TCR

ℚ++(Daan van Gent, Onno Berrevoets, Pim Spelier)

Contents

Test script (usage: ./test.sh A/B/..)

```
g++ -Wall -Wshadow -Wfatal-errors -Wpedantic
↪  -std=c++17 $1.cc || exit
for i in $1/*.in
do
  j="${i/.in/.ans}"
  ./a.out < $i > output
  diff output $j || echo "!!WA on $i!!"
done
```

template.cpp

```
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
typedef long double ld;
typedef pair<ll, ll> ii;
typedef vector<ll> vi;
typedef vector<vi> vvi;
typedef vector<ii> vii;

#define x first
#define y second
#define pb push_back
#define eb emplace_back
#define rep(i,a,b) for(auto i=(a);i!=(b); ++i)
#define REP(i,n) rep(i,0,n)
#define all(v) (v).begin(), (v).end()
#define rs resize
#define DBG(x) cerr << __LINE__ << ": " << #x << " =
↪  " << (x) << endl

template<class T> using min_queue = priority_queue<T,
↪  vector<T>, greater<T>>;
template<class T> int size(const T &x) { return
↪  x.size(); } // copy the ampersand(&)!

const ll INF = 2147483647;
const ll LLINF = ~(1LL<<63); // =
↪  9.223.372.036.854.775.807
const ld PI = acos(-1.0);

void run() {

}

signed main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  (cout << fixed).precision(18);
  run();
  return 0;
}
```

0.1. De winnende aanpak.

- Slaap goed & heb een vroeg ritme!
- Drink & eet genoeg voor & tijdens de wedstrijd!
- Houd een lijst bij met info over alle problemen.
- Iedereen moet **ALLE** opgaves **goed** lezen!
- Analyseer de voorbeeld test cases.
- Houd na 2 uur een pauze en overleg waar iedereen mee bezig is.
- Maak zelf (zware) test cases.
- Gebruik `ll`.

0.2. Wrong Answer.

(1) Print de oplossing om te debuggen!
(2) Kijk naar wellicht makkelijkere problemen.
(3) Bedenk zelf test cases met **randgevallen!**
(4) Controleer de **precisie**.
(5) Controleer op **overflow** (gebruik **OVERAL** `ll`, `ld`).
Kijk naar overflows in tussenantwoorden bij modulo.
(6) Controleer op **typo's**.
(7) Loop de voorbeeld test case accuraat langs.
(8) Controleer op off-by-one-errors (in indices of lus-grenzen)?

Detecting overflow This GNU builtin checks for over- and underflow. Result is in `res` if successful:

```
bool isOverflown =
↪  __builtin_[add|mul|sub]_overflow(a, b, &res);
```

0.3. Covering problems.

Minimum edge cover $\iff$ *Maximum independent set*

Matching: A set of edges without common vertices *(Maximum is the **largest** such set, maximal is a set which you cannot add more edges to without breaking the property).*

Minimum Vertex Cover: A set vertices (cover) such that each edge in the graph is incident to at least one vertex of the set.

Minimum Edge Cover: A set of edges (cover) such that every vertex is incident to at least one edge of the set.

Maximum Independent Set: A set of vertices in a graph such that no two of them are adjacent.

König's theorem: In any bipartite graph, MCBM = MVC = V - MIS.

A useful identity: $\bigoplus_{x=0}^{a-1} x = \{0, a-1, 1, a\}[a \bmod 4]$.

1. Mathematics

```
int abs(int x) { return x > 0 ? x : -x; }
int sign(int x) { return (x > 0) - (x < 0); }

// greatest common divisor
ll gcd(ll a,ll b){while(b)a%=b,swap(a,b);return a;};
// least common multiple
ll lcm(ll a, ll b) { return a/gcd(a, b)*b; }
ll mod(ll a, ll b) { return (a %= b) < 0 ? a+b : a; }

// ab % m for m <= 4e18 in O(log b)
ll mod_mul(ll a, ll b, ll m) {
  ll r = 0;
  while(b) {
    if (b & 1) r = mod(r+a,m);
    a = mod(a+a,m); b >>= 1;
  }
  return r;
}

// a^b % m for m <= 2e9 in O(log b)
ll mod_pow(ll a, ll b, ll m) {
  ll r = 1;
  while(b) {
    if (b & 1) r = (r * a) % m; // mod_mul
    a = (a * a) % m; // mod_mul
    b >>= 1;
  }
  return r;
}

// returns x, y such that ax + by = gcd(a, b)
ll egcd(ll a, ll b, ll &x, ll &y) {
  ll xx = y = 0, yy = x = 1;
  while (b) {
    x -= a / b * xx; swap(x, xx);
    y -= a / b * yy; swap(y, yy);
    a %= b; swap(a, b);
  }
  return a;
}

// Chinese Remainder Theorem: returns (u, v) s.t.
// x=u (mod v) <=> x=a (mod n) and x=b (mod m)
pair<ll, ll> crt(ll a, ll n, ll b, ll m) { //n,m<=1e9
  ll s, t, d = egcd(n, m, s, t);
  if (mod(a - b, d)) return { 0, -1 };
  return { mod(s*b%m*n + t*a%n*m, n*m)/d, n*m/d };
}

// phi[i] = #{ 0 < j <= i | gcd(i, j) = 1 } sieve
vi totient(int N) {
  vi phi(N);
  for (int i = 0; i < N; i++) phi[i] = i;
  for (int i = 2; i < N; i++) if (phi[i] == i)
    for (int j = i; j < N; j+=i) phi[j] -= phi[j]/i;
  return phi;
}

// calculate nCk % p (p prime!)
ll lucas(ll n, ll k, ll p) {
  ll ans = 1;
  while (n) {
    ll np = n % p, kp = k % p;
    if (np < kp) return 0;
    ans = mod(ans * binom(np, kp), p); // (np C kp)
    n /= p; k /= p;
  }
  return ans;
}

// returns if n is prime for n < 3e24 (>2^64)
// but use mul_mod for n > 2e9.
bool millerRabin(ll n){
  if (n < 2 || n % 2 == 0) return n == 2;
  ll d = n - 1, ad, s = 0, r;
  for (; d % 2 == 0; d /= 2) s++;
  for (int a : {  2,   3,   5,   7, 11, 13,
           17, 19, 23, 29, 31, 37, 41 }) {
    if (n == a) return true;
    if ((ad = mod_pow(a, d, n)) == 1) continue;
    for (r = 0; r < s && ad + 1 != n; r++)
      ad = (ad * ad) % n;
    if (r == s) return false;
  }
  return true;
}
```

1.1. Primitive Root $O(\sqrt{m})$.

Returns a generator of $\mathbb{F}_m^*$. If m not prime, replace $m-1$ by totient of m.

```
ll primitive_root(ll m) {
  vector<ll> div;
  for (ll i = 1; i*i < m; i++) {
    if ((m-1) % i == 0) {
      if (i < m-1) div.pb(i);
      if ((m-1)/i < m) div.pb((m-1)/i); } }
  rep(x,2,m) {
    bool ok = true;
    for (ll d : div)
      if (mod_pow(x, d, m) == 1) {
        ok = false; break; }
    if (ok) return x; }
  return -1; }
```

1.2. Tonelli-Shanks algorithm.

Given prime p and integer $1 \leq n < p$, returns the square root r of n modulo p. There is also another solution given by $-r$ modulo p.

```
ll legendre(ll a, ll p) {
  if (a % p == 0) return 0;
  if (p == 2) return 1;
  return mod_pow(a, (p-1)/2, p) == 1 ? 1 : -1; }
ll tonelli_shanks(ll n, ll p) {
  assert(legendre(n,p) == 1);
  if (p == 2) return 1;
  ll s = 0, q = p-1, z = 2;
  while (~q & 1) s++, q >>= 1;
  if (s == 1) return mod_pow(n, (p+1)/4, p);
  while (legendre(z,p) != -1) z++;
  ll c = mod_pow(z, q, p),
     r = mod_pow(n, (q+1)/2, p),
     t = mod_pow(n, q, p),
     m = s;
  while (t != 1) {
    ll i = 1, ts = (ll)t*t % p;
    while (ts != 1) i++, ts = ((ll)ts * ts) % p;
    ll b = mod_pow(c, 1LL<<(m-i-1), p);
    r = (ll)r * b % p;
    t = (ll)t * b % p * b % p;
    c = (ll)b * b % p;
    m = i; }
  return r; }
```

1.3. **Numeric Integration.** Numeric integration using Simpson's rule.

```
ld numint(ld (*f)(ld), ld a, ld b, ld EPS = 1e-6) {
  ld ba = b - a, m=(a+b)/2;
  return abs(ba) < EPS ?
  ↪  ba/8*(f(a)+f(b)+f(a+ba/3)*3+f(b-ba/3)*3)
    : numint(f,a,m,EPS) + numint(f,m,b,EPS);
}
```

1.4. **Fast Hadamard Transform.** Computes the Hadamard transform of the given array. Can be used to compute the XOR-convolution of arrays, exactly like with FFT. For AND-convolution, replace $(x-y, x+y), ((x+y)/2, (-x+y)/2)$ with $(x+y, y), (x-y, y)$. For OR-convolution, use $(x, x+y)$ and $(x, -x+y)$. **Note:** Size of array must be a power of 2.

```
void fht(vi &arr, bool inv=false, int l, int r) {
  if (l+1 == r) return;
  int k = (r-l)/2;
  if (!inv) fht(arr, inv, l, l+k), fht(arr, inv, l+k,
  ↪  r);
  rep(i,l,l+k) { int x = arr[i], y = arr[i+k];
    if (!inv) arr[i] = x-y, arr[i+k] = x+y;
    else arr[i] = (x+y)/2, arr[i+k] = (-x+y)/2; }
  if (inv) fht(arr, inv, l, l+k), fht(arr, inv, l+k,
  ↪  r); }
```

1.5. **Tridiagonal Matrix Algorithm.** Solves a tridiagonal system of linear equations $a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$ where $a_1 = c_n = 0$. Beware of numerical instability.

```
#define MAXN 5000
ld A[MAXN], B[MAXN], C[MAXN], D[MAXN], X[MAXN];
void solve(int n) {
  C[0] /= B[0]; D[0] /= B[0];
  rep(i,1,n-1) C[i] /= B[i] - A[i]*C[i-1];
  rep(i,1,n) D[i] = (D[i] - A[i]*D[i-1]) / (B[i] -
  ↪  A[i]*C[i-1]);
  X[n-1] = D[n-1];
  for (int i = n-1; i--;)
    X[i] = D[i] - C[i] * X[i+1]; }
```

1.6. **Josephus problem.** Last man standing out of n if every kth is killed. Zero-based, and does not kill 0 on first pass.

```
int J(int n, int k) {
  if (n == 1) return 0;
  if (k == 1) return n-1;
  if (n < k) return (J(n-1,k)+k)%n;
  int np = n - n/k;
  return k*((J(np,k)+np-n%k%np)%np) / (k-1); }
```

1.7. **Number of Integer Points under Line.** Count the number of integer solutions to $Ax + By \le C$, $0 \le x \le n$, $0 \le y$. In other words, evaluate the sum $\sum_{x=0}^{n} \left\lfloor \frac{C-Ax}{B} + 1 \right\rfloor$. To count all solutions, let $n = \left\lfloor \frac{c}{a} \right\rfloor$. In any case, it must hold that $C - nA \ge 0$. Be very careful about overflows.

```
ll floor_sum(ll n, ll a, ll b, ll c) {
  if (c == 0) return 1;
  if (c < 0) return 0;
  if (a % b == 0) return (n+1)*(c/b+1)-n*(n+1)/2*a/b;
  if (a >= b) return
  ↪  floor_sum(n,a%b,b,c)-a/b*n*(n+1)/2;
  ll t = (c-a*n+b)/b;
  return floor_sum((c-b*t)/b,b,a,c-b*t)+t*(n+1); }
```

1.8. **Misc.** Prime numbers:
1031, 32771, 1048583, 8125344, 33554467, 9982451653, 1073741827, 34359738421, 1099511627791, 35184372088891, 1125899906842679, 36028797018963971.
$10^3 + \{-9, -3, 9, 13\}$, $10^6 + \{-17, 3, 33\}$, $10^9 + \{7, 9, 21, 33, 87\}$.

- **Generating functions**: Ordinary (ogf): $A(x) := \sum_{n=0}^{\infty} a_i x^i$.
  Calculate product $c_n = \sum_{k=0}^{n} a_k b_{n-k}$ with FFT.
  Exponential (e.g.f.): $A(x) := \sum_{n=0}^{\infty} a_i x^i / i!$,
  $c_n = \sum_{k=0}^{n} \binom{n}{k} a_k b_{n-k} = n! \sum_{k=0}^{n} \frac{a_k}{k!} \frac{b_{n-k}}{(n-k)!}$ (use FFT).
- **General linear recurrences:** If $a_n = \sum_{k=0}^{n-1} a_k b_{n-k}$, then $A(x) = \frac{a_0}{1-B(x)}$.
- **Inverse polynomial modulo x^l:** Given $A(x)$, find $B(x)$ such that $A(x)B(x) = 1 + x^l Q(x)$ for some $Q(x)$.
  Step 1: Start with $B_0(x) = 1/a_0$
  Step 2: $B_{k+1}(x) = (-B_k(x)^2 A(x) + 2B_k(x)) \mod x^{2^{k+1}}$.
- **Fast subset convolution:** Given array a_i of size 2^k calculate $b_i = \sum_{j \& i = i} a_j$.

```
for (int b = 1; b < (1 << k); b <<= 1)
  for (int i = 0; i < (1<<k); i++)
    if (!(i & b)) a[i | b] += a[i];
// inv: if (!(i & b)) a[i | b] -= a[i];
```

- **Primitive Roots:** It only exists when n is $2, 4, p^k, 2p^k$, where p odd prime. If g is a primitive root, all primitive roots are of the form g^k where $k, \phi(p)$ are coprime (hence there are $\phi(\phi(p))$ primitive roots).
-

$\le N$	10^3	10^6	10^9	10^{12}	10^{18}
m	840	720720	735134400	963761198400	
$d(m)$	32	240	1344	6270	103680

For $n = 10^{18}, m = 897612484786617600$.

