  vector<point> ans;
  int i = 0, j = 0;
  while (i < sz(P) - 1 || j < sz(Q) - 1){
    ans.pb(P[i] + Q[j]);
    ld curmul;
    if (i == sz(P) - 1) curmul = -1;
    else if (j == sz(Q) - 1) curmul = +1;
    else curmul = vmul(P[i + 1] - P[i], Q[j + 1] - Q[j]);
    if (abs(curmul) < EPS || curmul > 0) i++;
    if (abs(curmul) < EPS || curmul < 0) j++;
  }
  return ans;
}
```

## 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.

```cpp
// Extra functions needed: point operations, smul, vmul
const ld EPS = 1e-9;

int sgn(ld a){
  return (a > EPS) - (a < -EPS);
}
int half(point p){
  return p.y != 0? sgn(p.y) : -sgn(p.x);
}
bool angle_comp(point a, point b){
  int A = half(a), B = half(b);
  return A == B? vmul(a, b) > 0 : A < B;
}
struct ray{
  point p, dp; // origin, direction
  ray(point p_, point dp_){
    p = p_, dp = dp_;
  }
  point isect(ray l){
    return p + dp * (vmul(l.dp, l.p - p) / vmul(l.dp, dp));
  }
  bool operator<(ray l){
    return angle_comp(dp, l.dp);
  }
};
vector<point> half_plane_isect(vector<ray> rays, ld DX = 1e9, ld DY = 1e9){
  // constrain the area to [0, DX] x [0, DY]
  rays.pb({point(0, 0), point(1, 0)});
  rays.pb({point(DX, 0), point(0, 1)});
  rays.pb({point(DX, DY), point(-1, 0)});
  rays.pb({point(0, DY), point(0, -1)});
  sort(all(rays));
  {
    vector<ray> nrays;
    for (auto t : rays){
      if (nrays.empty() || vmul(nrays.back().dp, t.dp) > EPS){
        nrays.pb(t);
        continue;
      }
      if (vmul(t.dp, t.p - nrays.back().p) > 0) nrays.back() = t;
    }
    swap(rays, nrays);
  }
  auto bad = [&] (ray a, ray b, ray c){
    point p1 = a.isect(b), p2 = b.isect(c);
    if (smul(p2 - p1, b.dp) <= EPS){
      if (vmul(a.dp, c.dp) <= 0) return 2;
      return 1;
    }
    return 0;
  };
  #define reduce(t) \
    while (sz(poly) > 1){ \
      int b = bad(poly[sz(poly) - 2], poly.back(), t); \
      if (b == 2) return {}; \
      if (b == 1) poly.pop_back(); \
      else break; \
    }
  deque<ray> poly;
  for (auto t : rays){
    reduce(t);
    poly.pb(t);
  }
  for (;; poly.pop_front()){
    reduce(poly[0]);
    if (!bad(poly.back(), poly[0], poly[1])) break;
  }
  assert(sz(poly) >= 3); // expect nonzero area
  vector<point> poly_points;
  for (int i = 0; i < sz(poly); i++){
    poly_points.pb(poly[i].isect(poly[(i + 1) % sz(poly)]));
  }
  return poly_points;
}
```

# Strings

```cpp
vector<int> prefix_function(string s){
  int n = sz(s);
  vector<int> pi(n);
  for (int i = 1; i < n; i++){
    int k = pi[i - 1];
    while (k > 0 && s[i] != s[k]){
      k = pi[k - 1];
    }
    pi[i] = k + (s[i] == s[k]);
  }
  return pi;
}
// Returns the positions of the first character
vector<int> kmp(string s, string k){
  string st = k + "#" + s;
  vector<int> res;
  auto pi = prefix_function(st);
  for (int i = 0; i < sz(st); i++){
    if (pi[i] == sz(k)){
      res.pb(i - 2 * sz(k));
    }
  }
  return res;
}
vector<int> z_function(string s){
  int n = sz(s);
  vector<int> z(n);
  int l = 0, r = 0;
  for (int i = 1; i < n; i++){
    if (r >= i) z[i] = min(z[i - l], r - i + 1);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]){
```

```
      z[i]++;
    }
    if (i + z[i] - 1 > r){
      l = i, r = i + z[i] - 1;
    }
  }
  return z;
}
```

## Manacher's algorithm

```
/*
Finds longest palindromes centered at each index
even[i] = d --> [i - d, i + d - 1] is a max-palindrome
odd[i] = d  --> [i - d, i + d] is a max-palindrome
*/
pair<vector<int>, vector<int>> manacher(string s) {
  vector<char> t{'^', '#'};
  for (char c : s) t.push_back(c), t.push_back('#');
  t.push_back('$');
  int n = t.size(), r = 0, c = 0;
  vector<int> p(n, 0);
  for (int i = 1; i < n - 1; i++) {
    if (i < r + c) p[i] = min(p[2 * c - i], r + c - i);
    while (t[i + p[i] + 1] == t[i - p[i] - 1]) p[i]++;
    if (i + p[i] > r + c) r = p[i], c = i;
  }
  vector<int> even(sz(s)), odd(sz(s));
  for (int i = 0; i < sz(s); i++){
    even[i] = p[2 * i + 1] / 2, odd[i] = p[2 * i + 2] / 2;
  }
  return {even, odd};
}
```

## 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 $add\_links()$.

```
const int S = 26;

// Function converting char to int.
int ctoi(char c){
  return c - 'a';
}

// To add terminal links, use DFS
struct Node{
  vector<int> nxt;
  int link;
  bool terminal;

  Node() {
    nxt.assign(S, -1), link = 0, terminal = 0;
  }
};

vector<Node> trie(1);

// add_string returns the terminal vertex.
int add_string(string& s){
  int v = 0;
  for (auto c : s){
    int cur = ctoi(c);
    if (trie[v].nxt[cur] == -1){
      trie[v].nxt[cur] = sz(trie);
      trie.emplace_back();
    }
    v = trie[v].nxt[cur];
  }
  trie[v].terminal = 1;
  return v;
}

void add_links(){
  queue<int> q;
  q.push(0);
  while (!q.empty()){
    auto v = q.front();
    int u = trie[v].link;
    q.pop();
    for (int i = 0; i < S; i++){
      int& ch = trie[v].nxt[i];
      if (ch == -1){
        ch = v? trie[u].nxt[i] : 0;
      }
      else{
        trie[ch].link = v? trie[u].nxt[i] : 0;
        q.push(ch);
      }
    }
  }
}

bool is_terminal(int v){
  return trie[v].terminal;
}

int get_link(int v){
  return trie[v].link;
}

int go(int v, char c){
  return trie[v].nxt[ctoi(c)];
}
```

## 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.