2. Datastructures

2.1. **Order tree.**

```
#include <bits/extc++.h>
using namespace __gnu_pbds;
template<class TK,class TM> using order_tree =
↪  tree<TK,TM,greater<TK>,rb_tree_tag,
↪  tree_order_statistics_node_update>;
template<class TK> using order_set =
↪  order_tree<TK,null_type>;

vi s;
order_set<ii> t;
void update( ll k, ll v ) {
  t.erase( ii{ s[k], k } );
  s[k] = v;
  t.insert( ii{ s[k], k } );
}

signed main() {
  ll n = 4;
  s.resize(n,0);
  rep(i,0,n) t.insert(ii{0,i});
  update( 2, 3 );
  cout << t.find_by_order( 2 )->y << endl;
  cout << t.order_of_key( ii{s[3],3} ) << endl;
}
```

2.2. **Segment tree $\mathcal{O}(\log n)$. Standard segment tree**

```
typedef /* Tree element */ S;
const int n = 1 << 20; S t[2 * n];

// required axiom: associativity
S combine(S l, S r) { return l + r; } // sum segment
↪  tree
S combine(S l, S r) { return max(l, r); } // max
↪  segment tree

void build() { for (int i = n; --i; ) t[i] =
↪  combine(t[2 * i], t[2 * i + 1]); }

// set value v on position i
void update(int i, S v) { for (t[i += n] = v; i /= 2;
↪  ) t[i] = combine(t[2 * i], t[2 * i + 1]);}

// sum on interval [l, r)
S query(int l, int r) {
  S resL = 0, resR = 0;
  for (l += n, r += n; l < r; l /= 2, r /= 2) {
    if (l & 1) resL = combine(resL, t[l++]);
    if (r & 1) resR = combine(t[--r], resR);
  }
  return combine(resL, resR);
}
```

Lazy segment tree

Be careful: all intervals are right-closed $[\ell, r]$.

```
struct node {
  int l, r, x, lazy;
  node() {}
  node(int _l, int _r) : l(_l), r(_r), x(INF),
  ↪  lazy(0) { }
  node(int _l, int _r, int _x) : node(_l,_r) { x =
  ↪  _x; }
  node(node a, node b) : node(a.l,b.r) { x = min(a.x,
  ↪  b.x); }
  void update(int v) { x = v; }
  void range_update(int v) { lazy = v; }
  void apply() { x += lazy; lazy = 0; }
  void push(node &u) { u.lazy += lazy; } };

struct segment_tree {
  int n;
```

```
  vector<node> arr;
  segment_tree() { }
  segment_tree(const vector<ll> &a) : n(size(a)),
  ↪  arr(4*n) {
    mk(a,0,0,n-1); }
  node mk(const vector<ll> &a, int i, int l, int r) {
    int m = (l+r)/2;
    return arr[i] = l > r ? node(l,r) :
      l == r ? node(l,r,a[l]) :
      node(mk(a,2*i+1,l,m),mk(a,2*i+2,m+1,r)); }
  node update(int at, ll v, int i=0) {
    propagate(i);
    int hl = arr[i].l, hr = arr[i].r;
    if (at < hl || hr < at) return arr[i];
    if (hl == at && at == hr) {
      arr[i].update(v); return arr[i]; }
    return arr[i] =
      node(update(at,v,2*i+1),update(at,v,2*i+2)); }
  node query(int l, int r, int i=0) {
    propagate(i);
    int hl = arr[i].l, hr = arr[i].r;
    if (r < hl || hr < l) return node(hl,hr);
    if (l <= hl && hr <= r) return arr[i];
    return node(query(l,r,2*i+1),query(l,r,2*i+2)); }
  node range_update(int l, int r, ll v, int i=0) {
    propagate(i);
    int hl = arr[i].l, hr = arr[i].r;
    if (r < hl || hr < l) return arr[i];
    if (l <= hl && hr <= r)
      return arr[i].range_update(v), propagate(i),
      ↪  arr[i];
    return arr[i] = node(range_update(l,r,v,2*i+1),
        range_update(l,r,v,2*i+2)); }
    void propagate(int i) {
      if (arr[i].l < arr[i].r)
        arr[i].push(arr[2*i+1]),
        ↪  arr[i].push(arr[2*i+2]);
      arr[i].apply(); } };
```

Persistent segment tree Be careful: all intervals are right-closed $[\ell, r]$, including build.

```
int segcnt = 0;
struct segment {
  int l, r, lid, rid, sum;
} segs[2000000];
int build(int l, int r) {
  if (l > r) return -1;
  int id = segcnt++;
  segs[id].l = l;
  segs[id].r = r;
  if (l == r) segs[id].lid = -1, segs[id].rid = -1;
  else {
    int m = (l + r) / 2;
    segs[id].lid = build(l , m);
    segs[id].rid = build(m + 1, r); }
  segs[id].sum = 0;
  return id; }
int update(int idx, int v, int id) {
  if (id == -1) return -1;
  if (idx < segs[id].l || idx > segs[id].r) return
  ↪  id;
  int nid = segcnt++;
  segs[nid].l = segs[id].l;
  segs[nid].r = segs[id].r;
  segs[nid].lid = update(idx, v, segs[id].lid);
  segs[nid].rid = update(idx, v, segs[id].rid);
  segs[nid].sum = segs[id].sum + v;
  return nid; }
int query(int id, int l, int r) {
  if (r < segs[id].l || segs[id].r < l) return 0;
  if (l <= segs[id].l && segs[id].r <= r) return
  ↪  segs[id].sum;
  return query(segs[id].lid, l, r)
       + query(segs[id].rid, l, r); }
```

2.3. Binary Indexed Tree $\mathcal{O}(\log n)$. Use one-based indices ($i > 0$)!

```
struct BIT {
  int n;
  vector<ll> A;

  BIT(int _n) : n(_n), A(n, 0) {}
  // A[i] += v
  void update(int i, ll v) {
    while (i < n) A[i] += v, i += i & -i;
  }
  // returns sum_{0<j<=i} A[j]
  ll query(int i) {
    ll v = 0; while (i > 0) v += A[i], i -= i & -i;
    ↪  return v;
  }
};
```

Use this if you add things, which depend on i:

```
struct fenwick_tree {
  int n; vi data;
  fenwick_tree(int _n) : n(_n), data(vi(n)) { }
  void update(int at, int by) {
    while (at < n) data[at] += by, at |= at + 1; }
  int query(int at) {
    int res = 0;
    while (at >= 0) res += data[at], at = (at & (at +
    ↪  1)) - 1;
    return res; }
  int rsq(int a, int b) { return query(b) - query(a -
  ↪  1); }
};
struct fenwick_tree_sq {
  int n; fenwick_tree x1, x0;
  fenwick_tree_sq(int _n) : n(_n),
  ↪  x1(fenwick_tree(n)),
    x0(fenwick_tree(n)) { }
  // insert f(y) = my + c if x <= y
  void update(int x, int m, int c) {
    x1.update(x, m); x0.update(x, c); }
  int query(int x) { return x*x1.query(x) +
  ↪  x0.query(x); }
};
void range_update(fenwick_tree_sq &s, int a, int b,
↪  int k) {
  s.update(a, k, k * (1 - a)); s.update(b+1, -k, k *
  ↪  b); }
int range_query(fenwick_tree_sq &s, int a, int b) {
  return s.query(b) - s.query(a-1); }
```

2.4. Disjoint-Set / Union-Find $\mathcal{O}(\alpha(n))$.

```
struct dsu {
  vi par, rnk;
  dsu(int n) : par(n, -1), rnk(n, 0) {}
  int find(int i) { return
    par[i] < 0 ? i : par[i] = find(par[i]); }
  void unite(int a, int b) {
    if ((a = find(a)) == (b = find(b))) return;
    if (rnk[a] < rnk[b]) swap(a, b);
    if (rnk[a] == rnk[b]) rnk[a]++;
    par[a] += par[b]; par[b] = a;
  }
};
```

Use this easy implementation for a map:

```
template <class K, class V> struct avl_map {
  struct node {
    K key; V value;
    node(K k, V v) : key(k), value(v) { }
    bool operator <(const node &other) const {
      return key < other.key; } };
  avl_tree<node> tree;
  V& operator [](K key) {
    typename avl_tree<node>::node *n =
      tree.find(node(key, V(0)));
    if (!n) n = tree.insert(node(key, V(0)));
    return n->item.value; } };
```

2.5. Cartesian tree.

```
struct node {
  int x, y, sz;
  node *l, *r;
  node(int _x, int _y)
    : x(_x), y(_y), sz(1), l(NULL), r(NULL) { } };
int tsize(node* t) { return t ? t->sz : 0; }
void augment(node *t) {
  t->sz = 1 + tsize(t->l) + tsize(t->r); }
pair<node*,node*> split(node *t, int x) {
  if (!t) return make_pair((node*)NULL,(node*)NULL);
  if (t->x < x) {
    pair<node*,node*> res = split(t->r, x);
    t->r = res.first; augment(t);
    return make_pair(t, res.second); }
  pair<node*,node*> res = split(t->l, x);
  t->l = res.second; augment(t);
  return make_pair(res.first, t); }
node* merge(node *l, node *r) {
  if (!l) return r; if (!r) return l;
  if (l->y > r->y) {
```

```
    l->r = merge(l->r, r); augment(l); return l; }
  r->l = merge(l, r->l); augment(r); return r; }
node* find(node *t, int x) {
  while (t) {
    if (x < t->x) t = t->l;
    else if (t->x < x) t = t->r;
    else return t; }
  return NULL; }
node* insert(node *t, int x, int y) {
  if (find(t, x) != NULL) return t;
  pair<node*,node*> res = split(t, x);
  return merge(res.first,
      merge(new node(x, y), res.second)); }
node* erase(node *t, int x) {
  if (!t) return NULL;
  if (t->x < x) t->r = erase(t->r, x);
  else if (x < t->x) t->l = erase(t->l, x);
  else { node *old = t; t = merge(t->l, t->r); delete
  ↪ old; }
  if (t) augment(t); return t; }
int kth(node *t, int k) {
  if (k < tsize(t->l)) return kth(t->l, k);
  else if (k == tsize(t->l)) return t->x;
  else return kth(t->r, k - tsize(t->l) - 1); }
```

2.6. **Heap.** An implementation of a binary heap.

```
#define RESIZE
#define SWP(x,y) tmp = x, x = y, y = tmp
struct default_int_cmp {
  default_int_cmp() { }
  bool operator ()(const int &a, const int &b) {
    return a < b; } };
template <class Compare = default_int_cmp> struct
↪ heap {
  int len, count, *q, *loc, tmp;
  Compare _cmp;
  inline bool cmp(int i, int j) { return _cmp(q[i],
  ↪ q[j]); }
  inline void swp(int i, int j) {
    SWP(q[i], q[j]), SWP(loc[q[i]], loc[q[j]]); }
  void swim(int i) {
    while (i > 0) {
      int p = (i - 1) / 2;
      if (!cmp(i, p)) break;
      swp(i, p), i = p; } }
  void sink(int i) {
    while (true) {
      int l = 2*i + 1, r = l + 1;
      if (l >= count) break;
      int m = r >= count || cmp(l, r) ? l : r;
      if (!cmp(m, i)) break;
      swp(m, i), i = m; } }
  heap(int init_len = 128)
    : count(0), len(init_len), _cmp(Compare()) {
    q = new int[len], loc = new int[len];
    memset(loc, 255, len << 2); }
  ~heap() { delete[] q; delete[] loc; }
  void push(int n, bool fix = true) {
    if (len == count || n >= len) {
#ifdef RESIZE
      int newlen = 2 * len;
      while (n >= newlen) newlen *= 2;
      int *newq = new int[newlen], *newloc = new
      ↪ int[newlen];
      rep(i,0,len) newq[i] = q[i], newloc[i] =
      ↪ loc[i];
      memset(newloc + len, 255, (newlen - len) << 2);
      delete[] q, delete[] loc;
      loc = newloc, q = newq, len = newlen;
#else
      assert(false);
#endif
    }
    assert(loc[n] == -1);
    loc[n] = count, q[count++] = n;
    if (fix) swim(count-1); }
  void pop(bool fix = true) {
    assert(count > 0);
    loc[q[0]] = -1, q[0] = q[--count], loc[q[0]] = 0;
    if (fix) sink(0);
  }
  int top() { assert(count > 0); return q[0]; }
  void heapify() { for (int i = count - 1; i > 0;
  ↪ i--)
    if (cmp(i, (i - 1) / 2)) swp(i, (i - 1) / 2); }
  void update_key(int n) {
    assert(loc[n] != -1), swim(loc[n]), sink(loc[n]);
    ↪ }
  bool empty() { return count == 0; }
  int size() { return count; }
  void clear() { count = 0, memset(loc, 255, len <<
  ↪ 2); }};
```

2.7. **Dancing Links.** An implementation of Donald Knuth's Dancing Links data structure. A linked list supporting deletion and restoration of elements.

```
template <class T>
struct dancing_links {
  struct node {
    T item;
    node *l, *r;
    node(const T &_item, node *_l = NULL, node *_r =
    ↪ NULL)
      : item(_item), l(_l), r(_r) {
      if (l) l->r = this;
      if (r) r->l = this; } };
  node *front, *back;
  dancing_links() { front = back = NULL; }
  node *push_back(const T &item) {
    back = new node(item, back, NULL);
    if (!front) front = back;
    return back; }
  node *push_front(const T &item) {
    front = new node(item, NULL, front);
    if (!back) back = front;
    return front; }
  void erase(node *n) {
    if (!n->l) front = n->r; else n->l->r = n->r;
    if (!n->r) back = n->l; else n->r->l = n->l; }
  void restore(node *n) {
    if (!n->l) front = n; else n->l->r = n;
    if (!n->r) back = n; else n->r->l = n; } };
```

2.8. **Misof Tree.** A simple tree data structure for inserting, erasing, and querying the nth largest element.

```
#define BITS 15
struct misof_tree {
  int cnt[BITS][1<<BITS];
  misof_tree() { memset(cnt, 0, sizeof(cnt)); }
  void insert(int x) {
    for (int i = 0; i < BITS; cnt[i++][x]++, x >>=
    ↪ 1); }
  void erase(int x) {
    for (int i = 0; i < BITS; cnt[i++][x]--, x >>=
    ↪ 1); }
  int nth(int n) {
    int res = 0;
    for (int i = BITS-1; i >= 0; i--)
      if (cnt[i][res <<= 1] <= n) n -= cnt[i][res],
      ↪ res |= 1;
    return res; } };
```

2.9. ***k*-d Tree.** A k-dimensional tree supporting fast construction, adding points, and nearest neighbor queries. NOTE: Not completely stable, occasionally segfaults.

```
#define INC(c) ((c) == K - 1 ? 0 : (c) + 1)
template <int K> struct kd_tree {
  struct pt {
    double coord[K];
    pt() {}
    pt(double c[K]) { rep(i,0,K) coord[i] = c[i]; }
    double dist(const pt &other) const {
      double sum = 0.0;
      rep(i,0,K) sum += pow(coord[i] -
      ↪ other.coord[i], 2.0);
      return sqrt(sum); } };
  struct cmp {
    int c;
    cmp(int _c) : c(_c) {}
    bool operator ()(const pt &a, const pt &b) {
      for (int i = 0, cc; i <= K; i++) {
        cc = i == 0 ? c : i - 1;
        if (abs(a.coord[cc] - b.coord[cc]) > EPS)
          return a.coord[cc] < b.coord[cc];
      }
      return false; } };
  struct bb {
    pt from, to;
    bb(pt _from, pt _to) : from(_from), to(_to) {}
    double dist(const pt &p) {
      double sum = 0.0;
```

```
    rep(i,0,K) {
      if (p.coord[i] < from.coord[i])
        sum += pow(from.coord[i] - p.coord[i], 2.0);
      else if (p.coord[i] > to.coord[i])
        sum += pow(p.coord[i] - to.coord[i], 2.0);
    }
    return sqrt(sum); }
  bb bound(double l, int c, bool left) {
    pt nf(from.coord), nt(to.coord);
    if (left) nt.coord[c] = min(nt.coord[c], l);
    else nf.coord[c] = max(nf.coord[c], l);
    return bb(nf, nt); } };
struct node {
  pt p; node *l, *r;
  node(pt _p, node *_l, node *_r)
    : p(_p), l(_l), r(_r) {  } };
node *root;
// kd_tree() : root(NULL) { }
kd_tree(vector<pt> pts) {
  root = construct(pts, 0, size(pts) - 1, 0); }
node* construct(vector<pt> &pts, int from, int to, int c) {
  if (from > to) return NULL;
  int mid = from + (to - from) / 2;
  nth_element(pts.begin() + from, pts.begin() + mid,
        pts.begin() + to + 1, cmp(c));
  return new node(pts[mid],
          construct(pts, from, mid - 1, INC(c)),
          construct(pts, mid + 1, to, INC(c))); }
bool contains(const pt &p) { return _con(p, root, 0); }
bool _con(const pt &p, node *n, int c) {
  if (!n) return false;
  if (cmp(c)(p, n->p)) return _con(p, n->l, INC(c));
  if (cmp(c)(n->p, p)) return _con(p, n->r, INC(c));
  return true; }
void insert(const pt &p) { _ins(p, root, 0); }
void _ins(const pt &p, node* &n, int c) {
  if (!n) n = new node(p, NULL, NULL);
  else if (cmp(c)(p, n->p)) _ins(p, n->l, INC(c));
  else if (cmp(c)(n->p, p)) _ins(p, n->r, INC(c)); }
void clear() { _clr(root); root = NULL; }
void _clr(node *n) {
  if (n) _clr(n->l), _clr(n->r), delete n; }
pt nearest_neighbour(const pt &p, bool allow_same=true) {
  assert(root);
  double mn = INFINITY, cs[K];
  rep(i,0,K) cs[i] = -INFINITY;
  pt from(cs);
  rep(i,0,K) cs[i] = INFINITY;
  pt to(cs);
  return _nn(p, root, bb(from, to), mn, 0, allow_same).first;
}
pair<pt, bool> _nn(const pt &p, node *n, bb b,
    double &mn, int c, bool same) {
  if (!n || b.dist(p) > mn) return make_pair(pt(), false);
  bool found = same || p.dist(n->p) > EPS,
       l1 = true, l2 = false;
  pt resp = n->p;
  if (found) mn = min(mn, p.dist(resp));
  node *n1 = n->l, *n2 = n->r;
  rep(i,0,2) {
    if (i == 1 || cmp(c)(n->p, p))
      swap(n1, n2), swap(l1, l2);
    pair<pt, bool> res =_nn(p, n1,
        b.bound(n->p.coord[c], c, l1), mn, INC(c), same);
    if (res.second &&
        (!found || p.dist(res.first) < p.dist(resp)))
      resp = res.first, found = true;
  }
  return make_pair(resp, found); } };
```

2.10. **Sqrt Decomposition.** Design principle that supports many operations in amortized $\sqrt{n}$ per operation.

```
struct segment {
  vi arr;
  segment(vi _arr) : arr(_arr) { } };
vector&lt;segment&gt; T;
int K;
void rebuild() {
  int cnt = 0;
  rep(i,0,size(T))
    cnt += size(T[i].arr);
  K = static_cast<int>(ceil(sqrt(cnt)) + 1e-9);
  vi arr(cnt);
  for (int i = 0, at = 0; i < size(T); i++)
    rep(j,0,size(T[i].arr))
      arr[at++] = T[i].arr[j];
  T.clear();
  for (int i = 0; i < cnt; i += K)
    T.push_back(segment(vi(arr.begin()+i,
                        arr.begin()+min(i+K, cnt)))); }
int split(int at) {
  int i = 0;
  while (i < size(T) && at >= size(T[i].arr))
    at -= size(T[i].arr), i++;
  if (i >= size(T)) return size(T);
  if (at == 0) return i;
  T.insert(T.begin() + i + 1,
      segment(vi(T[i].arr.begin() + at, T[i].arr.end())));
  T[i] = segment(vi(T[i].arr.begin(), T[i].arr.begin() + at));
  return i + 1; }
void insert(int at, int v) {
  vi arr; arr.push_back(v);
  T.insert(T.begin() + split(at), segment(arr)); }
void erase(int at) {
  int i = split(at); split(at + 1);
  T.erase(T.begin() + i); }
```

2.11. **Monotonic Queue.** A queue that supports querying for the minimum element. Useful for sliding window algorithms.

```
struct min_stack {
  stack<int> S, M;
  void push(int x) {
    S.push(x);
    M.push(M.empty() ? x : min(M.top(), x)); }
  int top() { return S.top(); }
  int mn() { return M.top(); }
  void pop() { S.pop(); M.pop(); }
  bool empty() { return S.empty(); } };
struct min_queue {
  min_stack inp, outp;
  void push(int x) { inp.push(x); }
  void fix() {
    if (outp.empty()) while (!inp.empty())
      outp.push(inp.top()), inp.pop(); }
  int top() { fix(); return outp.top(); }
  int mn() {
    if (inp.empty()) return outp.mn();
    if (outp.empty()) return inp.mn();
    return min(inp.mn(), outp.mn()); }
  void pop() { fix(); outp.pop(); }
  bool empty() { return inp.empty() && outp.empty(); } };
```

2.12. **Convex Hull Trick (replace $O(n^2)$ by respectively $O(n)$ and $O(n \log n)$).** If converting to integers, look out for division by 0 and $\pm\infty$.

```
struct convex_hull_trick {
  vector<pair<double,double> > h;
  double intersect(int i) {
    return (h[i+1].second-h[i].second) /
      (h[i].first-h[i+1].first); }
  void add(double m, double b) {
    h.push_back(make_pair(m,b));
    while (size(h) >= 3) {
      int n = size(h);
      if (intersect(n-3) < intersect(n-2)) break;
      swap(h[n-2], h[n-1]);
      h.pop_back(); } }
  double get_min(double x) {
    int lo = 0, hi = size(h) - 2, res = -1;
    while (lo <= hi) {
      int mid = lo + (hi - lo) / 2;
      if (intersect(mid) <= x) res = mid, lo = mid + 1;
      else hi = mid - 1; }
    return h[res+1].first * x + h[res+1].second; } };
```

And dynamic variant:

```
const ll is_query = -(1LL<<62);
struct Line {
  ll m, b;
  mutable function<const Line*()> succ;
  bool operator<(const Line& rhs) const {
    if (rhs.b != is_query) return m < rhs.m;
    const Line* s = succ();
    if (!s) return 0;
    ll x = rhs.m;
    return b - s->b < (s->m - m) * x; } };
// will maintain upper hull for maximum
struct HullDynamic : public multiset<Line> {
  bool bad(iterator y) {
    auto z = next(y);
    if (y == begin()) {
      if (z == end()) return 0;
      return y->m == z->m && y->b <= z->b; }
    auto x = prev(y);
    if (z == end()) return y->m == x->m && y->b <= x->b;
    return (x->b - y->b)*(z->m - y->m) >=
           (y->b - z->b)*(y->m - x->m); }
  void insert_line(ll m, ll b) {
    auto y = insert({ m, b });
    y->succ = [=] { return next(y) == end() ? 0 : &*next(y); };
    if (bad(y)) { erase(y); return; }
    while (next(y) != end() && bad(next(y))) erase(next(y));
    while (y != begin() && bad(prev(y))) erase(prev(y)); }
  ll eval(ll x) {
    auto l = *lower_bound((Line) { x, is_query });
    return l.m * x + l.b; } };
```

2.13. Sparse Table $O(\log n)$ per query.

```
struct sparse_table { vvi m;
  sparse_table(vi arr) {
    m.push_back(arr);
    for (int k = 0; (1<<(++k)) <= size(arr); ) {
      m.push_back(vi(size(arr)-(1<<k)+1));
      rep(i,0,size(arr)-(1<<k)+1)
        m[k][i] = min(m[k-1][i], m[k-1][i+(1<<(k-1))]); } }
  int query(int l, int r) {
    int k = 0; while (1<<(k+1) <= r-l+1) k++;
    return min(m[k][l], m[k][r-(1<<k)+1]); } };
```

3. Graph Algorithms

3.1. Shortest path.

3.1.1. *Dijkstra* $\mathcal{O}(|E|\log|V|)$.

```
#define INFTY -1
vi dijkstra( vector<vii> G, ll s ) {
  vi d( G.size(), INFTY );
  priority_queue<ii,vector<ii>,greater<ii>> Q;
  Q.emplace(0,s);
  while(!Q.empty()){
    ll c = Q.top().x, a = Q.top().y;
    Q.pop();
    if(d[a] != INFTY)
      continue;
    d[a] = c;
    for(ii e : G[a])
      Q.emplace(d[a] + e.y, e.x);
  }
  return d;
}
```

3.1.2. *Floyd-Warshall* $\mathcal{O}(V^3)$.

Be careful with negative edges! Note: |d[i][j]| can grow exponentially, and INFTY + negative < INFTY.

```
#define INFTY (1LL<<61LL)
void floyd_warshall( vvi& d ) {
  ll n = d.size();
  rep(i,0,n) rep(j,0,n) rep(k,0,n)
    if( d[j][i] < INFTY and d[i][k] < INFTY ) // !!! neg. edges
      d[j][k] = max(-INFTY,min(d[j][k],d[j][i]+d[i][k]));
}
vvi d(n,vi(n,INFTY)); rep(i,0,n) d[i][i] = 0;
```

3.1.3. *Bellman Ford* $\mathcal{O}(VE)$.

This is only useful if there are edges with weight $w_{ij} < 0$ in the graph.

```
#define INFTY (1LL<<61LL)
// G undirected, (v,w) in G[u] 'n edge van u naar v lengte w
vi bellman_ford( vector<vii> G, ll s ) {
  ll n = G.size();
  vi d(n,INFTY); d[s] = 0;
  rep( loops, 0, n )
    rep( u, 0, n ) if( d[u] != INFTY )
      for( ii e : G[u] )
        if( d[u] + e.y < d[e.x] )
          d[e.x] = d[u] + e.y;
  // detect paths of -INFTY length
  for( ll change = 1; change--; )
    rep( u, 0, n ) if( d[u] != INFTY )
      for( ii e : G[u] ) if( d[e.x] != -INFTY )
        if( d[u] + e.y < d[e.x] )
          d[e.x] = -INFTY, change = 1;
  return d;
}
```

3.1.4. *IDA* algorithm.*

```
int n, cur[100], pos;
int calch() {
  int h = 0;
  rep(i,0,n) if (cur[i] != 0) h += abs(i - cur[i]);
  return h; }
int dfs(int d, int g, int prev) {
  int h = calch();
  if (g + h > d) return g + h;
  if (h == 0) return 0;
  int mn = INF;
  rep(di,-2,3) {
    if (di == 0) continue;
    int nxt = pos + di;
    if (nxt == prev) continue;
    if (0 <= nxt && nxt < n) {
      swap(cur[pos], cur[nxt]);
      swap(pos,nxt);
      mn = min(mn, dfs(d, g+1, nxt));
      swap(pos,nxt);
      swap(cur[pos], cur[nxt]); }
    if (mn == 0) break; }
  return mn; }
int idastar() {
  rep(i,0,n) if (cur[i] == 0) pos = i;
  int d = calch();
  while (true) {
    int nd = dfs(d, 0, -1);
    if (nd == 0 || nd == INF) return d;
    d = nd; } }
```

3.2. Maximum matching $\mathcal{O}(nm)$.

```
const int sizeL = 1e4, sizeR = 1e4;

bool vis[sizeR];
int par[sizeR]; // par : R -> L
vi adj[sizeL]; // adj : L -> (N -> R)

bool match(int u) {
  for (int v : adj[u]) {
    if (vis[v]) continue; vis[v] = true;
    if (par[v] == -1 || match(par[v])) {
      par[v] = u;
      return true;
    }
  }
  return false;
}

// perfect matching iff ret == sizeL == sizeR
int maxmatch() {
  fill_n(par, sizeR, -1); int ret = 0;
  for (int i = 0; i < sizeL; i++) {
    fill_n(vis, sizeR, false);
    ret += match(i);
  }
  return ret;
}
```

3.3. Hopcroft-Karp bipartite matching $\mathcal{O}(E\sqrt{V})$.

```
#define INFTY (1LL<<61LL)

struct bi_graph {
  ll n, m;
  vvi adj;
  vi L, R, d;
  queue<ll> q;
  bi_graph( ll _n, ll _m ) : n(_n), m(_m),
```

```
    adj(n), L(n,-1), R(m,n), d(n+1) {}
  ll add_edge( ll a, ll b ) { adj[a].pb(b); }
  ll bfs() {
    rep(v,0,n)
      if( L[v] == -1 ) d[v] = 0, q.push(v);
      else d[v] = INFTY;
    d[n] = INFTY;
    while( !q.empty() ) {
      ll v = q.front(); q.pop();
      if( d[v] < d[n] )
        for( ll u : adj[v] ) if( d[R[u]] == INFTY )
          d[R[u]] = d[v]+1, q.push(R[u]);
    }
    return d[n] != INFTY;
  }
  ll dfs( ll v ) {
    if( v == n ) return true;
    for( ll u : adj[v] )
      if( d[R[u]] == d[v] + 1 and dfs(R[u]) ) {
        R[u] = v; L[v] = u;
        return true;
      }
    d[v] = INFTY;
    return false;
  }
  ll maximum_matching() {
    ll s = 0;
    while( bfs() ) rep(i,0,n)
      s += L[i] == - 1 && dfs( i );
    return s;
  }
};
```

3.3.1. *Minimum Vertex Cover in Bipartite Graphs.*

```
#include "hopcroft_karp.cpp"
vi alt;
void dfs( bi_graph &G, ll v ) {
  alt[v] = 1;
  for( ll u : G.adj[v] ) {
    alt[u+G.n] = 1;
    if( G.R[u] != G.n && !alt[G.R[u]] )
      dfs(G,G.R[u]);
} }
vi mvc_bipartite( bi_graph &G ) {
  vi res; G.maximum_matching();
  alt.assign( G.n + G.m, 0 );
  rep(i,0,G.n) if( G.L[i] == -1 ) dfs(G,i);
  rep(i,0,G.n) if( !alt[i] ) res.pb(i);
  rep(i,0,G.n) if( alt[G.n+i] ) res.pb(G.n+i);
  return res;
}
```

3.4. **Depth first searches.**

3.4.1. *Cut Points and Bridges* $O(V + E)$.

```
const int MAXN = 5000;
int low[MAXN], num[MAXN], curnum;
void dfs(const vvi &adj, vi &cp, vii &bri, int u, int
↪  p) {
  low[u] = num[u] = curnum++;
  int cnt = 0; bool found = false;
  rep(i,0,size(adj[u])) {
    int v = adj[u][i];
    if (num[v] == -1) {
      dfs(adj, cp, bri, v, u);
      low[u] = min(low[u], low[v]);
      cnt++;
      found = found || low[v] >= num[u];
      if (low[v] > num[u]) bri.push_back(ii(u, v));
    } else if (p != v) low[u] = min(low[u], num[v]);
    ↪  }
  if (found && (p != -1 || cnt > 1)) cp.push_back(u);
  ↪  }

pair<vi,vii> cut_points_and_bridges(const vvi &adj) {
  int n = size(adj);
  vi cp; vii bri;
  memset(num, -1, n << 2);
  curnum = 0;
  rep(i,0,n) if (num[i] == -1) dfs(adj, cp, bri, i,
  ↪  -1);
  return make_pair(cp, bri); }
```

3.4.2. *Strongly Connected Components* $\mathcal{O}(V + E)$.

```
vvi adj, comps;
vi tidx, lnk, cnr, st;
vector<bool> vis;
int age, ncomps;

void tarjan(int v) {
  tidx[v] = lnk[v] = ++age; vis[v] = true; st.pb(v);
  for (int w : adj[v]) {
    if (!tidx[w]) tarjan(w), lnk[v] = min(lnk[v],
    ↪  lnk[w]);
    else if (vis[w]) lnk[v] = min(lnk[v], tidx[w]);
  }
  if (lnk[v] != tidx[v]) return;
  comps.pb(vi());
  int w;
  do {
    vis[w = st.back()] = false; cnr[w] = ncomps;
    ↪  comps.back().pb(w);
    st.pop_back();
  } while (w != v);
  ncomps++;
}

void findSCC(int n) {
  age = ncomps = 0; vis.assign(n, false);
  ↪  tidx.assign(n, 0);
  lnk.resize(n); cnr.resize(n); comps.clear();
  for (int i = 0; i < n; i++)
    if (tidx[i] == 0) tarjan(i);
}
```

3.4.3. *Dominator graph.*

```
const int N = 1234567;

vi g[N], g_rev[N], bucket[N];
int pos[N], cnt, order[N], parent[N], sdom[N], p[N],
↪  best[N], idom[N], link[N];

void dfs(int v) {
  pos[v] = cnt;
  order[cnt++] = v;
  for (int u : g[v]) {
    if (pos[u] == -1) {
      parent[u] = v;
      dfs(u);
    }
  }
}

int find_best(int x) {
  if (p[x] == x) return best[x];
  int u = find_best(p[x]);
  if (pos[sdom[u]] < pos[sdom[best[x]]])
    best[x] = u;
  p[x] = p[p[x]];
  return best[x];
}

void dominators(int n, int root) {
  fill_n(pos, n, -1);
  cnt = 0;
  dfs(root);
  for (int i = 0; i < n; i++)
    for (int u : g[i]) g_rev[u].push_back(i);
  for (int i = 0; i < n; i++)
    p[i] = best[i] = sdom[i] = i;
  for (int it = cnt - 1; it >= 1; it--) {
    int w = order[it];
    for (int u : g_rev[w]) {
      int t = find_best(u);
      if (pos[sdom[t]] < pos[sdom[w]])
        sdom[w] = sdom[t];
    }
    bucket[sdom[w]].push_back(w);
    idom[w] = sdom[w];
    for (int u : bucket[parent[w]])
      link[u] = find_best(u);
    bucket[parent[w]].clear();
    p[w] = parent[w];
  }
  for (int it = 1; it < cnt; it++) {
    int w = order[it];
    idom[w] = idom[link[w]];
  }
}
```

3.4.4. *2-SAT* $\mathcal{O}(V + E)$. Include findSCC.

```
void init2sat(int n) { adj.assign(2 * n, vi()); }

// vl, vr = true -> variable l, variable r should be
↪  negated.
void imply(int xl, bool vl, int xr, bool vr) {
  adj[2 * xl + vl].pb(2 * xr + vr); adj[2 * xr
  ↪  +!vr].pb(2 * xl +!vl); }

void satOr(int xl, bool vl, int xr, bool vr) {
↪  imply(xl, !vl, xr, vr); }
void satConst(int x, bool v) { imply(x, !v, x, v); }
void satIff(int xl, bool vl, int xr, bool vr) {
  imply(xl, vl, xr, vr); imply(xr, vr, xl, vl);}

bool solve2sat(int n, vector<bool> &sol) {
  findSCC(2 * n);
  for (int i = 0; i < n; i++)
    if (cnr[2 * i] == cnr[2 * i + 1]) return false;
  vector<bool> seen(n, false); sol.assign(n, false);
  for (vi &comp : comps) {
    for (int v : comp) {
      if (seen[v / 2]) continue;
      seen[v / 2] = true; sol[v / 2] = v & 1;
    }
  }
  return true;
}
```