```
const int MAXLEN = 1e5 + 20;

struct suffix_automaton{
```

```cpp
  struct state {
    int len, link;
    bool terminal = 0, used = 0;
    map<char, int> next;
  };

  state st[MAXLEN * 2];
  int sz = 0, last;

  suffix_automaton(){
    st[0].len = 0;
    st[0].link = -1;
    sz++;
    last = 0;
  };

  void extend(char c) {
    int cur = sz++;
    st[cur].len = st[last].len + 1;
    int p = last;
    while (p != -1 && !st[p].next.count(c)) {
      st[p].next[c] = cur;
      p = st[p].link;
    }
    if (p == -1) {
      st[cur].link = 0;
    } else {
      int q = st[p].next[c];
      if (st[p].len + 1 == st[q].len) {
        st[cur].link = q;
      } else {
        int clone = sz++;
        st[clone].len = st[p].len + 1;
        st[clone].next = st[q].next;
        st[clone].link = st[q].link;
        while (p != -1 && st[p].next[c] == q) {
            st[p].next[c] = clone;
            p = st[p].link;
        }
        st[q].link = st[cur].link = clone;
      }
    }
    last = cur;
  }

  void mark_terminal(){
    int cur = last;
    while (cur) st[cur].terminal = 1, cur = st[cur].link;
  }
};
/*
Usage:
suffix_automaton sa;
for (int i = 0; i < sz(str); i++) sa.extend(str[i]);
sa.mark_terminal();
*/
```

# Flows

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

```cpp
struct FlowEdge {
  int from, to;
  ll cap, flow = 0;
  FlowEdge(int u, int v, ll cap) : from(u), to(v), cap(cap) {}
};
struct Dinic {
  const ll flow_inf = 1e18;
  vector<FlowEdge> edges;
  vector<vector<int>> adj;
  int n, m = 0;
  int s, t;
  vector<int> level, ptr;
  vector<bool> used;
  queue<int> q;
  Dinic(int n, int s, int t) : n(n), s(s), t(t) {
    adj.resize(n);
    level.resize(n);
    ptr.resize(n);
  }
  void add_edge(int u, int v, ll cap) {
    edges.emplace_back(u, v, cap);
    edges.emplace_back(v, u, 0);
    adj[u].push_back(m);
    adj[v].push_back(m + 1);
    m += 2;
  }
  bool bfs() {
    while (!q.empty()) {
      int v = q.front();
      q.pop();
      for (int id : adj[v]) {
        if (edges[id].cap - edges[id].flow < 1)
          continue;
        if (level[edges[id].to] != -1)
          continue;
        level[edges[id].to] = level[v] + 1;
        q.push(edges[id].to);
      }
    }
    return level[t] != -1;
  }
  ll dfs(int v, ll pushed) {
    if (pushed == 0)
      return 0;
    if (v == t)
      return pushed;
    for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
      int id = adj[v][cid];
      int u = edges[id].to;
      if (level[v] + 1 != level[u] || edges[id].cap - edges[id].flow < 1)
        continue;
      ll tr = dfs(u, min(pushed, edges[id].cap - edges[id].flow));
      if (tr == 0)
        continue;
      edges[id].flow += tr;
      edges[id ^ 1].flow -= tr;
      return tr;
    }
    return 0;
  }
  ll flow() {
    ll f = 0;
    while (true) {
      fill(level.begin(), level.end(), -1);
      level[s] = 0;
      q.push(s);
      if (!bfs())
        break;
      fill(ptr.begin(), ptr.end(), 0);
      while (ll pushed = dfs(s, flow_inf)) {
        f += pushed;
      }
    }
    return f;
  }

  void cut_dfs(int v){
    used[v] = 1;
    for (auto i : adj[v]){
      if (edges[i].flow < edges[i].cap && !used[edges[i].to]){
        cut_dfs(edges[i].to);
      }
    }
  }

  // Assumes that max flow is already calculated
  // true -> vertex is in S, false -> vertex is in T
  vector<bool> min_cut(){
    used = vector<bool>(n);
```

```
    cut_dfs(s);
    return used;
  }
};
// To recover flow through original edges: iterate over even indices in edges.
```

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

```
#include <ext/pb_ds/priority_queue.hpp>
template <typename T, typename C>
class MCMF {
  public:
    static constexpr T eps = (T) 1e-9;

    struct edge {
      int from;
      int to;
      T c;
      T f;
      C cost;
    };

    int n;
    vector<vector<int>> g;
    vector<edge> edges;
    vector<C> d;
    vector<C> pot;
    __gnu_pbds::priority_queue<pair<C, int>> q;
    vector<typename decltype(q)::point_iterator> its;
    vector<int> pe;
    const C INF_C = numeric_limits<C>::max() / 2;

    explicit MCMF(int n_) : n(n_), g(n), d(n), pot(n, 0), its(n), pe(n) {}

    int add(int from, int to, T forward_cap, C edge_cost, T backward_cap = 0) {
      assert(0 <= from && from < n && 0 <= to && to < n);
      assert(forward_cap >= 0 && backward_cap >= 0);
      int id = static_cast<int>(edges.size());
      g[from].push_back(id);
      edges.push_back({from, to, forward_cap, 0, edge_cost});
      g[to].push_back(id + 1);
      edges.push_back({to, from, backward_cap, 0, -edge_cost});
      return id;
    }

    void expath(int st) {
      fill(d.begin(), d.end(), INF_C);
      q.clear();
      fill(its.begin(), its.end(), q.end());
      its[st] = q.push({pot[st], st});
      d[st] = 0;
      while (!q.empty()) {
        int i = q.top().second;
        q.pop();
        its[i] = q.end();
        for (int id : g[i]) {
          const edge &e = edges[id];
          int j = e.to;
          if (e.c - e.f > eps && d[i] + e.cost < d[j]) {
            d[j] = d[i] + e.cost;
            pe[j] = id;
            if (its[j] == q.end()) {
              its[j] = q.push({pot[j] - d[j], j});
            } 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;
                }
              }
            }
          }
        }
```

```cpp
        }
      }
      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.

```cpp
int INF = 1e9; // constant greater than any number in the matrix
vector<int> u(n+1), v(m+1), p(m+1), way(m+1);
for (int i=1; i<=n; ++i) {
  p[0] = i;
  int j0 = 0;
  vector<int> minv (m+1, INF);
  vector<bool> used (m+1, false);
  do {
    used[j0] = true;
    int i0 = p[j0],  delta = INF,  j1;
    for (int j=1; j<=m; ++j)
      if (!used[j]) {
        int cur = A[i0][j]-u[i0]-v[j];
        if (cur < minv[j])
          minv[j] = cur,  way[j] = j0;
        if (minv[j] < delta)
          delta = minv[j],  j1 = j;
      }
    for (int j=0; j<=m; ++j)
      if (used[j])
        u[p[j]] += delta,  v[j] -= delta;
      else
        minv[j] -= delta;
    j0 = j1;
  } while (p[j0] != 0);
  do {
    int j1 = way[j0];
    p[j0] = p[j1];
    j0 = j1;
  } while (j0);
}
vector<int> ans (n+1); // ans[i] stores the column selected for row i
for (int j=1; j<=m; ++j)
  ans[p[j]] = j;
int cost = -v[0]; // the total cost of the matching
```

## Dijkstra's Algorithm

```cpp
priority_queue<pair<ll, ll>, vector<pair<ll, ll>>, greater<pair<ll, ll>>> q;
dist[start] = 0;
q.push({0, start});
while (!q.empty()){
  auto [d, v] = q.top();
  q.pop();
  if (d != dist[v]) continue;
  for (auto [u, w] : g[v]){
    if (dist[u] > dist[v] + w){
      dist[u] = dist[v] + w;
      q.push({dist[u], u});
    }
  }
}
```

## Eulerian Cycle DFS

```cpp
void dfs(int v){
  while (!g[v].empty()){
```

```cpp
    int u = g[v].back();
    g[v].pop_back();
    dfs(u);
    ans.pb(v);
  }
}
```

## SCC and 2-SAT

```cpp
void scc(vector<vector<int>>& g, int* idx) {
  int n = g.size(), ct = 0;
  int out[n];
  vector<int> ginv[n];
  memset(out, -1, sizeof out);
  memset(idx, -1, n * sizeof(int));
  function<void(int)> dfs = [&](int cur) {
    out[cur] = INT_MAX;
    for(int v : g[cur]) {
      ginv[v].push_back(cur);
      if(out[v] == -1) dfs(v);
    }
    ct++; out[cur] = ct;
  };
  vector<int> order;
  for(int i = 0; i < n; i++) {
    order.push_back(i);
    if(out[i] == -1) dfs(i);
  }
  sort(order.begin(), order.end(), [&](int& u, int& v) {
    return out[u] > out[v];
  });
  ct = 0;
  stack<int> s;
  auto dfs2 = [&](int start) {
    s.push(start);
    while(!s.empty()) {
      int cur = s.top();
      s.pop();
      idx[cur] = ct;
      for(int v : ginv[cur])
        if(idx[v] == -1) s.push(v);
    }
  };
  for(int v : order) {
    if(idx[v] == -1) {
      dfs2(v);
      ct++;
    }
  }
}

// 0 => impossible, 1 => possible
pair<int,vector<int>> sat2(int n, vector<pair<int,int>>& clauses) {
  vector<int> ans(n);
  vector<vector<int>> g(2*n + 1);
  for(auto [x, y] : clauses) {
    x = x < 0 ? -x + n : x;
    y = y < 0 ? -y + n : y;
    int nx = x <= n ? x + n : x - n;
    int ny = y <= n ? y + n : y - n;
    g[nx].push_back(y);
    g[ny].push_back(x);
  }
  int idx[2*n + 1];
  scc(g, idx);
  for(int i = 1; i <= n; i++) {
    if(idx[i] == idx[i + n]) return {0, {}};
    ans[i - 1] = idx[i + n] < idx[i];
  }
  return {1, ans};
}
```

## Finding Bridges

```cpp
/*
Bridges.
Results are stored in a map "is_bridge".
For each connected component, call "dfs(starting vertex, starting vertex)".
*/
const int N = 2e5 + 10; // Careful with the constant!

vector<int> g[N];
int tin[N], fup[N], timer;
map<pair<int, int>, bool> is_bridge;

void dfs(int v, int p){
  tin[v] = ++timer;
  fup[v] = tin[v];
  for (auto u : g[v]){
    if (!tin[u]){
      dfs(u, v);
      if (fup[u] > tin[v]){
        is_bridge[{u, v}] = is_bridge[{v, u}] = true;
      }
      fup[v] = min(fup[v], fup[u]);
    }
    else{
      if (u != p) fup[v] = min(fup[v], tin[u]);
    }
  }
}
```

## 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);
```

```cpp
      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
  }
};
```

## Biconnected Components and Block-Cut Tree

- Biconnected components are the ones that have no articulation 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: $O(n)$.

```cpp
// Usage: pass in adjacency list in 0-based indexation.
// Return: adjacency list of block-cut tree (nodes 0...n-1 represent original nodes, the rest are component nodes).
vector<vector<int>> biconnected_components(vector<vector<int>> g) {
    int n = sz(g);
    vector<vector<int>> comps;
    vector<int> stk, num(n), low(n);
  int timer = 0;
    // Finds the biconnected components
    function<void(int, int)> dfs = [&](int v, int p) {
        num[v] = low[v] = ++timer;
        stk.pb(v);
        for (int son : g[v]) {
            if (son == p) continue;
            if (num[son]) low[v] = min(low[v], num[son]);
      else{
                dfs(son, v);
                low[v] = min(low[v], low[son]);
                if (low[son] >= num[v]){
                    comps.pb({v});
                    while (comps.back().back() != son){
                        comps.back().pb(stk.back());
                        stk.pop_back();
                    }
                }
            }
        }
    };
    dfs(0, -1);
    // Build the block-cut tree
    auto build_tree = [&]() {
        vector<vector<int>> t(n);
        for (auto &comp : comps){
            t.push_back({});
            for (int u : comp){
                t.back().pb(u);
        t[u].pb(sz(t) - 1);
      }
        }
        return t;
    };
    return 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;
  }
};

```

```cpp
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

- $O(\max(\log a, \log b))$
- 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
ll euclid(ll a, ll b, ll &x, ll &y) {
  if (!b) return x = 1, y = 0, a;
  ll d = euclid(b, a % b, y, x);
  return y -= a/b * x, d;
}
```

## 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 $0 \le x < \mathrm{lcm}(m, n)$.
- Assumes $mn < 2^{62}$.
- $O(\max(\log m, \log n))$

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

## 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>
```

```cpp
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};
}
```

## 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 $p$

```cpp
ll sqrt(ll a, ll p) {
  a %= p; if (a < 0) a += p;
  if (a == 0) return 0;
  assert(pow(a, (p-1)/2, p) == 1); // else no solution
  if (p % 4 == 3) return pow(a, (p+1)/4, p);
  // a^(n+3)/8 or 2^(n+3)/8 * 2^(n-1)/4 works if p % 8 == 5
  ll s = p - 1, n = 2;
  int r = 0, m;
  while (s % 2 == 0)
    ++r, s /= 2;
  /// find a non-square mod p
  while (pow(n, (p - 1) / 2, p) != p - 1) ++n;
  ll x = pow(a, (s + 1) / 2, p);
  ll b = pow(a, s, p), g = pow(n, s, p);
  for (;; r = m) {
    ll t = b;
    for (m = 0; m < r && t != 1; ++m)
      t = t * t % p;
    if (m == 0) return x;
    ll gs = pow(g, 1LL << (r - m - 1), p);
    g = gs * gs % p;
    x = x * gs % p;
    b = b * g % p;
  }
}
```

## 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: $O(N^2)$

```cpp
vector<ll> berlekamp_massey(vector<ll> s) {
  int n = sz(s), l = 0, m = 1;
  vector<ll> b(n), c(n);
  ll ldd = b[0] = c[0] = 1;
  for (int i = 0; i < n; i++, m++) {
    ll d = s[i];
    for (int j = 1; j <= l; j++) d = (d + c[j] * s[i - j]) % MOD;
    if (d == 0) continue;
    vector<ll> temp = c;
    ll coef = d * power(ldd, MOD - 2) % MOD;
    for (int j = m; j < n; j++){
      c[j] = (c[j] + MOD - coef * b[j - m]) % MOD;
      if (c[j] < 0) c[j] += MOD;
    }
    if (2 * l <= i) {
      l = i + 1 - l;
      b = temp;
      ldd = d;
      m = 0;
```

```
    }
  }
  c.resize(l + 1);
  c.erase(c.begin());
  for (ll &x : c)
    x = (MOD - x) % MOD;
  return c;
}
```