3.5. **Cycle Detection** $\mathcal{O}(V + E)$.

```
vvi adj; // assumes bidirected graph, adjust
↪  accordingly

bool cycle_detection() {
  stack<int> s; vector<bool> vis(MAXN, false); vi
  ↪  par(MAXN, -1); s.push(0);
  vis[0] = true;
  while(!s.empty()) {
    int cur = s.top(); s.pop();
    for(int i : adj[cur]) {
      if(vis[i] && par[cur] != i) return true;
      s.push(i); par[i] = cur; vis[i] = true;
    }
  }
  return false;}
```

3.6. **Maximum Flow Algorithms.**

3.6.1. *Dinic's Algorithm* $\mathcal{O}(V^2E)$.

```
struct Edge { int t; ll c, f; };
struct Dinic {
  vi H, P; vvi E;
  vector<Edge> G;
  Dinic(int n) : H(n), P(n), E(n) {}

  void addEdge(int u, int v, ll c) {
    E[u].pb(G.size()); G.pb({v, c, 0LL});
    E[v].pb(G.size()); G.pb({u, 0LL, 0LL});
  }
  ll dfs(int t, int v, ll f) {
    if (v == t || !f) return f;
    for ( ; P[v] < (int) E[v].size(); P[v]++) {
      int e = E[v][P[v]], w = G[e].t;
      if (H[w] != H[v] + 1) continue;
      ll df = dfs(t, w, min(f, G[e].c - G[e].f));
      if (df > 0) {
        G[e].f += df, G[e ^ 1].f -= df;
        return df;
      }
    } return 0;
  }
  ll maxflow(int s, int t, ll f = 0) {
    while (1) {
      fill(all(H), 0); H[s] = 1;
      queue<int> q; q.push(s);
      while (!q.empty()) {
        int v = q.front(); q.pop();
        for (int w : E[v]) if (G[w].f < G[w].c &&
        ↪  !H[G[w].t])
          H[G[w].t] = H[v] + 1, q.push(G[w].t);
      }
      if (!H[t]) return f;
      fill(all(P), 0);
      while (ll df = dfs(t, s, LLINF)) f += df;
    }
  }
};
```

3.6.2. *Min-cost max-flow* $O(n^2m^2)$. Find the cheapest possible way of sending a certain amount of flow through a flow network.

```
const int maxn = 300;

struct edge { ll x, y, f, c, w; };
ll V, par[maxn], D[maxn]; vector<edge> g;
inline void addEdge(int u, int v, ll c, ll w) {
  g.pb({u, v, 0, c, w});
  g.pb({v, u, 0, 0, -w});
}

void sp(int s, int t) {
  fill_n(D, V, LLINF); D[s] = 0;
  for (int ng = g.size(), _ = V; _--; ) {
    bool ok = false;
    for (int i = 0; i < ng; i++)
      if (D[g[i].x] != LLINF && g[i].f < g[i].c &&
      ↪  D[g[i].x] + g[i].w < D[g[i].y]) {
        D[g[i].y] = D[g[i].x] + g[i].w;
        par[g[i].y] = i; ok = true;
      }
    if (!ok) break;
  }
}

void minCostMaxFlow(int s, int t, ll &c, ll &f) {
  for (c = f = 0; sp(s, t), D[t] < LLINF; ) {
    ll df = LLINF, dc = 0;
    for (int v = t, e; e = par[v], v != s; v =
    ↪  g[e].x) df = min(df, g[e].c - g[e].f);
    for (int v = t, e; e = par[v], v != s; v =
    ↪  g[e].x) g[e].f += df, g[e^1].f -= df, dc +=
    ↪  g[e].w;
    f += df; c += dc * df;
  }
}
```

3.6.3. *Gomory-Hu Tree - All Pairs Maximum Flow.* An implementation of the Gomory-Hu Tree. The spanning tree is constructed using Gusfield's algorithm in $O(|V|^2)$ plus $|V|-1$ times the time it takes to calculate the maximum flow. If Dinic's algorithm is used to calculate the max flow, the running time is $O(|V|^3|E|)$. NOTE: Not sure if it works correctly with disconnected graphs.

```
#include "dinic.cpp"
bool same[MAXV];
pair<vii, vvi> construct_gh_tree(flow_network &g) {
  int n = g.n, v;
  vii par(n, ii(0, 0)); vvi cap(n, vi(n, -1));
  rep(s,1,n) {
    int l = 0, r = 0;
    par[s].second = g.max_flow(s, par[s].first,
    ↪  false);
    memset(d, 0, n * sizeof(int));
    memset(same, 0, n * sizeof(bool));
    d[q[r++] = s] = 1;
    while (l < r) {
      same[v = q[l++]] = true;
      for (int i = g.head[v]; i != -1; i =
      ↪  g.e[i].nxt)
        if (g.e[i].cap > 0 && d[g.e[i].v] == 0)
          d[q[r++] = g.e[i].v] = 1; }
    rep(i,s+1,n)
      if (par[i].first == par[s].first && same[i])
        par[i].first = s;
    g.reset(); }
  rep(i,0,n) {
    int mn = INF, cur = i;
    while (true) {
      cap[cur][i] = mn;
      if (cur == 0) break;
      mn = min(mn, par[cur].second), cur =
      ↪  par[cur].first; } }
  return make_pair(par, cap); }
int compute_max_flow(int s, int t, const pair<vii,
↪  vvi> &gh) {
  int cur = INF, at = s;
  while (gh.second[at][t] == -1)
    cur = min(cur, gh.first[at].second),
    at = gh.first[at].first;
  return min(cur, gh.second[at][t]); }
```

3.7. **Minimal Spanning Tree.**

3.7.1. *Kruskal* $\mathcal{O}(E \log V)$.

```
struct edge { int x, y; ll w; };
ll kruskal(int n, vector<edge> edges) {
  dsu D(n);
  sort(all(edges), [] (edge a, edge b) -> bool {
    return a.w < b.w; });
  ll ret = 0;
  for (edge e : edges) if (D.find(e.x) !=
  ↪  D.find(e.y))
    ret += e.w, D.unite(e.x, e.y);
  return ret;
}
```

3.8. Topological Sort $O(V+E)$.

3.8.1. *Modified Depth-First Search.*

```
void tsort_dfs(int cur, char* color, const vvi& adj,
    stack<int>& res, bool& cyc) {
  color[cur] = 1;
  rep(i,0,size(adj[cur])) {
    int nxt = adj[cur][i];
    if (color[nxt] == 0)
      tsort_dfs(nxt, color, adj, res, cyc);
    else if (color[nxt] == 1)
      cyc = true;
    if (cyc) return; }
  color[cur] = 2;
  res.push(cur); }
vi tsort(int n, vvi adj, bool& cyc) {
  cyc = false;
  stack<int> S;
  vi res;
  char* color = new char[n];
  memset(color, 0, n);
  rep(i,0,n) {
    if (!color[i]) {
      tsort_dfs(i, color, adj, S, cyc);
      if (cyc) return res; } }
  while (!S.empty()) res.push_back(S.top()), S.pop();
  return res; }
```

3.9. Euler Path $O(V+E)$ hopefully.

Finds an Euler Path (or circuit) in a *directed* graph iff one exists.

```
const int MAXV = 1000, MAXE = 5000;
vi adj[MAXV];
int n, m, indeg[MAXV], outdeg[MAXV], res[MAXE + 1];
ii start_end() {
  int start = -1, end = -1, any = 0, c = 0;
  rep(i,0,n) {
    if (outdeg[i] > 0) any = i;
    if (indeg[i] + 1 == outdeg[i]) start = i, c++;
    else if (indeg[i] == outdeg[i] + 1) end = i, c++;
    else if (indeg[i] != outdeg[i]) return ii(-1,-1);
  ↪  }
  if ((start == -1) != (end == -1) || (c != 2 && c !=
  ↪  0))
    return ii(-1,-1);
  if (start == -1) start = end = any;
  return ii(start, end); }
bool euler_path() {
  ii se = start_end();
  int cur = se.first, at = m + 1;
  if (cur == -1) return false;
  stack<int> s;
  while (true) {
    if (outdeg[cur] == 0) {
      res[--at] = cur;
      if (s.empty()) break;
      cur = s.top(); s.pop();
    } else s.push(cur), cur =
    ↪  adj[cur][--outdeg[cur]]; }
  return at == 0; }
```

Finds an Euler *cycle* in a *undirected* graph:

```
const int MAXV = 1000;
multiset<int> adj[MAXV];
list<int> L;
list<int>::iterator euler(int at, int to,
    list<int>::iterator it) {
  if (at == to) return it;
  L.insert(it, at), --it;
  while (!adj[at].empty()) {
    int nxt = *adj[at].begin();
    adj[at].erase(adj[at].find(nxt));
    adj[nxt].erase(adj[nxt].find(at));
    if (to == -1) {
      it = euler(nxt, at, it);
      L.insert(it, at);
      --it;
    } else {
      it = euler(nxt, to, it);
      to = -1; } }
  return it; }
// usage: euler(0,-1,L.begin());
```

3.10. Heavy-Light Decomposition.

```
#include "../data-structures/segment_tree.cpp"
const int ID = 0;
int f(int a, int b) { return a + b; }
struct HLD {
  int n, curhead, curloc;
  vi sz, head, parent, loc;
  vvi adj; segment_tree values;
  HLD(int _n) : n(_n), sz(n, 1), head(n),
                parent(n, -1), loc(n), adj(n) {
    vector<ll> tmp(n, ID); values =
    ↪  segment_tree(tmp); }
  void add_edge(int u, int v) {
    adj[u].push_back(v); adj[v].push_back(u); }
  void update_cost(int u, int v, int c) {
    if (parent[v] == u) swap(u, v); assert(parent[u]
    ↪  == v);
    values.update(loc[u], c); }
  int csz(int u) {
    rep(i,0,size(adj[u])) if (adj[u][i] != parent[u])
      sz[u] += csz(adj[parent[adj[u][i]] = u][i]);
    return sz[u]; }
  void part(int u) {
    head[u] = curhead; loc[u] = curloc++;
    int best = -1;
    rep(i,0,size(adj[u]))
      if (adj[u][i] != parent[u] &&
          (best == -1 || sz[adj[u][i]] > sz[best]))
        best = adj[u][i];
    if (best != -1) part(best);
    rep(i,0,size(adj[u]))
      if (adj[u][i] != parent[u] && adj[u][i] !=
      ↪  best)
        part(curhead = adj[u][i]); }
  void build(int r = 0) {
    curloc = 0, csz(curhead = r), part(r); }
  int lca(int u, int v) {
    vi uat, vat; int res = -1;
    while (u != -1) uat.push_back(u), u =
    ↪  parent[head[u]];
    while (v != -1) vat.push_back(v), v =
    ↪  parent[head[v]];
    u = size(uat) - 1, v = size(vat) - 1;
    while (u >= 0 && v >= 0 && head[uat[u]] ==
    ↪  head[vat[v]])
      res = (loc[uat[u]] < loc[vat[v]] ? uat[u] :
      ↪  vat[v]),
      u--, v--;
    return res; }
  int query_upto(int u, int v) { int res = ID;
    while (head[u] != head[v])
      res = f(res, values.query(loc[head[u]],
      ↪  loc[u]).x),
      u = parent[head[u]];
    return f(res, values.query(loc[v] + 1,
    ↪  loc[u]).x); }
  int query(int u, int v) { int l = lca(u, v);
    return f(query_upto(u, l), query_upto(v, l)); }
    ↪  };
```

3.11. Centroid Decomposition.

```
#define MAXV 100100
#define LGMAXV 20
int jmp[MAXV][LGMAXV],
  path[MAXV][LGMAXV],
  sz[MAXV], seph[MAXV],
  shortest[MAXV];
struct centroid_decomposition {
  int n; vvi adj;
  centroid_decomposition(int _n) : n(_n), adj(n) { }
  void add_edge(int a, int b) {
    adj[a].push_back(b); adj[b].push_back(a); }
  int dfs(int u, int p) {
    sz[u] = 1;
    rep(i,0,size(adj[u]))
      if (adj[u][i] != p) sz[u] += dfs(adj[u][i], u);
    return sz[u]; }
```

```
  void makepaths(int sep, int u, int p, int len) {
    jmp[u][seph[sep]] = sep, path[u][seph[sep]] =
    ↪ len;
    int bad = -1;
    rep(i,0,size(adj[u])) {
      if (adj[u][i] == p) bad = i;
      else makepaths(sep, adj[u][i], u, len + 1);
    }
    if (p == sep)
      swap(adj[u][bad], adj[u].back()),
      ↪ adj[u].pop_back(); }
  void separate(int h=0, int u=0) {
    dfs(u,-1); int sep = u;
    down: iter(nxt,adj[sep])
      if (sz[*nxt] < sz[sep] && sz[*nxt] > sz[u]/2) {
        sep = *nxt; goto down; }
    seph[sep] = h, makepaths(sep, sep, -1, 0);
    rep(i,0,size(adj[sep])) separate(h+1,
    ↪ adj[sep][i]); }
  void paint(int u) {
    rep(h,0,seph[u]+1)
      shortest[jmp[u][h]] = min(shortest[jmp[u][h]],
                                path[u][h]); }
  int closest(int u) {
    int mn = INF/2;
    rep(h,0,seph[u]+1)
      mn = min(mn, path[u][h] + shortest[jmp[u][h]]);
    return mn; } };
```

3.12. Least Common Ancestors, Binary Jumping.

```
const int LOGSZ = 20, SZ = 1 << LOGSZ;
int P[SZ], BP[SZ][LOGSZ];

void initLCA() { // assert P[root] == root
  rep(i, 0, SZ) BP[i][0] = P[i];
  rep(j, 1, LOGSZ) rep(i, 0, SZ)
    BP[i][j] = BP[BP[i][j-1]][j-1];
}

int LCA(int a, int b) {
  if (H[a] > H[b]) swap(a, b);
  int dh = H[b] - H[a], j = 0;
  rep(i, 0, LOGSZ) if (dh & (1 << i)) b = BP[b][i];
  while (BP[a][j] != BP[b][j]) j++;
  while (--j >= 0) if (BP[a][j] != BP[b][j])
    a = BP[a][j], b = BP[b][j];
  return a == b ? a : P[a];
}
```

3.13. Tarjan's Off-line Lowest Common Ancestors Algorithm.

```
#include "../data-structures/union_find.cpp"
struct tarjan_olca {
  int *ancestor;
  vi *adj, answers;
  vii *queries;
  bool *colored;
  union_find uf;
  tarjan_olca(int n, vi *_adj) : adj(_adj), uf(n) {
    colored = new bool[n];
    ancestor = new int[n];
    queries = new vii[n];
    memset(colored, 0, n); }
  void query(int x, int y) {
    queries[x].push_back(ii(y, size(answers)));
    queries[y].push_back(ii(x, size(answers)));
    answers.push_back(-1); }
  void process(int u) {
    ancestor[u] = u;
    rep(i,0,size(adj[u])) {
      int v = adj[u][i];
      process(v);
      uf.unite(u,v);
      ancestor[uf.find(u)] = u; }
    colored[u] = true;
    rep(i,0,size(queries[u])) {
      int v = queries[u][i].first;
      if (colored[v]) {
        answers[queries[u][i].second] =
        ↪ ancestor[uf.find(v)];
      } } } };
```

3.14. Misra-Gries $D+1$-edge coloring.

Finds a $\max_i \deg(i)+1$-edge coloring where there all incident edges have distinct colors. Finding a D-edge coloring is NP-hard.

```
struct Edge { int to, col, rev; };

struct MisraGries {
  int N, K=0; vvi F;
  vector<vector<Edge>> G;

  MisraGries(int n) : N(n), G(n) {}
  // add an undirected edge, NO DUPLICATES ALLOWED
  void addEdge(int u, int v) {
    G[u].pb({v, -1, (int) G[v].size()});
    G[v].pb({u, -1, (int) G[u].size()-1});
  }

  void color(int v, int i) {
    vi fan = { i };
    vector<bool> used(G[v].size());
    used[i] = true;
    for (int j = 0; j < (int) G[v].size(); j++)
      if (!used[j] && G[v][j].col >= 0 &&
      ↪ F[G[v][fan.back()].to][G[v][j].col] < 0)
        used[j] = true, fan.pb(j), j = -1;
    int c = 0; while (F[v][c] >= 0) c++;
    int d = 0; while (F[G[v][fan.back()].to][d] >= 0)
    ↪ d++;
    int w = v, a = d, k = 0, ccol;
    while (true) {
      swap(F[w][c], F[w][d]);
      if (F[w][c] >= 0) G[w][F[w][c]].col = c;
      if (F[w][d] >= 0) G[w][F[w][d]].col = d;
      if (F[w][a^=c^d] < 0) break;
      w = G[w][F[w][a]].to;
    }
    do {
      Edge &e = G[v][fan[k]];
      ccol = F[e.to][d] < 0 ? d : G[v][fan[k+1]].col;
      if (e.col >= 0) F[e.to][e.col] = -1;
      F[e.to][ccol] = e.rev;
      F[v][ccol] = fan[k];
      e.col = G[e.to][e.rev].col = ccol;
      k++;
    } while (ccol != d);
  }
  // finds a K-edge-coloring
  void color() {
    REP(v, N) K = max(K, (int) G[v].size() + 1);
    F = vvi(N, vi(K, -1));
    REP(v, N) for (int i = G[v].size(); i--; )
      if (G[v][i].col < 0) color(v, i);
  }
};
```

3.15. Minimum Mean Weight Cycle.

Given a strongly connected directed graph, finds the cycle of minimum mean weight. If you have a graph that is not strongly connected, run this on each strongly connected component.

```
double
↪ min_mean_cycle(vector<vector<pair<int,double>>>
↪ adj){
  int n = size(adj); double mn = INFINITY;
  vector<vector<double> > arr(n+1, vector<double>(n,
  ↪ mn));
  arr[0][0] = 0;
  rep(k,1,n+1) rep(j,0,n) iter(it,adj[j])
    arr[k][it->first] = min(arr[k][it->first],
                            it->second +
                            ↪ arr[k-1][j]);
  rep(k,0,n) {
    double mx = -INFINITY;
    rep(i,0,n) mx = max(mx,
    ↪ (arr[n][i]-arr[k][i])/(n-k));
    mn = min(mn, mx); }
  return mn; }
```

3.16. Minimum Arborescence.

Given a weighted directed graph, finds a subset of edges of minimum total weight so that there is a unique path from the root r to each vertex. Returns a vector of size n, where the ith element is the edge for the ith vertex. The answer for the root is undefined!

```
#include "../data-structures/union_find.cpp"
struct arborescence {
  int n; union_find uf;
  vector<vector<pair<ii,int> > > adj;
  arborescence(int _n) : n(_n), uf(n), adj(n) { }
  void add_edge(int a, int b, int c) {
    adj[b].push_back(make_pair(ii(a,b),c)); }
  vii find_min(int r) {
    vi vis(n,-1), mn(n,INF); vii par(n);
```

```
    rep(i,0,n) {
      if (uf.find(i) != i) continue;
      int at = i;
      while (at != r && vis[at] == -1) {
        vis[at] = i;
        iter(it,adj[at]) if (it->second < mn[at] &&
            uf.find(it->first.first) != at)
          mn[at] = it->second, par[at] = it->first;
        if (par[at] == ii(0,0)) return vii();
        at = uf.find(par[at].first); }
      if (at == r || vis[at] != i) continue;
      union_find tmp = uf; vi seq;
      do { seq.push_back(at); at =
      ↪  uf.find(par[at].first);
      } while (at != seq.front());
      iter(it,seq) uf.unite(*it,seq[0]);
      int c = uf.find(seq[0]);
      vector<pair<ii,int> > nw;
      iter(it,seq) iter(jt,adj[*it])
        nw.push_back(make_pair(jt->first,
              jt->second - mn[*it]));
      adj[c] = nw;
      vii rest = find_min(r);
      if (size(rest) == 0) return rest;
      ii use = rest[c];
      rest[at = tmp.find(use.second)] = use;
      iter(it,seq) if (*it != at)
        rest[*it] = par[*it];
      return rest; }
    return par; } };
```

3.17. **Blossom algorithm.** Finds a maximum matching in an arbitrary graph in $O(|V|^4)$ time. Be aware of loop edges.

```
#define MAXV 300
bool marked[MAXV], emarked[MAXV][MAXV];
int S[MAXV];
vi find_augmenting_path(const vector<vi> &adj,const
↪  vi &m){
  int n = size(adj), s = 0;
  vi par(n,-1), height(n), root(n,-1), q, a, b;
  memset(marked,0,sizeof(marked));
  memset(emarked,0,sizeof(emarked));
  rep(i,0,n) if (m[i] >= 0) emarked[i][m[i]] = true;
             else root[i] = i, S[s++] = i;
  while (s) {
    int v = S[--s];
    iter(wt,adj[v]) {
      int w = *wt;
      if (emarked[v][w]) continue;
      if (root[w] == -1) {
        int x = S[s++] = m[w];
        par[w]=v, root[w]=root[v],
        ↪  height[w]=height[v]+1;
        par[x]=w, root[x]=root[w],
        ↪  height[x]=height[w]+1;
      } else if (height[w] % 2 == 0) {
        if (root[v] != root[w]) {
          while(v != -1) q.push_back(v), v = par[v];
          reverse(q.begin(), q.end());
          while(w != -1) q.push_back(w), w = par[w];
          return q;
        } else {
          int c = v;
          while(c != -1) a.push_back(c), c = par[c];
          c = w;
          while(c != -1) b.push_back(c), c = par[c];

          ↪  while(!a.empty()&&!b.empty()&&a.back()==b.back())
            c = a.back(), a.pop_back(), b.pop_back();
          memset(marked,0,sizeof(marked));
          fill(par.begin(), par.end(), 0);
          iter(it,a) par[*it] = 1; iter(it,b)
          ↪  par[*it] = 1;
          par[c] = s = 1;
          rep(i,0,n) root[par[i] = par[i] ? 0 : s++]
          ↪  = i;
          vector<vi> adj2(s);
          rep(i,0,n) iter(it,adj[i]) {
            if (par[*it] == 0) continue;
            if (par[i] == 0) {
              if (!marked[par[*it]]) {
                adj2[par[i]].push_back(par[*it]);
                adj2[par[*it]].push_back(par[i]);
                marked[par[*it]] = true; }
            } else adj2[par[i]].push_back(par[*it]);
            ↪  }
          vi m2(s, -1);
          if (m[c] != -1) m2[m2[par[m[c]]] = 0] =
          ↪  par[m[c]];
          rep(i,0,n)
          ↪  if(par[i]!=0&&m[i]!=-1&&par[m[i]]!=0)
            m2[par[i]] = par[m[i]];
          vi p = find_augmenting_path(adj2, m2);
          int t = 0;
          while (t < size(p) && p[t]) t++;
          if (t == size(p)) {
            rep(i,0,size(p)) p[i] = root[p[i]];
            return p; }
          if (!p[0] || (m[c] != -1 && p[t+1] !=
          ↪  par[m[c]]))
            reverse(p.begin(), p.end()), t =
            ↪  size(p)-t-1;
          rep(i,0,t) q.push_back(root[p[i]]);
          iter(it,adj[root[p[t-1]]]) {
            if (par[*it] != (s = 0)) continue;
            a.push_back(c), reverse(a.begin(),
            ↪  a.end());
            iter(jt,b) a.push_back(*jt);
            while (a[s] != *it) s++;
            if ((height[*it] & 1) ^ (s < size(a) -
            ↪  size(b)))
              reverse(a.begin(), a.end()), s =
              ↪  size(a)-s-1;
              ↪  while(a[s]!=c)q.push_back(a[s]),s=(s+1)%size
              q.push_back(c);
              rep(i,t+1,size(p))
              ↪  q.push_back(root[p[i]]);
              return q; } } }
        emarked[v][w] = emarked[w][v] = true; }
      marked[v] = true; } return q; }
vii max_matching(const vector<vi> &adj) {
  vi m(size(adj), -1), ap; vii res, es;
  rep(i,0,size(adj)) iter(it,adj[i])
  ↪  es.emplace_back(i,*it);
  random_shuffle(es.begin(), es.end());
  iter(it,es) if (m[it->first] == -1 && m[it->second]
  ↪  == -1)
    m[it->first] = it->second, m[it->second] =
    ↪  it->first;
  do { ap = find_augmenting_path(adj, m);
       rep(i,0,size(ap)) m[m[ap[i^1]] = ap[i]] =
       ↪  ap[i^1];
  } while (!ap.empty());
  rep(i,0,size(m)) if (i < m[i]) res.emplace_back(i,
  ↪  m[i]);
  return res; }
```

3.18. **Maximum Density Subgraph.** Given (weighted) undirected graph G. Binary search density. If g is current density, construct flow network: (S, u, m), $(u, T, m+2g-d_u)$, $(u, v, 1)$, where m is a large constant (larger than sum of edge weights). Run floating-point max-flow. If minimum cut has empty S-component, then maximum density is smaller than g, otherwise it's larger. Distance between valid densities is at least $1/(n(n-1))$. Edge case when density is 0. This also works for weighted graphs by replacing d_u by the weighted degree, and doing more iterations (if weights are not integers).

3.19. **Maximum-Weight Closure.** Given a vertex-weighted directed graph G. Turn the graph into a flow network, adding weight ∞ to each edge. Add vertices S, T. For each vertex v of weight w, add edge (S, v, w) if $w \geq 0$, or edge $(v, T, -w)$ if $w < 0$. Sum of positive weights minus minimum $S-T$ cut is the answer. Vertices reachable from S are in the closure. The maximum-weight closure is the same as the complement of the minimum-weight closure on the graph with edges reversed.

3.20. **Maximum Weighted Independent Set in a Bipartite Graph.** This is the same as the minimum weighted vertex cover. Solve this by constructing a flow network with edges $(S, u, w(u))$ for $u \in L$, $(v, T, w(v))$ for $v \in R$ and (u, v, ∞) for $(u, v) \in E$. The minimum S,T-cut is the answer. Vertices adjacent to a cut edge are in the vertex cover.

3.21. **Synchronizing word problem.** A DFA has a synchronizing word (an input sequence that moves all states to the same state) iff. each pair of states has a synchronizing word. That can be checked using reverse DFS over pairs of states. Finding the shortest synchronizing word is NP-complete.

4. String algorithms

4.1. Trie.

```
const int SIGMA = 26;

struct trie {
  bool word; trie **adj;

  trie() : word(false), adj(new trie*[SIGMA]) {
    for (int i = 0; i < SIGMA; i++) adj[i] = NULL;
  }

  void addWord(const string &str) {
    trie *cur = this;
    for (char ch : str) {
      int i = ch - 'a';
      if (!cur->adj[i]) cur->adj[i] = new trie();
      cur = cur->adj[i];
    }
    cur->word = true;
  }

  bool isWord(const string &str) {
    trie *cur = this;
    for (char ch : str) {
      int i = ch - 'a';
      if (!cur->adj[i]) return false;
      cur = cur->adj[i];
    }
    return cur->word;
  }
};
```

4.2. Z-algorithm $\mathcal{O}(n)$.

```
// z[i] = length of longest substring starting from
↪  s[i] which is also a prefix of s.
vi z_function(const string &s) {
  int n = (int) s.length();
  vi z(n);
  for (int i = 1, l = 0, r = 0; i < n; ++i) {
    if (i <= r) z[i] = min (r - i + 1, z[i - l]);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]])
    ↪  ++z[i];
    if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
  }
  return z;
}
```

4.3. Suffix array $\mathcal{O}(n \log^2 n)$.

This creates an array $P[0], P[1], \ldots, P[n-1]$ such that the suffix $S[i \ldots n]$ is the $P[i]^{th}$ suffix of S when lexicographically sorted.

```
typedef pair<ii, int> tii;
const int maxlogn = 17, maxn = 1 << maxlogn;

int p[maxlogn + 1][maxn]; tii L[maxn];

int suffixArray(string S) {
  int N = S.size(), stp = 1, cnt = 1;
  REP(i, N) p[0][i] = S[i];
  for (; cnt < N; stp++, cnt <<= 1) {
    REP(i, N)
      L[i] = tii(ii(p[stp-1][i], i + cnt < N ?
      ↪  p[stp-1][i + cnt] : -1), i);
    sort(L, L + N);
    REP(i, N)
      p[stp][L[i].y] = i > 0 && L[i].x == L[i-1].x ?
      ↪  p[stp][L[i-1].y] : i;
  }
  return stp - 1; // result is in p[stp - 1][0 .. (N
  ↪  - 1)]
}
```

4.4. Longest Common Subsequence $\mathcal{O}(n^2)$.

Substring: *consecutive characters* !!!

```
int dp[STR_SIZE][STR_SIZE]; // DP problem

int lcs(const string &w1, const string &w2) {
  int n1 = w1.size(), n2 = w2.size();
  for (int i = 0; i < n1; i++) {
    for (int j = 0; j < n2; j++) {
      if (i == 0 || j == 0) dp[i][j] = 0;
      else if (w1[i - 1] == w2[j - 1]) dp[i][j] =
      ↪  dp[i - 1][j - 1] + 1;
      else dp[i][j] = max(dp[i - 1][j], dp[i][j -
      ↪  1]);
    }
  }
  return dp[n1][n2];
}

// backtrace
string getLCS(const string &w1, const string &w2) {
  int i = w1.size(), j = w2.size(); string ret = "";
  while (i > 0 && j > 0) {
    if (w1[i - 1] == w2[j - 1]) ret += w1[--i], j--;
    else if (dp[i][j - 1] > dp[i - 1][j]) j--;
    else i--;
  }
  reverse(ret.begin(), ret.end());
  return ret;
}
```

4.5. Levenshtein Distance $\mathcal{O}(n^2)$.

Minimal number of insertions, removals and edits required to transform one string in the other.

```
int dp[MAX_SIZE][MAX_SIZE]; // DP problem

int levDist(const string &w1, const string &w2) {
  int n1 = w1.size(), n2 = w2.size();
  for (int i = 0; i <= n1; i++) dp[i][0] = i; //
  ↪  removal
  for (int j = 0; j <= n2; j++) dp[0][j] = j; //
  ↪  insertion
  for (int i = 1; i <= n1; i++)
    for (int j = 1; j <= n2; j++)
      dp[i][j] = min(
        1 + min(dp[i - 1][j], dp[i][j - 1]),
        dp[i - 1][j - 1] + (w1[i - 1] != w2[j - 1])
      );
  return dp[n1][n2];
}
```

4.6. Knuth-Morris-Pratt algorithm $\mathcal{O}(N + M)$.

```
int kmp_search(const string &word, const string
↪  &text) {
  int n = word.size();
  vi T(n + 1, 0);
  for (int i = 1, j = 0; i < n; ) {
    if (word[i] == word[j]) T[++i] = ++j; // match
    else if (j > 0) j = T[j]; // fallback
    else i++; // no match, keep zero
  }
  int matches = 0;
  for (int i = 0, j = 0; i < text.size(); ) {
    if (text[i] == word[j]) {
      i++;
      if (++j == n) // match at interval [i - n, i)
        matches++, j = T[j];
    } else if (j > 0) j = T[j];
    else i++;
  }
  return matches;
}
```

4.7. Aho-Corasick Algorithm $\mathcal{O}(N + \sum_{i=1}^{m} |S_i|)$.

Dictionary substring matching as automaton. All given P must be unique!

```
const int MAXP = 100, MAXLEN = 200, SIGMA = 26,
↪  MAXTRIE = MAXP * MAXLEN;

int nP;
string P[MAXP], S;

int pnr[MAXTRIE], to[MAXTRIE][SIGMA], sLink[MAXTRIE],
↪  dLink[MAXTRIE], nnodes;

void ahoCorasick() {
  fill_n(pnr, MAXTRIE, -1);
  for (int i = 0; i < MAXTRIE; i++) fill_n(to[i],
  ↪  SIGMA, 0);
  fill_n(sLink, MAXTRIE, 0); fill_n(dLink, MAXTRIE,
  ↪  0);
  nnodes = 1;
  // STEP 1: MAKE A TREE
  for (int i = 0; i < nP; i++) {
    int cur = 0;
    for (char c : P[i]) {
      int i = c - 'a';
      if (to[cur][i] == 0) to[cur][i] = nnodes++;
      cur = to[cur][i];
    }
    pnr[cur] = i;
  }
  // STEP 2: CREATE SUFFIX_LINKS AND DICT_LINKS
```

```
  queue<int> q; q.push(0);
  while (!q.empty()) {
    int cur = q.front(); q.pop();
    for (int c = 0; c < SIGMA; c++) {
      if (to[cur][c]) {
        int sl = sLink[to[cur][c]] = cur == 0 ? 0 :
        ↪  to[sLink[cur]][c];
        // if all strings have equal length, remove
        ↪  this:
        dLink[to[cur][c]] = pnr[sl] >= 0 ? sl :
        ↪  dLink[sl];
        q.push(to[cur][c]);
      } else to[cur][c] = to[sLink[cur]][c];
    }
  }
  // STEP 3: TRAVERSE S
  for (int cur = 0, i = 0, n = S.size(); i < n; i++)
  ↪  {
    cur = to[cur][S[i] - 'a'];
    for (int hit = pnr[cur] >= 0 ? cur : dLink[cur];
    ↪  hit; hit = dLink[hit]) {
      cerr << P[pnr[hit]] << " found at [" << (i + 1
      ↪  - P[pnr[hit]].size()) << ", " << i << "]"
      ↪  << endl;
    }
  }
}
```