## 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: $O(n^2 \log k)$

```
vector<ll> poly_mult_mod(vector<ll> p, vector<ll> q, vector<ll>& c){
  vector<ll> ans(sz(p) + sz(q) - 1);
  for (int i = 0; i < sz(p); i++){
    for (int j = 0; j < sz(q); j++){
      ans[i + j] = (ans[i + j] + p[i] * q[j]) % MOD;
    }
  }
  int n = sz(ans), m = sz(c);
  for (int i = n - 1; i >= m; i--){
    for (int j = 0; j < m; j++){
      ans[i - 1 - j] = (ans[i - 1 - j] + c[j] * ans[i]) % MOD;
    }
  }
  ans.resize(m);
  return ans;
}

ll calc_kth(vector<ll> s, vector<ll> c, ll k){
  assert(sz(s) >= sz(c)); // size of s can be greater than c, but not less
  if (k < sz(s)) return s[k];
  vector<ll> res{1};
  for (vector<ll> poly = {0, 1}; k; poly = poly_mult_mod(poly, poly, c), k >>= 1){
    if (k & 1) res = poly_mult_mod(res, poly, c);
  }
  ll ans = 0;
  for (int i = 0; i < min(sz(res), sz(c)); i++) ans = (ans + s[i] * res[i]) % MOD;
  return ans;
}
```

## Partition Function

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

```
int partition(int n) {
  int dp[n + 1];
  dp[0] = 1;
  for (int i = 1; i <= n; i++) {
    dp[i] = 0;
    for (int j = 1, r = 1; i - (3 * j * j - j) / 2 >= 0; ++j, r *= -1) {
      dp[i] += dp[i - (3 * j * j - j) / 2] * r;
      if (i - (3 * j * j + j) / 2 >= 0) dp[i] += dp[i - (3 * j * j + j) / 2] * r;
    }
  }
  return dp[n];
}
```

## NTT

```
void ntt(vector<ll>& a, int f) {
  int n = int(a.size());
  vector<ll> w(n);
  vector<int> rev(n);
  for (int i = 0; i < n; i++) rev[i] = (rev[i / 2] / 2) | ((i & 1) * (n / 2));
  for (int i = 0; i < n; i++) {
    if (i < rev[i]) swap(a[i], a[rev[i]]);
  }
  ll wn = power(f ? (MOD + 1) / 3 : 3, (MOD - 1) / n);
  w[0] = 1;
  for (int i = 1; i < n; i++) w[i] = w[i - 1] * wn % MOD;
  for (int mid = 1; mid < n; mid *= 2) {
    for (int i = 0; i < n; i += 2 * mid) {
      for (int j = 0; j < mid; j++) {
        ll x = a[i + j], y = a[i + j + mid] * w[n / (2 * mid) * j] % MOD;
        a[i + j] = (x + y) % MOD, a[i + j + mid] = (x + MOD - y) % MOD;
      }
    }
  }
  if (f) {
    ll iv = power(n, MOD - 2);
    for (auto& x : a) x = x * iv % MOD;
  }
}
vector<ll> mul(vector<ll> a, vector<ll> b) {
  int n = 1, m = (int)a.size() + (int)b.size() - 1;
  while (n < m) n *= 2;
  a.resize(n), b.resize(n);
  ntt(a, 0), ntt(b, 0); // if squaring, you can save one NTT here
  for (int i = 0; i < n; i++) a[i] = a[i] * b[i] % MOD;
  ntt(a, 1);
  a.resize(m);
  return a;
}
```

## FFT

```
const ld PI = acosl(-1);
auto mul = [&](const vector<ld>& aa, const vector<ld>& bb) {
  int n = (int)aa.size(), m = (int)bb.size(), bit = 1;
  while ((1 << bit) < n + m - 1) bit++;
  int len = 1 << bit;
  vector<complex<ld>> a(len), b(len);
  vector<int> rev(len);
  for (int i = 0; i < n; i++) a[i].real(aa[i]);
  for (int i = 0; i < m; i++) b[i].real(bb[i]);
  for (int i = 0; i < len; i++) rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (bit - 1));
  auto fft = [&](vector<complex<ld>>& p, int inv) {
    for (int i = 0; i < len; i++)
      if (i < rev[i]) swap(p[i], p[rev[i]]);
    for (int mid = 1; mid < len; mid *= 2) {
      auto w1 = complex<ld>(cos(PI / mid), (inv ? -1 : 1) * sin(PI / mid));
      for (int i = 0; i < len; i += mid * 2) {
        auto wk = complex<ld>(1, 0);
        for (int j = 0; j < mid; j++, wk = wk * w1) {
          auto x = p[i + j], y = wk * p[i + j + mid];
          p[i + j] = x + y, p[i + j + mid] = x - y;
        }
      }
    }
    if (inv == 1) {
      for (int i = 0; i < len; i++) p[i].real(p[i].real() / len);
    }
  };
  fft(a, 0), fft(b, 0);
  for (int i = 0; i < len; i++) a[i] = a[i] * b[i];
  fft(a, 1);
  a.resize(n + m - 1);
```

```
  vector<ld> res(n + m - 1);
  for (int i = 0; i < n + m - 1; i++) res[i] = a[i].real();
  return res;
};
```