4.8. **eerTree.** Constructs an eerTree in $O(n)$, one character at a time.

```
#define MAXN 100100
#define SIGMA 26
#define BASE 'a'
char *s = new char[MAXN];
struct state {
  int len, link, to[SIGMA];
} *st = new state[MAXN+2];
struct eertree {
  int last, sz, n;
  eertree() : last(1), sz(2), n(0) {
    st[0].len = st[0].link = -1;
    st[1].len = st[1].link = 0; }
  int extend() {
    char c = s[n++]; int p = last;
    while (n - st[p].len - 2 < 0 || c != s[n -
    ↪  st[p].len - 2])
      p = st[p].link;
    if (!st[p].to[c-BASE]) {
      int q = last = sz++;
      st[p].to[c-BASE] = q;
      st[q].len = st[p].len + 2;
      do { p = st[p].link;
      } while (p != -1 && (n < st[p].len + 2 ||
               c != s[n - st[p].len - 2]));
      if (p == -1) st[q].link = 1;
      else st[q].link = st[p].to[c-BASE];
      return 1; }
    last = st[p].to[c-BASE];
    return 0; } };
```

4.9. **Suffix Automaton.** Minimum automata that accepts all suffixes of a string with $O(n)$ construction. The automata itself is a DAG therefore suitable for DP, examples are counting unique substrings, occurrences of substrings and suffix.

```
// TODO: Add longest common subsring
const int MAXL = 100000;
struct suffix_automaton {
  vi  len, link, occur, cnt;
  vector<map<char,int> > next;
  vector<bool> isclone;
  ll *occuratleast;
  int sz, last;
  string s;
  suffix_automaton() : len(MAXL*2), link(MAXL*2),
    occur(MAXL*2), next(MAXL*2), isclone(MAXL*2) {
    ↪  clear(); }
  void clear() { sz = 1; last = len[0] = 0; link[0] =
  ↪  -1;
                 next[0].clear(); isclone[0] = false;
                 ↪  }
  bool issubstr(string other){
    for(int i = 0, cur = 0; i < size(other); ++i){
      if(cur == -1) return false; cur =
      ↪  next[cur][other[i]]; }
    return true; }
  void extend(char c){ int cur = sz++; len[cur] =
  ↪  len[last]+1;
    next[cur].clear(); isclone[cur] = false; int p =
    ↪  last;
    for(; p != -1 && !next[p].count(c); p = link[p])
      next[p][c] = cur;
    if(p == -1){ link[cur] = 0; }
    else{ int q = next[p][c];
      if(len[p] + 1 == len[q]){ link[cur] = q; }
      else { int clone = sz++; isclone[clone] = true;
        len[clone] = len[p] + 1;
        link[clone] = link[q]; next[clone] = next[q];
        for(; p != -1 && next[p].count(c) &&
        ↪  next[p][c] == q;
              p = link[p]){
          next[p][c] = clone; }
        link[q] = link[cur] = clone;
    } } last = cur; }
  void count(){
    cnt=vi(sz, -1); stack<ii> S; S.push(ii(0,0));
    map<char,int>::iterator i;
    while(!S.empty()){
      ii cur = S.top(); S.pop();
      if(cur.second){
        for(i = next[cur.first].begin();
            i != next[cur.first].end();++i){
          cnt[cur.first] += cnt[(*i).second]; } }
      else if(cnt[cur.first] == -1){
        cnt[cur.first] = 1; S.push(ii(cur.first, 1));
        for(i = next[cur.first].begin();
            i != next[cur.first].end();++i){
          S.push(ii((*i).second, 0)); } } } }
  string lexicok(ll k){
    int st = 0; string res; map<char,int>::iterator
    ↪  i;
    while(k){
      for(i = next[st].begin(); i != next[st].end();
      ↪  ++i){
        if(k <= cnt[(*i).second]){ st = (*i).second;
          res.push_back((*i).first); k--; break;
        } else { k -= cnt[(*i).second]; } } }
    return res; }
  void countoccur(){
    for(int i = 0; i < sz; ++i){ occur[i] = 1 -
    ↪  isclone[i]; }
    vii states(sz);
    for(int i = 0; i < sz; ++i){ states[i] =
    ↪  ii(len[i],i); }
    sort(states.begin(), states.end());
    for(int i = size(states)-1; i >= 0; --i){
      int v = states[i].second;
      if(link[v] != -1) { occur[link[v]] += occur[v];
      ↪  }}}};
```

4.10. **Hashing.** Modulus should be a large prime. Can also use multiple instances with different moduli to minimize chance of collision.

```
struct hasher { int b = 311, m; vi h, p;
  hasher(string s, int _m)
    : m(_m), h(size(s)+1), p(size(s)+1) {
    p[0] = 1; h[0] = 0;
    rep(i,0,size(s)) p[i+1] = (ll)p[i] * b % m;
    rep(i,0,size(s)) h[i+1] = ((ll)h[i] * b + s[i]) %
    ↪  m; }
  int hash(int l, int r) {
    return (h[r+1] + m - (ll)h[l] * p[r-l+1] % m) %
    ↪  m; } };
```

5. Geometry

```
const double EPS = 1e-7, PI = acos(-1.0);

typedef long long NUM; // EITHER double OR long long
typedef pair<NUM, NUM> pt;
#define x first
#define y second

pt operator+(pt p,pt q){return pt(p.x+q.x, p.y+q.y);}
pt operator-(pt p,pt q){return pt(p.x-q.x, p.y-q.y);}

pt& operator+=(pt &p, pt q) { return p = p+q; }
pt& operator-=(pt &p, pt q) { return p = p-q; }

pt operator*(pt p,NUM l) { return pt(p.x*l, p.y*l); }
pt operator/(pt p,NUM l) { return pt(p.x/l, p.y/l); }

NUM operator*(pt p, pt q) { return p.x*q.x+p.y*q.y; }
NUM operator^(pt p, pt q) { return p.x*q.y-p.y*q.x; }
```

```
NUM lenSq(pt p) { return p * p; }
NUM lenSq(pt p, pt q) { return lenSq(p - q); }
double len(pt p) { return hypot(p.x, p.y); }
double len(pt p, pt q) { return len(p - q); }

typedef pt frac;
typedef pair<double, double> vec;
vec getvec(pt p, pt dp, frac t) { return vec(p.x + 1. * dp.x * t.x / t.y, p.y + 1. * dp.y * t.x / t.y); }

// square distance from pt a to line bc
frac distPtLineSq(pt a, pt b, pt c) {
  a -= b, c -= b;
  return frac((a ^ c) * (a ^ c), c * c);
}

// square distance from pt a to linesegment bc
frac distPtSegmentSq(pt a, pt b, pt c) {
  a -= b; c -= b;
  NUM dot = a * c, len = c * c;
  if (dot <= 0) return frac(a * a, 1);
  if (dot >= len) return frac((a - c) * (a - c), 1);
  return frac(a * a * len - dot * dot, len);
}

// projects pt a onto linesegment bc
frac proj(pt a, pt b, pt c) { return frac((a - b) * (c - b), (c - b) * (c - b)); }
vec projv(pt a, pt b, pt c) { return getvec(b, c - b, proj(a, b, c)); }

bool collinear(pt a, pt b, pt c) { return ((a - b) ^ (a - c)) == 0; }

// true => 1 intersection, false => parallel, so 0 or \infty solutions
bool linesIntersect(pt a, pt b, pt c, pt d) { return ((a - b) ^ (c - d)) != 0; }
vec lineLineIntersection(pt a, pt b, pt c, pt d) {
  double det = (a - b) ^ (c - d); pt ret = (c - d) * (a ^ b) - (a - b) * (c ^ d);
  return vec(ret.x / det, ret.y / det);
}

// dp, dq are directions from p, q
// intersection at p + t_i dp, for 0 <= i < return value
int segmentIntersection(pt p, pt dp, pt q, pt dq, frac &t0, frac &t1){
  if (dp * dp == 0) swap(p, q), swap(dp, dq); // dq = 0
  if (dp * dp == 0) { t0 = t1 = frac(0, 1); return p == q; } // dp = dq = 0
  pt dpq = (q - p); NUM c = dp ^ dq, c0 = dpq ^ dp, c1 = dpq ^ dq;
  if (c == 0) { // parallel, dp > 0, dq >= 0
    if (c0 != 0) return 0; // not collinear
    NUM v0 = dpq * dp, v1 = v0 + dq * dp, dp2 = dp * dp;
    if (v1 < v0) swap(v0, v1);
    t0 = frac(v0 = max(v0, (NUM) 0), dp2);
    t1 = frac(v1 = min(v1, dp2), dp2);
    return (v0 <= v1) + (v0 < v1);
  } else if (c < 0) c = -c, c0 = -c0, c1 = -c1;
  t0 = t1 = frac(c1, c);
  return 0 <= min(c0, c1) && max(c0, c1) <= c;
}

// Returns TWICE the area of a polygon to keep it an integer
NUM polygonTwiceArea(const vector<pt> &pts) {
  NUM area = 0;
  for (int N = pts.size(), i = 0, j = N - 1; i < N; j = i++)
    area += pts[i] ^ pts[j];
  return abs(area); // area < 0 <=> pts ccw
}

bool segmenthaspt(pt s, pt e, pt p) {
  pt ds = p-s, de = p-e;
  return (ds ^ de) == 0LL && (ds * de) <= 0LL;
}

bool insidePolygon(const vector<pt> &pts, pt p, bool strict = true) {
  int n = 0;
  for (int N = pts.size(), i = 0, j = N - 1; i < N; j = i++) {
    // if p is on edge of polygon
    if (segmenthaspt(pts[i], pts[j], p)) return !strict;
    // or: if(distPtSegmentSq(p, pts[i], pts[j]) <= EPS) return !strict;

    // increment n if segment intersects line from p
    n += (max(pts[i].y, pts[j].y) > p.y && min(pts[i].y, pts[j].y) <= p.y &&
      (((pts[j] - pts[i])^(p-pts[i])) > 0) == (pts[i].y <= p.y));
  }
  return n & 1; // inside if odd number of intersections
}
```

5.1. Convex Hull $\mathcal{O}(n \log n)$.

```
// the convex hull consists of: { pts[ret[0]], pts[ret[1]], ... pts[ret.back()] }
vi convexHull(const vector<pt> &pts) {
  if (pts.empty()) return vi();
  vi ret, ord;
  int n = pts.size(), st = min_element(all(pts)) - pts.begin();
  rep(i, 0, n)
    if (pts[i] != pts[st]) ord.pb(i);
  sort(all(ord), [&pts,&st] (int a, int b) {
    pt p = pts[a] - pts[st], q = pts[b] - pts[st];
    return (p ^ q) != 0 ? (p ^ q) > 0 : lenSq(p) < lenSq(q);
  });
  ret.pb(st);
  for (int i : ord) {
    // use '>' to include ALL points on the hull-line
    for (int s = ret.size() - 1; s > 0 && ((pts[ret[s-1]] - pts[ret[s]]) ^ (pts[i] - pts[ret[s]])) >= 0; s--)
      ret.pop_back();
    ret.pb(i);
  }
  return ret;
}
```

5.2. Rotating Calipers $\mathcal{O}(n)$.

Finds the longest distance between two points in a convex hull.

```
NUM rotatingCalipers(vector<pt> &hull) {
  int n = hull.size(), a = 0, b = 1;
  if (n <= 1) return 0.0;
  while (((hull[1] - hull[0]) ^ (hull[(b + 1) % n] - hull[b])) > 0) b++;
  NUM ret = 0.0;
  while (a < n) {
    ret = max(ret, lenSq(hull[a], hull[b]));
    if (((hull[(a + 1) % n] - hull[a % n]) ^ (hull[(b + 1) % n] - hull[b])) <= 0) a++;
    else if (++b == n) b = 0;
  }
  return ret;
}
```

5.3. Closest points $\mathcal{O}(n \log n)$.

```
int n; pt pts[maxn];

struct byY {
  bool operator()(int a, int b) const { return pts[a].y < pts[b].y; }
};

inline NUM dist(ii p) { return hypot(pts[p.x].x - pts[p.y].x, pts[p.x].y - pts[p.y].y); }

ii minpt(ii p1, ii p2) { return dist(p1) < dist(p2) ? p1 : p2; }

// closest pts (by index) inside pts[l ... r], with sorted y values in ys
ii closest(int l, int r, vi &ys) {
  if (r - l == 2) { // don't assume 1 here.
    ys = { l, l + 1 };
    return ii(l, l + 1);
```

```
  } else if (r - l == 3) { // brute-force
    ys = { l, l + 1, l + 2 };
    sort(all(ys), byY());
    return minpt(ii(l, l + 1), minpt(ii(l, l + 2),
     ↪ ii(l + 1, l + 2)));
  }
  int m = (l + r) / 2; vi yl, yr;
  ii delta = minpt(closest(l, m, yl), closest(m, r,
  ↪ yr));
  NUM ddelta = dist(delta), xm = .5 * (pts[m-1].x +
  ↪ pts[m].x);
  merge(all(yl), all(yr), back_inserter(ys), byY());
  deque<int> q;
  for (int i : ys) if (abs(pts[i].x - xm) <= ddelta)
  ↪ {
    for (int j : q) delta = minpt(delta, ii(i, j));
    q.pb(i);
    if (q.size() > 8) q.pop_front(); // magic from
     ↪ Introduction to Algorithms.
  }
  return delta;
}
```

5.4. **Great-Circle Distance.** Computes the distance between two points (given as latitude/longitude coordinates) on a sphere of radius r.

```
ld gc_distance(ld pLat, ld pLong, ld qLat, ld qLong,
↪ ld r) {
  pLat *= pi / 180; pLong *= pi / 180;
  qLat *= pi / 180; qLong *= pi / 180;
  return r * acos(cos(pLat)*cos(qLat)*cos(pLong -
  ↪ qLong) + sin(pLat)*sin(qLat)); }
```

5.5. **3D Primitives.**

```
#define P(p) const point3d &p
#define L(p0, p1) P(p0), P(p1)
#define PL(p0, p1, p2) P(p0), P(p1), P(p2)
struct point3d {
  double x, y, z;
  point3d() : x(0), y(0), z(0) {}
  point3d(double _x, double _y, double _z)
    : x(_x), y(_y), z(_z) {}
  point3d operator+(P(p)) const {
    return point3d(x + p.x, y + p.y, z + p.z); }
  point3d operator-(P(p)) const {
    return point3d(x - p.x, y - p.y, z - p.z); }
  point3d operator-() const {
    return point3d(-x, -y, -z); }
  point3d operator*(double k) const {
    return point3d(x * k, y * k, z * k); }
  point3d operator/(double k) const {
    return point3d(x / k, y / k, z / k); }
  double operator%(P(p)) const {
    return x * p.x + y * p.y + z * p.z; }
  point3d operator*(P(p)) const {
    return point3d(y*p.z - z*p.y,
                   z*p.x - x*p.z, x*p.y - y*p.x); }
  double length() const {
    return sqrt(*this % *this); }
  double distTo(P(p)) const {
    return (*this - p).length(); }
  double distTo(P(A), P(B)) const {
    // A and B must be two different points
    return ((*this - A) * (*this - B)).length() /
    ↪ A.distTo(B);}
  point3d normalize(double k = 1) const {
    // length() must not return 0
    return (*this) * (k / length()); }
  point3d getProjection(P(A), P(B)) const {
    point3d v = B - A;
    return A + v.normalize((v % (*this - A)) /
    ↪ v.length()); }
  point3d rotate(P(normal)) const {
    //normal must have length 1 and be orthogonal to
    ↪ the vector
    return (*this) * normal; }
  point3d rotate(double alpha, P(normal)) const {
    return (*this) * cos(alpha) + rotate(normal) *
    ↪ sin(alpha);}
  point3d rotatePoint(P(O), P(axe), double alpha)
  ↪ const{
    point3d Z = axe.normalize(axe % (*this - O));
    return O + Z + (*this - O - Z).rotate(alpha, O);
    ↪ }
  bool isZero() const {
    return abs(x) < EPS && abs(y) < EPS && abs(z) <
    ↪ EPS; }
  bool isOnLine(L(A, B)) const {
    return ((A - *this) * (B - *this)).isZero(); }
  bool isInSegment(L(A, B)) const {
    return isOnLine(A, B) && ((A - *this) % (B -
    ↪ *this))<EPS;}
  bool isInSegmentStrictly(L(A, B)) const {
    return isOnLine(A, B) && ((A - *this) % (B -
    ↪ *this))<-EPS;}
  double getAngle() const {
    return atan2(y, x); }
  double getAngle(P(u)) const {
    return atan2((*this * u).length(), *this % u); }
  bool isOnPlane(PL(A, B, C)) const {
    return
      abs((A - *this) * (B - *this) % (C - *this)) <
      ↪ EPS; } };
int line_line_intersect(L(A, B), L(C, D), point3d
↪ &O){
  if (abs((B - A) * (C - A) % (D - A)) > EPS) return
  ↪ 0;
  if (((A - B) * (C - D)).length() < EPS)
    return A.isOnLine(C, D) ? 2 : 0;
  point3d normal = ((A - B) * (C - B)).normalize();
  double s1 = (C - A) * (D - A) % normal;
  O = A + ((B - A) / (s1 + ((D - B) * (C - B) %
  ↪ normal))) * s1;
  return 1; }
int line_plane_intersect(L(A, B), PL(C, D, E),
↪ point3d & O) {
  double V1 = (C - A) * (D - A) % (E - A);
  double V2 = (D - B) * (C - B) % (E - B);
  if (abs(V1 + V2) < EPS)
    return A.isOnPlane(C, D, E) ? 2 : 0;
  O = A + ((B - A) / (V1 + V2)) * V1;
  return 1; }
bool plane_plane_intersect(P(A), P(nA), P(B), P(nB),
    point3d &P, point3d &Q) {
  point3d n = nA * nB;
  if (n.isZero()) return false;
  point3d v = n * nA;
  P = A + (n * nA) * ((B - A) % nB / (v % nB));
  Q = P + n;
  return true; }
```

5.6. **Polygon Centroid.**

$$C_x = \frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

$$C_y = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

$$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)$$

5.7. **Rectilinear Minimum Spanning Tree.** Given a set of n points in the plane, and the aim is to find a minimum spanning tree connecting these n points, assuming the Manhattan distance is used. The function `candidates` returns at most $4n$ edges that are a superset of the edges in a minimum spanning tree, and then one can use Kruskal's algorithm.

```
#define MAXN 100100
struct RMST {
  struct point {
    int i; ll x, y;
    point() : i(-1) { }
    ll d1() { return x + y; }
    ll d2() { return x - y; }
    ll dist(point other) {
      return abs(x - other.x) + abs(y - other.y); }
    bool operator <(const point &other) const {
      return y == other.y ? x > other.x : y <
      ↪ other.y; }
  } best[MAXN], arr[MAXN], tmp[MAXN];
  int n;
  RMST() : n(0) {}
  void add_point(int x, int y) {
    arr[arr[n].i = n].x = x, arr[n++].y = y; }
  void rec(int l, int r) {
    if (l >= r) return;
    int m = (l+r)/2;
    rec(l,m), rec(m+1,r);
    point bst;
```

```
    for (int i = l, j = m+1, k = l; i <= m || j <= r;
    ↪  k++) {
      if (j > r || (i <= m && arr[i].d1() <
      ↪  arr[j].d1())) {
        tmp[k] = arr[i++];
        if (bst.i != -1 && (best[tmp[k].i].i == -1
                        || best[tmp[k].i].d2() <
                        ↪  bst.d2()))
          best[tmp[k].i] = bst;
      } else {
        tmp[k] = arr[j++];
        if (bst.i == -1 || bst.d2() < tmp[k].d2())
          bst = tmp[k]; } }
    rep(i,l,r+1) arr[i] = tmp[i]; }
  vector<pair<ll,ii> > candidates() {
    vector<pair<ll, ii> > es;
    rep(p,0,2) {
      rep(q,0,2) {
        sort(arr, arr+n);
        rep(i,0,n) best[i].i = -1;
        rec(0,n-1);
        rep(i,0,n) {
          if(best[arr[i].i].i != -1)

            ↪  es.push_back({arr[i].dist(best[arr[i].i]),
                        {arr[i].i,
                        ↪  best[arr[i].i].i}});
          swap(arr[i].x, arr[i].y);
          arr[i].x *= -1, arr[i].y *= -1; } }
      rep(i,0,n) arr[i].x *= -1; }
    return es; } };
```

5.8. **Formulas.** Let $a = (a_x, a_y)$ and $b = (b_x, b_y)$ be two-dimensional vectors.

- $a \cdot b = |a||b| \cos\theta$, where θ is the angle between a and b.
- $a \times b = |a||b| \sin\theta$, where θ is the signed angle between a and b.
- $a \times b$ is equal to the area of the parallelogram with two of its sides formed by a and b. Half of that is the area of the triangle formed by a and b.
- **Euler's formula:** $V - E + F = 2$
- Side lengths a, b, c can form a triangle iff. $a + b > c$, $b + c > a$ and $a + c > b$.
- Sum of internal angles of a regular convex n-gon is $(n-2)\pi$.
- **Law of sines:** $\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$
- **Law of cosines:** $b^2 = a^2 + c^2 - 2ac\cos B$
- Internal tangents of circles $(c_1, r_1), (c_2, r_2)$ intersect at $(c_1 r_2 + c_2 r_1)/(r_1 + r_2)$, external intersect at $(c_1 r_2 - c_2 r_1)/(r_1 + r_2)$.

6. Miscellaneous

6.1. **Binary search** $\mathcal{O}(\log(hi - lo))$.

```
bool test(int n);

int search(int lo, int hi) {
  assert(test(lo) && !test(hi)); // BE CERTAIN
  while (hi - lo > 1) {
    int m = (lo + hi) / 2;
    (test(m) ? lo : hi) = m;
  }
  // assert(test(lo) && !test(hi));
  return lo;
}
```

6.2. **Fast Fourier Transform** $\mathcal{O}(n \log n)$. Given two polynomials $A(x) = a_0 + a_1 x + \cdots + a_{n/2} x^{n/2}$ and $B(x) = b_0 + b_1 x + \cdots + b_{n/2} x^{n/2}$, FFT calculates all coefficients of $C(x) = A(x) \cdot B(x) = c_0 + c_1 x + \ldots c_n x^n$, with $c_i = \sum_{j=0}^{i} a_j b_{i-j}$.

```
typedef complex<double> cpx;
const int LOGN = 19, MAXN = 1 << LOGN;

int rev[MAXN];
cpx rt[MAXN], a[MAXN] = {}, b[MAXN] = {};

void fft(cpx *A) {
  REP(i, MAXN) if (i < rev[i]) swap(A[i], A[rev[i]]);
  for (int k = 1; k < MAXN; k *= 2)
    for (int i = 0; i < MAXN; i += 2*k) REP(j, k) {
        cpx t = rt[j + k] * A[i + j + k];
        A[i + j + k] = A[i + j] - t;
        A[i + j] += t;
      }
}

void multiply() { // a = convolution of a * b
  rev[0] = 0; rt[1] = cpx(1, 0);
  REP(i, MAXN) rev[i] = (rev[i/2] | (i&1)<<LOGN)/2;
  for (int k = 2; k < MAXN; k *= 2) {
    cpx z(cos(PI/k), sin(PI/k));
    rep(i, k/2, k) rt[2*i]=rt[i], rt[2*i+1]=rt[i]*z;
  }
  fft(a); fft(b);
  REP(i, MAXN) a[i] *= b[i] / (double)MAXN;
  reverse(a+1,a+MAXN); fft(a);
}
```

6.3. **Minimum Assignment (Hungarian Algorithm)** $\mathcal{O}(n^3)$.

```
int a[MAXN + 1][MAXM + 1]; // matrix, 1-based
int minimum_assignment(int n, int m) { // n rows, m
↪  columns
  vi 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;
    vi minv(m + 1, INF);
    vector<char> 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);
  }
  // column j is assigned to row p[j]
  return -v[0];
}
```

6.4. **Partial linear equation solver** $\mathcal{O}(N^3)$.

```
typedef double NUM;
const int MAXROWS = 200, MAXCOLS = 200;
const NUM EPS = 1e-5;

// F2: bitset<MAXCOLS+1> mat[MAXROWS];
↪  bitset<MAXROWS> vals;
NUM mat[MAXROWS][MAXCOLS + 1], vals[MAXCOLS]; bool
↪  hasval[MAXCOLS];
bool is0(NUM a) { return -EPS < a && a < EPS; }

// finds x such that Ax = b
// A_ij is mat[i][j], b_i is mat[i][m]
int solvemat(int n, int m) {
  // F2: vals.reset();
  int pr = 0, pc = 0;
  while (pc < m) {
    int r = pr, c;
    while (r < n && is0(mat[r][pc])) r++;
    if (r == n) { pc++; continue; }

    // F2: mat[pr] ^= mat[r]; mat[r] ^= mat[pr];
    ↪  mat[pr] ^= mat[r];
    for (c = 0; c <= m; c++) swap(mat[pr][c],
    ↪  mat[r][c]);

    r = pr++; c = pc++;
    // F2: vals.set(pc, mat[pr][m]);
    NUM div = mat[r][c];
    for (int col = c; col <= m; col++) mat[r][col] /=
    ↪  div;
    REP(row, n) {
      if (row == r) continue;
      // F2: if (mat[row].test(c)) mat[row] ^=
      ↪  mat[r];
      NUM times = -mat[row][c];
      for (int col = c; col <= m; col++)
```

```
      mat[row][col] += times * mat[r][col];
    }
  } // now mat is in RREF

  for (int r = pr; r < n; r++)
    if (!is0(mat[r][n])) return 0;
  // F2: return 1;
  fill_n(hasval, n, false);
  for (int col = 0, row; col < m; col++) {
    hasval[col] = !is0(mat[row][col]);
    if (!hasval[col]) continue;
    for (int c = col + 1; c < m; c++) {
      if (!is0(mat[row][c])) hasval[col] = false;
    }
    if (hasval[col]) vals[col] = mat[row][n];
    row++;
  }
  REP(i, n) if (!hasval[i]) return 2;
  return 1;
}
```

6.5. Cycle-Finding.

```
ii find_cycle(int x0, int (*f)(int)) {
  int t = f(x0), h = f(t), mu = 0, lam = 1;
  while (t != h) t = f(t), h = f(f(h));
  h = x0;
  while (t != h) t = f(t), h = f(h), mu++;
  h = f(t);
  while (t != h) h = f(h), lam++;
  return ii(mu, lam); }
```

6.6. Longest Increasing Subsequence.

```
vi lis(vi arr) {
  vi seq, back(size(arr)), ans;
  rep(i,0,size(arr)) {
    int res = 0, lo = 1, hi = size(seq);
    while (lo <= hi) {
      int mid = (lo+hi)/2;
      if (arr[seq[mid-1]] < arr[i]) res = mid, lo = mid + 1;
      else hi = mid - 1; }
    if (res < size(seq)) seq[res] = i;
    else seq.push_back(i);
    back[i] = res == 0 ? -1 : seq[res-1]; }
  int at = seq.back();
  while (at != -1) ans.push_back(at), at = back[at];
  reverse(ans.begin(), ans.end());
  return ans; }
```

6.7. Dates.

```
int intToDay(int jd) { return jd % 7; }
int dateToInt(int y, int m, int d) {
  return 1461 * (y + 4800 + (m - 14) / 12) / 4 +
    367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
    3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 +
    d - 32075; }
void intToDate(int jd, int &y, int &m, int &d) {
  int x, n, i, j;
  x = jd + 68569;
  n = 4 * x / 146097;
  x -= (146097 * n + 3) / 4;
  i = (4000 * (x + 1)) / 1461001;
  x -= 1461 * i / 4 - 31;
  j = 80 * x / 2447;
  d = x - 2447 * j / 80;
  x = j / 11;
  m = j + 2 - 12 * x;
  y = 100 * (n - 49) + i + x; }
```

6.8. Simplex.

```
typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;
const DOUBLE EPS = 1e-9;
struct LPSolver {
 int m, n;
 VI B, N;
 VVD D;
 LPSolver(const VVD &A, const VD &b, const VD &c) :
  m(b.size()), n(c.size()),
  N(n + 1), B(m), D(m + 2, VD(n + 2)) {
  for (int i = 0; i < m; i++) for (int j = 0; j < n; j++)
    D[i][j] = A[i][j];
  for (int i = 0; i < m; i++) { B[i] = n + i; D[i][n] = -1;
    D[i][n + 1] = b[i]; }
  for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
  N[n] = -1; D[m + 1][n] = 1; }
 void Pivot(int r, int s) {
  double inv = 1.0 / D[r][s];
  for (int i = 0; i < m + 2; i++) if (i != r)
   for (int j = 0; j < n + 2; j++) if (j != s)
    D[i][j] -= D[r][j] * D[i][s] * inv;
  for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] *= inv;
  for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] *= -inv;
  D[r][s] = inv;
  swap(B[r], N[s]); }
 bool Simplex(int phase) {
  int x = phase == 1 ? m + 1 : m;
  while (true) {
   int s = -1;
   for (int j = 0; j <= n; j++) {
    if (phase == 2 && N[j] == -1) continue;
    if (s == -1 || D[x][j] < D[x][s] ||
        D[x][j] == D[x][s] && N[j] < N[s]) s = j; }
   if (D[x][s] > -EPS) return true;
   int r = -1;
   for (int i = 0; i < m; i++) {
    if (D[i][s] < EPS) continue;
    if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] /
        D[r][s] || (D[i][n + 1] / D[i][s]) == (D[r][n + 1] /
        D[r][s]) && B[i] < B[r]) r = i; }
   if (r == -1) return false;
   Pivot(r, s); } }
 DOUBLE Solve(VD &x) {
  int r = 0;
  for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n + 1])
    r = i;
  if (D[r][n + 1] < -EPS) {
   Pivot(r, n);
   if (!Simplex(1) || D[m + 1][n + 1] < -EPS)
     return -numeric_limits<DOUBLE>::infinity();
   for (int i = 0; i < m; i++) if (B[i] == -1) {
    int s = -1;
    for (int j = 0; j <= n; j++)
     if (s == -1 || D[i][j] < D[i][s] ||
         D[i][j] == D[i][s] && N[j] < N[s])
       s = j;
    Pivot(i, s); } }
  if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
  x = VD(n);
  for (int i = 0; i < m; i++) if (B[i] < n)
    x[B[i]] = D[i][n + 1];
  return D[m][n + 1]; } };
// Two-phase simplex algorithm for solving linear programs
// of the form
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be
//             stored
// OUTPUT: value of the optimal solution (infinity if
//                   unbounded above, nan if infeasible)
// To use this code, create an LPSolver object with A, b,
// and c as arguments.  Then, call Solve(x).
// #include <iostream>
// #include <iomanip>
// #include <vector>
// #include <cmath>
// #include <limits>
// using namespace std;
// int main() {
//   const int m = 4;
//   const int n = 3;
//   DOUBLE _A[m][n] = {
```

```
//     { 6, -1, 0 },
//     { -1, -5, 0 },
//     { 1, 5, 1 },
//     { -1, -5, -1 }
//   };
//   DOUBLE _b[m] = { 10, -4, 5, -5 };
//   DOUBLE _c[n] = { 1, -1, 0 };
//   VVD A(m);
//   VD b(_b, _b + m);
//   VD c(_c, _c + n);
//   for (int i = 0; i < m; i++) A[i] = VD(_A[i],
↪  _A[i] + n);
//   LPSolver solver(A, b, c);
//   VD x;
//   DOUBLE value = solver.Solve(x);
//   cerr << "VALUE: " << value << endl; // VALUE:
↪  1.29032
//   cerr << "SOLUTION:"; // SOLUTION: 1.74194
↪  0.451613 1
//   for (size_t i = 0; i < x.size(); i++) cerr << "
↪  " << x[i];
//   cerr << endl;
//   return 0;
// }
```