## 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 $O(n \log^2 n)$

```
// use #define FFT 1 to use FFT instead of NTT (default)
// Examples:
// poly a(n+1); // constructs degree n poly
// a[0].v = 10; // assigns constant term a_0 = 10
// poly b = exp(a);
// poly is vector<num>
// for NTT, num stores just one int named v
// for FFT, num stores two doubles named x (real), y (imag)

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

namespace fft {
#if FFT
// FFT
using dbl = double;
struct num {
  dbl x, y;
  num(dbl x_ = 0, dbl y_ = 0): x(x_), y(y_) {}
};
inline num operator+(num a, num b) {
  return num(a.x + b.x, a.y + b.y);
}
inline num operator-(num a, num b) {
  return num(a.x - b.x, a.y - b.y);
}
inline num operator*(num a, num b) {
  return num(a.x * b.x - a.y * b.y, a.x * b.y + a.y * b.x);
}
inline num conj(num a) { return num(a.x, -a.y); }
inline num inv(num a) {
  dbl n = (a.x * a.x + a.y * a.y);
  return num(a.x / n, -a.y / n);
}

#else
// NTT
const int mod = 998244353, g = 3;
// For p < 2^30 there is also (5 << 25, 3), (7 << 26, 3),
// (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)
    for (int i = 0; i < n; i += 2 * k) rep(j, 0, k) {
        num t = rt[j + k] * a[i + j + k];
        a[i + j + k] = a[i + j] - t;
        a[i + j] = a[i + j] + t;
      }
}
// Complex/NTT
vn multiply(vn a, vn b) {
  int s = sz(a) + sz(b) - 1;
  if (s <= 0) return {};
  int L = s > 1 ? 32 - __builtin_clz(s - 1) : 0, n = 1 << L;
  a.resize(n), b.resize(n);
  fft(a, n);
  fft(b, n);
  num d = inv(num(n));
  rep(i, 0, n) a[i] = a[i] * b[i] * d;
  reverse(a.begin() + 1, a.end());
  fft(a, n);
  a.resize(s);
  return a;
}
// Complex/NTT power-series inverse
// Doubles b as b[:n] = (2 - a[:n] * b[:n/2]) * b[:n/2]
vn inverse(const vn& a) {
  if (a.empty()) return {};
  vn b({inv(a[0])});
  b.reserve(2 * a.size());
  while (sz(b) < sz(a)) {
    int n = 2 * sz(b);
    b.resize(2 * n, 0);
    if (sz(fa) < 2 * n) fa.resize(2 * n);
    fill(fa.begin(), fa.begin() + 2 * n, 0);
    copy(a.begin(), a.begin() + min(n, sz(a)), fa.begin());
    fft(b, 2 * n);
    fft(fa, 2 * n);
    num d = inv(num(2 * n));
    rep(i, 0, 2 * n) b[i] = b[i] * (2 - fa[i] * b[i]) * d;
    reverse(b.begin() + 1, b.end());
    fft(b, 2 * n);
    b.resize(n);
  }
```

```cpp
  b.resize(a.size());
  return b;
}
#if FFT
// Double multiply (num = complex)
using vd = vector<double>;
vd multiply(const vd& a, const vd& b) {
  int s = sz(a) + sz(b) - 1;
  if (s <= 0) return {};
  int L = s > 1 ? 32 - __builtin_clz(s - 1) : 0, n = 1 << L;
  if (sz(fa) < n) fa.resize(n);
  if (sz(fb) < n) fb.resize(n);
  fill(fa.begin(), fa.begin() + n, 0);
  rep(i, 0, sz(a)) fa[i].x = a[i];
  rep(i, 0, sz(b)) fa[i].y = b[i];
  fft(fa, n);
  trav(x, fa) x = x * x;
  rep(i, 0, n) fb[i] = fa[(n - i) & (n - 1)] - conj(fa[i]);
  fft(fb, n);
  vd r(s);
  rep(i, 0, s) r[i] = fb[i].y / (4 * n);
  return r;
}
// Integer multiply mod m (num = complex)
vi multiply_mod(const vi& a, const vi& b, int m) {
  int s = sz(a) + sz(b) - 1;
  if (s <= 0) return {};
  int L = s > 1 ? 32 - __builtin_clz(s - 1) : 0, n = 1 << L;
  if (sz(fa) < n) fa.resize(n);
  if (sz(fb) < n) fb.resize(n);
  rep(i, 0, sz(a)) fa[i] =
    num(a[i] & ((1 << 15) - 1), a[i] >> 15);
  fill(fa.begin() + sz(a), fa.begin() + n, 0);
  rep(i, 0, sz(b)) fb[i] =
    num(b[i] & ((1 << 15) - 1), b[i] >> 15);
  fill(fb.begin() + sz(b), fb.begin() + n, 0);
  fft(fa, n);
  fft(fb, n);
  double r0 = 0.5 / n; // 1/2n
  rep(i, 0, n / 2 + 1) {
    int j = (n - i) & (n - 1);
    num g0 = (fb[i] + conj(fb[j])) * r0;
    num g1 = (fb[i] - conj(fb[j])) * r0;
    swap(g1.x, g1.y);
    g1.y *= -1;
    if (j != i) {
      swap(fa[j], fa[i]);
      fb[j] = fa[j] * g1;
      fa[j] = fa[j] * g0;
    }
    fb[i] = fa[i] * conj(g1);
    fa[i] = fa[i] * conj(g0);
  }
  fft(fa, n);
  fft(fb, n);
  vi r(s);
  rep(i, 0, s) r[i] =
    int((ll(fa[i].x + 0.5) + (ll(fa[i].y + 0.5) % m << 15) +
         (ll(fb[i].x + 0.5) % m << 15) +
         (ll(fb[i].y + 0.5) % m << 30)) %
      m);
  return r;
}
#endif
} // namespace fft
// For multiply_mod, use num = modnum, poly = vector<num>
using fft::num;
using poly = fft::vn;
using fft::multiply;
using fft::inverse;

poly& operator+=(poly& a, const poly& b) {
  if (sz(a) < sz(b)) a.resize(b.size());
  rep(i, 0, sz(b)) a[i] = a[i] + b[i];
  return a;
}
poly operator+(const poly& a, const poly& b) {
  poly r = a;
  r += b;
  return r;
}
poly& operator-=(poly& a, const poly& b) {
  if (sz(a) < sz(b)) a.resize(b.size());
  rep(i, 0, sz(b)) a[i] = a[i] - b[i];
  return a;
}
poly operator-(const poly& a, const poly& b) {
  poly r = a;
  r -= b;
  return r;
}
poly operator*(const poly& a, const poly& b) {
  return multiply(a, b);
}
poly& operator*=(poly& a, const poly& b) { return a = a * b; }

poly& operator*=(poly& a, const num& b) { // Optional
  trav(x, a) x = x * b;
  return a;
}
poly operator*(const poly& a, const num& b) {
  poly r = a;
  r *= b;
  return r;
}
// Polynomial floor division; no leading 0's please
poly operator/(poly a, poly b) {
  if (sz(a) < sz(b)) return {};
  int s = sz(a) - sz(b) + 1;
  reverse(a.begin(), a.end());
  reverse(b.begin(), b.end());
  a.resize(s);
  b.resize(s);
  a = a * inverse(move(b));
  a.resize(s);
  reverse(a.begin(), a.end());
  return a;
}
poly& operator/=(poly& a, const poly& b) { return a = a / b; }
poly& operator%=(poly& a, const poly& b) {
  if (sz(a) >= sz(b)) {
    poly c = (a / b) * b;
    a.resize(sz(b) - 1);
    rep(i, 0, sz(a)) a[i] = a[i] - c[i];
  }
  return a;
}
poly operator%(const poly& a, const poly& b) {
  poly r = a;
  r %= b;
  return r;
}
// Log/exp/pow
poly deriv(const poly& a) {
  if (a.empty()) return {};
  poly b(sz(a) - 1);
  rep(i, 1, sz(a)) b[i - 1] = a[i] * i;
  return b;
}
poly integ(const poly& a) {
  poly b(sz(a) + 1);
  b[1] = 1; // mod p
  rep(i, 2, sz(b)) b[i] =
    b[fft::mod % i] * (-fft::mod / i); // mod p
  rep(i, 1, sz(b)) b[i] = a[i - 1] * b[i]; // mod p
  //rep(i,1,sz(b)) b[i]=a[i-1]*inv(num(i)); // else
  return b;
}
poly log(const poly& a) { // MUST have a[0] == 1
  poly b = integ(deriv(a) * inverse(a));
  b.resize(a.size());
  return b;
}
poly exp(const poly& a) { // MUST have a[0] == 0
```

```cpp
  poly b(1, num(1));
  if (a.empty()) return b;
  while (sz(b) < sz(a)) {
    int n = min(sz(b) * 2, sz(a));
    b.resize(n);
    poly v = poly(a.begin(), a.begin() + n) - log(b);
    v[0] = v[0] + num(1);
    b *= v;
    b.resize(n);
  }
  return b;
}
poly pow(const poly& a, int m) { // m >= 0
  poly b(a.size());
  if (!m) {
    b[0] = 1;
    return b;
  }
  int p = 0;
  while (p < sz(a) && a[p].v == 0) ++p;
  if (1ll * m * p >= sz(a)) return b;
  num mu = pow(a[p], m), di = inv(a[p]);
  poly c(sz(a) - m * p);
  rep(i, 0, sz(c)) c[i] = a[i + p] * di;
  c = log(c);
  trav(v, c) v = v * m;
  c = exp(c);
  rep(i, 0, sz(c)) b[i + m * p] = c[i] * mu;
  return b;
}
// Multipoint evaluation/interpolation

vector<num> eval(const poly& a, const vector<num>& x) {
  int n = sz(x);
  if (!n) return {};
  vector<poly> up(2 * n);
  rep(i, 0, n) up[i + n] = poly({0 - x[i], 1});
  per(i, 1, n) up[i] = up[2 * i] * up[2 * i + 1];
  vector<poly> down(2 * n);
  down[1] = a % up[1];
  rep(i, 2, 2 * n) down[i] = down[i / 2] % up[i];
  vector<num> y(n);
  rep(i, 0, n) y[i] = down[i + n][0];
  return y;
}

poly interp(const vector<num>& x, const vector<num>& y) {
  int n = sz(x);
  assert(n);
  vector<poly> up(n * 2);
  rep(i, 0, n) up[i + n] = poly({0 - x[i], 1});
  per(i, 1, n) up[i] = up[2 * i] * up[2 * i + 1];
  vector<num> a = eval(deriv(up[1]), x);
  vector<poly> down(2 * n);
  rep(i, 0, n) down[i + n] = poly({y[i] * inv(a[i])});
  per(i, 1, n) down[i] =
    down[i * 2] * up[i * 2 + 1] + down[i * 2 + 1] * up[i * 2];
  return down[1];
}
```

## Simplex method for linear programs

- Maximize $c^T x$ subject to $Ax \leq b$, $x \geq 0$.
- 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).

```cpp
typedef double T; // might be much slower with long doubles
typedef vector<T> vd;
typedef vector<vd> vvd;
const T eps = 1e-8, inf = 1/.0;
#define MP make_pair
#define ltj(X) if(s == -1 || MP(X[j],N[j]) < MP(X[s],N[s])) s=j
#define rep(i, a, b) for(int i = a; i < (b); ++i)

struct LPSolver {
  int m, n;
  vector<int> N,B;
  vvd D;
  LPSolver(const vvd& A, const vd& b, const vd& c) : m(sz(b)), n(sz(c)), N(n+1), B(m), D(m+2, vd(n+2)){
    rep(i,0,m) rep(j,0,n) D[i][j] = A[i][j];
    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]; }
    N[n] = -1; D[m+1][n] = 1;
  };
  void pivot(int r, int s){
    T *a = D[r].data(), inv = 1 / a[s];
    rep(i,0,m+2) if (i != r && abs(D[i][s]) > eps) {
      T *b = D[i].data(), inv2 = b[s] * inv;
      rep(j,0,n+2) b[j] -= a[j] * inv2;
      b[s] = a[s] * inv2;
    }
    rep(j,0,n+2) if (j != s) D[r][j] *= inv;
    rep(i,0,m+2) if (i != r) D[i][s] *= -inv;
    D[r][s] = inv;
    swap(B[r], N[s]);
  }
  bool simplex(int phase){
    int x = m + phase - 1;
    for (;;) {
      int s = -1;
      rep(j,0,n+1) if (N[j] != -phase) ltj(D[x]); if (D[x][s] >= -eps) return true;
      int r = -1;
      rep(i,0,m) {
        if (D[i][s] <= eps) continue;
        if (r == -1 || MP(D[i][n+1] / D[i][s], B[i]) < MP(D[r][n+1] / D[r][s], B[r])) r = i;
      }
      if (r == -1) return false;
      pivot(r, s);
    }
  }
  T solve(vd &x){
    int r = 0;
    rep(i,1,m) if (D[i][n+1] < D[r][n+1]) r = i;
    if (D[r][n+1] < -eps) {
      pivot(r, n);
      if (!simplex(2) || D[m+1][n+1] < -eps) return -inf;
      rep(i,0,m) if (B[i] == -1) {
        int s = 0;
        rep(j,1,n+1) ltj(D[i]);
        pivot(i, s);
      }
    }
    bool ok = simplex(1); x = vd(n);
    rep(i,0,m) if (B[i] < n) x[B[i]] = D[i][n+1];
    return ok ? D[m][n+1] : inf;
  }
};
```

## 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 $R = answer$.

```cpp

// Example matroid
struct GraphicMatroid{
  vector<pair<int, int>> e;
  int n;
  DSU dsu;

  GraphicMatroid(vector<pair<int, int>> edges, int vertices){
    e = edges, n = vertices;
    dsu = DSU(n);
  };
  bool check(int idx){
    return !dsu.same(e[idx].fi, e[idx].se);
  }
  void add(int idx){
    dsu.unite(e[idx].fi, e[idx].se);
  }
  void clear(){
    dsu = DSU(n);
  }
};

template <class M1, class M2> struct MatroidIsect {
    int n;
    vector<char> iset;
    M1 m1; M2 m2;
    MatroidIsect(M1 m1, M2 m2, int n) : n(n), iset(n + 1), m1(m1), m2(m2) {}
    vector<int> solve() {
        for (int i = 0; i < n; i++) if (m1.check(i) && m2.check(i))
            iset[i] = true, m1.add(i), m2.add(i);
        while (augment());
        vector<int> ans;
        for (int i = 0; i < n; i++) if (iset[i]) ans.push_back(i);
        return ans;
    }
    bool augment() {
        vector<int> frm(n, -1);
        queue<int> q({n}); // starts at dummy node
        auto fwdE = [&](int a) {
            vector<int> ans;
            m1.clear();
            for (int v = 0; v < n; v++) if (iset[v] && v != a) m1.add(v);
            for (int b = 0; b < n; b++) if (!iset[b] && frm[b] == -1 && m1.check(b))
                ans.push_back(b), frm[b] = a;
            return ans;
        };
        auto backE = [&](int b) {
            m2.clear();
            for (int cas = 0; cas < 2; cas++) for (int v = 0; v < n; v++){
                if ((v == b || iset[v]) && (frm[v] == -1) == cas) {
                    if (!m2.check(v))
                        return cas ? q.push(v), frm[v] = b, v : -1;
                    m2.add(v);
                }
      }
            return n;
        };
        while (!q.empty()) {
            int a = q.front(), c; q.pop();
            for (int b : fwdE(a))
                while((c = backE(b)) >= 0) if (c == n) {
                    while (b != n) iset[b] ^= 1, b = frm[b];
                    return true;
                }
        }
        return false;
    }
};

/*
Usage:
MatroidIsect<GraphicMatroid, ColorfulMatroid> solver(matroid1, matroid2, n);
vector<int> answer = solver.solve();
*/
```

# Data Structures

## Fenwick Tree

```cpp
ll sum(int r) {
  ll ret = 0;
  for (; r >= 0; r = (r & r + 1) - 1) ret += bit[r];
  return ret;
}
void add(int idx, ll delta) {
  for (; idx < n; idx |= idx + 1) bit[idx] += delta;
}
```

## Lazy Propagation SegTree

```cpp
// Clear: clear() or build()
const int N = 2e5 + 10; // Change the constant!
template<typename T>
struct LazySegTree{
  T t[4 * N];
  T lazy[4 * N];
  int n;

  // Change these functions, default return, and lazy mark.
  T default_return = 0, lazy_mark = numeric_limits<T>::min();
  // Lazy mark is how the algorithm will identify that no propagation is needed.
  function<T(T, T)> f = [&] (T a, T b){
    return a + b;
  };
  // f_on_seg calculates the function f, knowing the lazy value on segment,
  // segment's size and the previous value.
  // The default is segment modification for RSQ. For increments change to:
  //     return cur_seg_val + seg_size * lazy_val;
  // For RMQ.   Modification: return lazy_val;   Increments: return cur_seg_val + lazy_val;
  function<T(T, int, T)> f_on_seg = [&] (T cur_seg_val, int seg_size, T lazy_val){
    return seg_size * lazy_val;
  };
  // upd_lazy updates the value to be propagated to child segments.
```

```
  // Default: modification. For increments change to:
  //     lazy[v] = (lazy[v] == lazy_mark? val : lazy[v] +
  ↪  val);
  function<void(int, T)> upd_lazy = [&] (int v, T val){
    lazy[v] = val;
  };
  // Tip: for "get element on single index" queries, use max()
  ↪  on segment: no overflows.

  LazySegTree(int n_) : n(n_) {
    clear(n);
  }

  void build(int v, int tl, int tr, vector<T>& a){
    if (tl == tr) {
      t[v] = a[tl];
      return;
    }
    int tm = (tl + tr) / 2;
    // left child: [tl, tm]
    // right child: [tm + 1, tr]
    build(2 * v + 1, tl, tm, a);
    build(2 * v + 2, tm + 1, tr, a);
    t[v] = f(t[2 * v + 1], t[2 * v + 2]);
  }

  LazySegTree(vector<T>& a){
    build(a);
  }

  void push(int v, int tl, int tr){
    if (lazy[v] == lazy_mark) return;
    int tm = (tl + tr) / 2;
    t[2 * v + 1] = f_on_seg(t[2 * v + 1], tm - tl + 1,
  ↪  lazy[v]);
    t[2 * v + 2] = f_on_seg(t[2 * v + 2], tr - tm, lazy[v]);
    upd_lazy(2 * v + 1, lazy[v]), upd_lazy(2 * v + 2,
  ↪  lazy[v]);
    lazy[v] = lazy_mark;
  }

  void modify(int v, int tl, int tr, int l, int r, T val){
    if (l > r) return;
    if (tl == l && tr == r){
      t[v] = f_on_seg(t[v], tr - tl + 1, val);
      upd_lazy(v, val);
      return;
    }
    push(v, tl, tr);
    int tm = (tl + tr) / 2;
    modify(2 * v + 1, tl, tm, l, min(r, tm), val);
    modify(2 * v + 2, tm + 1, tr, max(l, tm + 1), r, val);
    t[v] = f(t[2 * v + 1], t[2 * v + 2]);
  }

  T query(int v, int tl, int tr, int l, int r) {
    if (l > r) return default_return;
    if (tl == l && tr == r) return t[v];
    push(v, tl, tr);
    int tm = (tl + tr) / 2;
    return f(
      query(2 * v + 1, tl, tm, l, min(r, tm)),
      query(2 * v + 2, tm + 1, tr, max(l, tm + 1), r)
    );
  }

  void modify(int l, int r, T val){
    modify(0, 0, n - 1, l, r, val);
  }

  T query(int l, int r){
    return query(0, 0, n - 1, l, r);
  }

  T get(int pos){
    return query(pos, pos);
  }

  // Change clear() function to t.clear() if using
  ↪  unordered_map for SegTree!!!
  void clear(int n_){
    n = n_;
    for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] =
  ↪  lazy_mark;
  }

  void build(vector<T>& a){
    n = sz(a);
    clear(n);
    build(0, 0, n - 1, a);
  }
};
```

## Sparse Table

```
const int N = 2e5 + 10, LOG = 20; // Change the constant!
template<typename T>
struct SparseTable{
int lg[N];
T st[N][LOG];
int n;

// Change this function
function<T(T, T)> f = [&] (T a, T b){
  return min(a, b);
};

void build(vector<T>& a){
  n = sz(a);
  lg[1] = 0;
  for (int i = 2; i <= n; i++) lg[i] = lg[i / 2] + 1;

  for (int k = 0; k < LOG; k++){
    for (int i = 0; i < n; i++){
      if (!k) st[i][k] = a[i];
      else st[i][k] = f(st[i][k - 1], st[min(n - 1, i + (1 <<
  ↪  (k - 1)))][k - 1]);
    }
  }
}

T query(int l, int r){
  int sz = r - l + 1;
  return f(st[l][lg[sz]], st[r - (1 << lg[sz]) + 1][lg[sz]]);
}
};
```

## Suffix Array and LCP array

- (uses SparseTable above)

```
struct SuffixArray{
  vector<int> p, c, h;
  SparseTable<int> st;
  /*
  In the end, array c gives the position of each suffix in p
  using 1-based indexation!
  */

  SuffixArray() {}

  SuffixArray(string s){
    buildArray(s);
    buildLCP(s);
    buildSparse();
  }

  void buildArray(string s){
    int n = sz(s) + 1;
    p.resize(n), c.resize(n);
    for (int i = 0; i < n; i++) p[i] = i;
    sort(all(p), [&] (int a, int b){return s[a] < s[b];});
    c[p[0]] = 0;
```

```cpp
    for (int i = 1; i < n; i++){
      c[p[i]] = c[p[i - 1]] + (s[p[i]] != s[p[i - 1]]);
    }
    vector<int> p2(n), c2(n);
    // w is half-length of each string.
    for (int w = 1; w < n; w <<= 1){
      for (int i = 0; i < n; i++){
        p2[i] = (p[i] - w + n) % n;
      }
      vector<int> cnt(n);
      for (auto i : c) cnt[i]++;
      for (int i = 1; i < n; i++) cnt[i] += cnt[i - 1];
      for (int i = n - 1; i >= 0; i--){
        p[--cnt[c[p2[i]]]] = p2[i];
      }
      c2[p[0]] = 0;
      for (int i = 1; i < n; i++){
        c2[p[i]] = c2[p[i - 1]] +
        (c[p[i]] != c[p[i - 1]] ||
        c[(p[i] + w) % n] != c[(p[i - 1] + w) % n]);
      }
      c.swap(c2);
    }
    p.erase(p.begin());
  }

  void buildLCP(string s){
    // The algorithm assumes that suffix array is already built on the same string.
    int n = sz(s);
    h.resize(n - 1);
    int k = 0;
    for (int i = 0; i < n; i++){
      if (c[i] == n){
        k = 0;
        continue;
      }
      int j = p[c[i]];
      while (i + k < n && j + k < n && s[i + k] == s[j + k]) k++;
      h[c[i] - 1] = k;
      if (k) k--;
    }
    /*
    Then an RMQ Sparse Table can be built on array h
    to calculate LCP of 2 non-consecutive suffixes.
    */
  }

  void buildSparse(){
    st.build(h);
  }

  // l and r must be in 0-BASED INDEXATION
  int lcp(int l, int r){
    l = c[l] - 1, r = c[r] - 1;
    if (l > r) swap(l, r);
    return st.query(l, r - 1);
  }
};
```

## 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).

```cpp
const int S = 26;

// Function converting char to int.
int ctoi(char c){
  return c - 'a';
}

// To add terminal links, use DFS
struct Node{
  vector<int> nxt;
  int link;
  bool terminal;

  Node() {
    nxt.assign(S, -1), link = 0, terminal = 0;
  }
};

vector<Node> trie(1);

// add_string returns the terminal vertex.
int add_string(string& s){
  int v = 0;
  for (auto c : s){
    int cur = ctoi(c);
    if (trie[v].nxt[cur] == -1){
      trie[v].nxt[cur] = sz(trie);
      trie.emplace_back();
    }
    v = trie[v].nxt[cur];
  }
  trie[v].terminal = 1;
  return v;
}

/*
Suffix links are compressed.
This means that:
  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.
*/
void add_links(){
  queue<int> q;
  q.push(0);
  while (!q.empty()){
    auto v = q.front();
    int u = trie[v].link;
    q.pop();
    for (int i = 0; i < S; i++){
      int& ch = trie[v].nxt[i];
      if (ch == -1){
        ch = v? trie[u].nxt[i] : 0;
      }
      else{
        trie[ch].link = v? trie[u].nxt[i] : 0;
        q.push(ch);
      }
    }
  }
}

bool is_terminal(int v){
  return trie[v].terminal;
}

int get_link(int v){
  return trie[v].link;
}

int go(int v, char c){
  return trie[v].nxt[ctoi(c)];
}
```

## 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.

```
struct line{
  ll k, b;
  ll f(ll x){
    return k * x + b;
  };
};

vector<line> hull;

void add_line(line nl){
  if (!hull.empty() && hull.back().k == nl.k){
    nl.b = min(nl.b, hull.back().b); // Default: minimum. For maximum change "min" to "max".
    hull.pop_back();
  }
  while (sz(hull) > 1){
    auto& l1 = hull.end()[-2], l2 = hull.back();
    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 <=.
    else break;
  }
  hull.pb(nl);
}

ll get(ll x){
  int l = 0, r = sz(hull);
  while (r - l > 1){
    int mid = (l + r) / 2;
    if (hull[mid - 1].f(x) >= hull[mid].f(x)) l = mid; // Default: minimum. For maximum change the sign to <=.
    else r = mid;
  }
  return hull[l].f(x);
}
```

## 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;
    val = 0;
    if (l) val += l->val;
    if (r) val += r->val;
  }
  Node(Node *cp) : val(cp->val), l(cp->l), r(cp->r) {}
};
const int N = 2e5 + 20;
ll a[N];
Node *roots[N];
int n, cnt = 1;
Node *build(int l = 1, int r = n) {
  if (l == r) return new Node(a[l]);
  int mid = (l + r) / 2;
  return new Node(build(l, mid), build(mid + 1, r));
}
```

```cpp
Node *update(Node *node, int val, int pos, int l = 1, int r = n) {
  if (l == r) return new Node(val);
  int mid = (l + r) / 2;
  if (pos > mid)
    return new Node(node->l, update(node->r, val, pos, mid + 1, r));
  else return new Node(update(node->l, val, pos, l, mid), node->r);
}
ll query(Node *node, int a, int b, int l = 1, int r = n) {
  if (l > b || r < a) return 0;
  if (l >= a && r <= b) return node->val;
  int mid = (l + r) / 2;
  return query(node->l, a, b, l, mid) + query(node->r, a, b, mid + 1, r);
}
```

# Dynamic Programming

## Sum over Subset DP

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

```cpp
for (int i = 0; i < (1 << n); i++) f[i] = a[i];
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:
- $dp[i][j] = \min\limits_{0 \le k \le j-1} (dp[i-1][k] + cost(k+1, j))$
- **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 $dp[M][N]$.

```cpp
vector<ll> dp_old(N), dp_new(N);

void rec(int l, int r, int optl, int optr){
  if (l > r) return;
  int mid = (l + r) / 2;
  pair<ll, int> best = {INF, optl};
  for (int i = optl; i <= min(mid - 1, optr); i++){ // If k can be j, change to "i <= min(mid, optr)".
    ll cur = dp_old[i] + cost(i + 1, mid);
    if (cur < best.fi) best = {cur, i};
  }
  dp_new[mid] = best.fi;

  rec(l, mid - 1, optl, best.se);
  rec(mid + 1, r, best.se, optr);
}

// Computes the DP "by layers"
fill(all(dp_old), INF);
dp_old[0] = 0;
while (layers--){
   rec(0, n, 0, n);
   dp_old = dp_new;
 }
```

## Knuth's DP Optimization

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

```cpp
int N;
int dp[N][N], opt[N][N];
auto C = [&](int i, int j) {
  // Implement cost function C.
};
for (int i = 0; i < N; i++) {
  opt[i][i] = i;
  // Initialize dp[i][i] according to the problem
}
for (int i = N-2; i >= 0; i--) {
  for (int j = i+1; j < N; j++) {
    int mn = INT_MAX;
    int cost = C(i, j);
    for (int k = opt[i][j-1]; k <= min(j-1, opt[i+1][j]); k++) {
      if (mn >= dp[i][k] + dp[k+1][j] + cost) {
        opt[i][j] = k;
        mn = dp[i][k] + dp[k+1][j] + cost;
      }
    }
    dp[i][j] = mn;
  }
}
```

# Miscellaneous

## Ordered Set

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

## Measuring Execution Time

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

## Setting Fixed D.P. Precision

```cpp
cout << setprecision(d) << fixed;
// 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!