7. Geometry (CP3)

7.1. Points and lines.

```
#define INF 1e9
#define EPS 1e-9
#define PI acos(-1.0) // important constant;
↪  alternative #define PI (2.0 * acos(0.0))

double DEG_to_RAD(double d) { return d * PI / 180.0;
↪  }

double RAD_to_DEG(double r) { return r * 180.0 / PI;
↪  }

struct point { double x, y;   // only used if more
↪  precision is needed
  point() { x = y = 0.0; }                        //
  ↪  default constructor
  point(double _x, double _y) : x(_x), y(_y) {}
  ↪  // user-defined
  bool operator < (point other) const { // override
  ↪  less than operator
    if (fabs(x - other.x) > EPS)                  //
    ↪  useful for sorting
      return x < other.x;          // first criteria
      ↪  , by x-coordinate
    return y < other.y; }          // second
    ↪  criteria, by y-coordinate
  // use EPS (1e-9) when testing equality of two
  ↪  floating points
  bool operator == (point other) const {
   return (fabs(x - other.x) < EPS && (fabs(y -
   ↪  other.y) < EPS)); } };

double dist(point p1, point p2) {                  //
↪  Euclidean distance
                              // hypot(dx, dy) returns
                              ↪  sqrt(dx * dx + dy * dy)
  return hypot(p1.x - p2.x, p1.y - p2.y); }
  ↪  // return double

// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {
  double rad = DEG_to_RAD(theta);    // multiply
  ↪  theta with PI / 180.0
  return point(p.x * cos(rad) - p.y * sin(rad),
               p.x * sin(rad) + p.y * cos(rad)); }

struct line { double a, b, c; };          // a way to
↪  represent a line

// the answer is stored in the third parameter (pass
↪  by reference)
void pointsToLine(point p1, point p2, line &l) {
  if (fabs(p1.x - p2.x) < EPS) {                 //
  ↪  vertical line is fine
    l.a = 1.0;   l.b = 0.0;   l.c = -p1.x;
    ↪  // default values
  } else {
    l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
    l.b = 1.0;                  // IMPORTANT: we fix the
    ↪  value of b to 1.0
    l.c = -(double)(l.a * p1.x) - p1.y;
} }

bool areParallel(line l1, line l2) {       // check
↪  coefficients a & b
  return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b)
  ↪  < EPS); }

bool areSame(line l1, line l2) {           // also
↪  check coefficient c
  return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) <
  ↪  EPS); }

// returns true (+ intersection point) if two lines
↪  are intersect
bool areIntersect(line l1, line l2, point &p) {
  if (areParallel(l1, l2)) return false;
  ↪  // no intersection
  // solve system of 2 linear algebraic equations
  ↪  with 2 unknowns
  p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b -
  ↪  l1.a * l2.b);
  // special case: test for vertical line to avoid
  ↪  division by zero
  if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
  else                  p.y = -(l2.a * p.x + l2.c);
  return true; }

struct vec { double x, y;  // name: `vec' is
↪  different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {       // convert 2
↪  points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) {        // nonnegative s
↪  = [<1 .. 1 .. >1]
  return vec(v.x * s, v.y * s); }                //
  ↪  shorter.same.longer

point translate(point p, vec v) {        // translate
↪  p according to v
  return point(p.x + v.x , p.y + v.y); }

// convert point and gradient/slope to line
void pointSlopeToLine(point p, double m, line &l) {
  l.a = -m;
  ↪  // always -m
  l.b = 1;
  ↪  // always 1
  l.c = -((l.a * p.x) + (l.b * p.y)); }
  ↪  // compute this

void closestPoint(line l, point p, point &ans) {
  line perpendicular;         // perpendicular to l
  ↪  and pass through p
  if (fabs(l.b) < EPS) {              // special case
  ↪  1: vertical line
    ans.x = -(l.c);   ans.y = p.y;      return; }

  if (fabs(l.a) < EPS) {              // special case
  ↪  2: horizontal line
    ans.x = p.x;      ans.y = -(l.c);   return; }

  pointSlopeToLine(p, 1 / l.a, perpendicular);
  ↪  // normal line
  // intersect line l with this perpendicular line
  // the intersection point is the closest point
  areIntersect(l, perpendicular, ans); }

// returns the reflection of point on a line
void reflectionPoint(line l, point p, point &ans) {
  point b;
  closestPoint(l, p, b);                        //
  ↪  similar to distToLine
  vec v = toVec(p, b);                              //
  ↪  create a vector
  ans = translate(translate(p, v), v); }          //
  ↪  translate p twice
```

```
double dot(vec a, vec b) { return (a.x * b.x + a.y *
↪  b.y); }

double norm_sq(vec v) { return v.x * v.x + v.y * v.y;
↪  }

// returns the distance from p to the line defined by
// two points a and b (a and b must be different)
// the closest point is stored in the 4th parameter
↪  (byref)
double distToLine(point p, point a, point b, point
↪  &c) {
  // formula: c = a + u * ab
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  c = translate(a, scale(ab, u));                    //
  ↪  translate a to c
  return dist(p, c); }           // Euclidean
  ↪  distance between p and c

// returns the distance from p to the line segment ab
↪  defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th parameter
↪  (byref)
double distToLineSegment(point p, point a, point b,
↪  point &c) {
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  if (u < 0.0) { c = point(a.x, a.y);
  ↪  // closer to a
    return dist(p, a); }         // Euclidean
    ↪  distance between p and a
  if (u > 1.0) { c = point(b.x, b.y);
  ↪  // closer to b
    return dist(p, b); }         // Euclidean
    ↪  distance between p and b
  return distToLine(p, a, b, c); }          // run
  ↪  distToLine as above

double angle(point a, point o, point b) {  // returns
↪  angle aob in rad
  vec oa = toVec(o, a), ob = toVec(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) *
  ↪  norm_sq(ob))); }

double cross(vec a, vec b) { return a.x * b.y - a.y *
↪  b.x; }

// note: to accept collinear points, we have to
↪  change the `> 0'
// returns true if point r is on the left side of
↪  line pq
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0; }
```

```
// returns true if point r is on the same line as the
↪  line pq
bool collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))) < EPS;
  ↪  }
```

7.2. **Polygon.**

```
// returns the perimeter, which is the sum of
↪  Euclidian distances
// of consecutive line segments (polygon edges)
double perimeter(const vector<point> &P) {
  double result = 0.0;
  for (int i = 0; i < (int)P.size()-1; i++)  //
  ↪  remember that P[0] = P[n-1]
    result += dist(P[i], P[i+1]);
  return result; }

// returns the area, which is half the determinant
double area(const vector<point> &P) {
  double result = 0.0, x1, y1, x2, y2;
  for (int i = 0; i < (int)P.size()-1; i++) {
    x1 = P[i].x; x2 = P[i+1].x;
    y1 = P[i].y; y2 = P[i+1].y;
    result += (x1 * y2 - x2 * y1);
  }
  return fabs(result) / 2.0; }

// returns true if we always make the same turn while
↪  examining
// all the edges of the polygon one by one
bool isConvex(const vector<point> &P) {
  int sz = (int)P.size();
  if (sz <= 3) return false;   // a point/sz=2 or a
  ↪  line/sz=3 is not convex
  bool isLeft = ccw(P[0], P[1], P[2]);
  ↪  // remember one result
  for (int i = 1; i < sz-1; i++)             // then
  ↪  compare with the others
    if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2])
    ↪  != isLeft)
      return false;            // different sign ->
      ↪  this polygon is concave
  return true; }                                     //
  ↪  this polygon is convex

// returns true if point p is in either
↪  convex/concave polygon P
bool inPolygon(point pt, const vector<point> &P) {
  if ((int)P.size() == 0) return false;
  double sum = 0;    // assume the first vertex is
  ↪  equal to the last vertex
  for (int i = 0; i < (int)P.size()-1; i++) {
    if (ccw(pt, P[i], P[i+1]))
         sum += angle(P[i], pt, P[i+1]);
         ↪  // left turn/ccw
    else sum -= angle(P[i], pt, P[i+1]); }
    ↪  // right turn/cw
  return fabs(fabs(sum) - 2*PI) < EPS; }

// line segment p-q intersect with line A-B.
point lineIntersectSeg(point p, point q, point A,
↪  point B) {
  double a = B.y - A.y;
  double b = A.x - B.x;
  double c = B.x * A.y - A.x * B.y;
  double u = fabs(a * p.x + b * p.y + c);
  double v = fabs(a * q.x + b * q.y + c);
  return point((p.x * v + q.x * u) / (u+v), (p.y * v
  ↪  + q.y * u) / (u+v)); }

// cuts polygon Q along the line formed by point a ->
↪  point b
// (note: the last point must be the same as the
↪  first point)
vector<point> cutPolygon(point a, point b, const
↪  vector<point> &Q) {
  vector<point> P;
  for (int i = 0; i < (int)Q.size(); i++) {
    double left1 = cross(toVec(a, b), toVec(a,
    ↪  Q[i])), left2 = 0;
    if (i != (int)Q.size()-1) left2 = cross(toVec(a,
    ↪  b), toVec(a, Q[i+1]));
    if (left1 > -EPS) P.push_back(Q[i]);       //
    ↪  Q[i] is on the left of ab
    if (left1 * left2 < -EPS)        // edge (Q[i],
    ↪  Q[i+1]) crosses line ab
      P.push_back(lineIntersectSeg(Q[i], Q[i+1], a,
      ↪  b));
  }
  if (!P.empty() && !(P.back() == P.front()))
    P.push_back(P.front());        // make P's first
    ↪  point = P's last point
  return P; }

point pivot;
bool angleCmp(point a, point b) {                  //
↪  angle-sorting function
  if (collinear(pivot, a, b))
  ↪  // special case
    return dist(pivot, a) < dist(pivot, b);    //
    ↪  check which one is closer
  double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
  double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
  return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0; }
  ↪  // compare two angles

vector<point> CH(vector<point> P) {   // the content
↪  of P may be reshuffled
  int i, j, n = (int)P.size();
  if (n <= 3) {
```

```
    if (!(P[0] == P[n-1])) P.push_back(P[0]); //
    ↪ safeguard from corner case
    return P;                              // special
    ↪ case, the CH is P itself
  }

  // first, find P0 = point with lowest Y and if tie:
  ↪ rightmost X
  int P0 = 0;
  for (i = 1; i < n; i++)
    if (P[i].y < P[P0].y || (P[i].y == P[P0].y &&
    ↪ P[i].x > P[P0].x))
      P0 = i;

  point temp = P[0]; P[0] = P[P0]; P[P0] = temp;
  ↪ // swap P[P0] with P[0]

  // second, sort points by angle w.r.t. pivot P0
  pivot = P[0];                        // use this global
  ↪ variable as reference
  sort(++P.begin(), P.end(), angleCmp);
  ↪ // we do not sort P[0]

  // third, the ccw tests
  vector<point> S;
  S.push_back(P[n-1]); S.push_back(P[0]);
  ↪ S.push_back(P[1]);   // initial S
  i = 2;                                          //
  ↪ then, we check the rest
  while (i < n) {            // note: N must be >= 3
  ↪ for this method to work
    j = (int)S.size()-1;
    if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]);
    ↪ // left turn, accept
    else S.pop_back(); }   // or pop the top of S
    ↪ until we have a left turn
  return S; }
  ↪ // return the result
```

7.3. Triangle.

```
double perimeter(double ab, double bc, double ca) {
  return ab + bc + ca; }

double perimeter(point a, point b, point c) {
  return dist(a, b) + dist(b, c) + dist(c, a); }

double area(double ab, double bc, double ca) {
  // Heron's formula, split sqrt(a * b) into sqrt(a)
  ↪ * sqrt(b); in implementation
  double s = 0.5 * perimeter(ab, bc, ca);
  return sqrt(s) * sqrt(s - ab) * sqrt(s - bc) *
  ↪ sqrt(s - ca); }

double area(point a, point b, point c) {
  return area(dist(a, b), dist(b, c), dist(c, a)); }

double rInCircle(double ab, double bc, double ca) {
  return area(ab, bc, ca) / (0.5 * perimeter(ab, bc,
  ↪ ca)); }

double rInCircle(point a, point b, point c) {
  return rInCircle(dist(a, b), dist(b, c), dist(c,
  ↪ a)); }

// assumption: the required points/lines functions
↪ have been written
// returns 1 if there is an inCircle center, returns
↪ 0 otherwise
// if this function returns 1, ctr will be the
↪ inCircle center
// and r is the same as rInCircle
int inCircle(point p1, point p2, point p3, point
↪ &ctr, double &r) {
  r = rInCircle(p1, p2, p3);
  if (fabs(r) < EPS) return 0;                    //
  ↪ no inCircle center

  line l1, l2;                        // compute these
  ↪ two angle bisectors
  double ratio = dist(p1, p2) / dist(p1, p3);
  point p = translate(p2, scale(toVec(p2, p3), ratio
  ↪ / (1 + ratio)));
  pointsToLine(p1, p, l1);

  ratio = dist(p2, p1) / dist(p2, p3);
  p = translate(p1, scale(toVec(p1, p3), ratio / (1 +
  ↪ ratio)));
  pointsToLine(p2, p, l2);

  areIntersect(l1, l2, ctr);           // get their
  ↪ intersection point
  return 1; }

double rCircumCircle(double ab, double bc, double ca)
↪ {
  return ab * bc * ca / (4.0 * area(ab, bc, ca)); }

double rCircumCircle(point a, point b, point c) {
  return rCircumCircle(dist(a, b), dist(b, c),
  ↪ dist(c, a)); }

// assumption: the required points/lines functions
↪ have been written
// returns 1 if there is a circumCenter center,
↪ returns 0 otherwise
// if this function returns 1, ctr will be the
↪ circumCircle center
// and r is the same as rCircumCircle
int circumCircle(point p1, point p2, point p3, point
↪ &ctr, double &r){
  double a = p2.x - p1.x, b = p2.y - p1.y;
  double c = p3.x - p1.x, d = p3.y - p1.y;
  double e = a * (p1.x + p2.x) + b * (p1.y + p2.y);
  double f = c * (p1.x + p3.x) + d * (p1.y + p3.y);
  double g = 2.0 * (a * (p3.y - p2.y) - b * (p3.x -
  ↪ p2.x));
  if (fabs(g) < EPS) return 0;

  ctr.x = (d*e - b*f) / g;
  ctr.y = (a*f - c*e) / g;
  r = dist(p1, ctr);  // r = distance from center to
  ↪ 1 of the 3 points
  return 1; }

// returns if pt d is inside the circumCircle defined
↪ by a,b,c
bool inCircumCircle(point a,point b,point c,point d){
  vec va=toVec(a, d), vb=toVec(b, d), vc=toVec(c, d);
  return 0 <
    (va.x)*(vb.y)*((vc.x)*(vc.x)+(vc.y)*(vc.y))+
    (va.y)*((vb.x)*(vb.x)+(vb.y)*(vb.y))*(vc.x)+
    ((va.x)*(va.x)+(va.y)*(va.y))*(vb.x)*(vc.y)-
    ((va.x)*(va.x)+(va.y)*(va.y))*(vb.y)*(vc.x)-
    (va.y)*(vb.x)*((vc.x)*(vc.x)+(vc.y)*(vc.y))-
    (va.x)*((vb.x)*(vb.x)+(vb.y)*(vb.y))*(vc.y);
}

bool canFormTriangle(double a, double b, double c) {
  return (a + b > c) && (a + c > b) && (b + c > a); }
```

7.4. Circle.

```
int insideCircle(point_i p, point_i c, int r) { //
↪ all integer version
  int dx = p.x - c.x, dy = p.y - c.y;
  int Euc = dx * dx + dy * dy, rSq = r * r;
  ↪ // all integer
  return Euc < rSq ? 0 : Euc == rSq ? 1 : 2; }
  ↪ //inside/border/outside

bool circle2PtsRad(point p1, point p2, double r,
↪ point &c) {
  double d2 = (p1.x - p2.x) * (p1.x - p2.x) +
              (p1.y - p2.y) * (p1.y - p2.y);
  double det = r * r / d2 - 0.25;
  if (det < 0.0) return false;
  double h = sqrt(det);
  c.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
  c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
  return true; }          // to get the other center,
  ↪ reverse p1 and p2
```

8. Combinatorics

- Catalan numbers (valid bracket seq's of length $2n$): $C_0 = 1, C_n = \frac{1}{n+1}\binom{2n}{n} = \sum_{i=0}^{n-1} C_i C_{n-i-1}$.
- Stirling 1^{th} kind ($\#\pi \in \mathfrak{S}_n$ with exactly k cycles): $\left[{n \atop 0}\right] = \left[{0 \atop n}\right] = \delta_{0n}, \left[{n \atop k}\right] = (n-1)\left[{n-1 \atop k}\right] + \left[{n-1 \atop k-1}\right]$.

- Stirling 2^{nd} kind (k-partitions of $[n]$):
  $\left\{ {n \atop 1} \right\} = \left\{ {n \atop n} \right\} = 1, \left\{ {n \atop k} \right\} = k \left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\}$.
- Bell numbers (partitions of $[n]$):
  $B_0 = 1, B_n = \sum_{k=0}^{n-1} B_k \binom{n-1}{k} = \sum_{k=0}^{n} \left\{ {n \atop k} \right\}$.
- Euler ($\#\pi \in \mathfrak{S}_n$ with exactly k ascents):
  $\left\langle {n \atop 0} \right\rangle = \left\langle {n \atop n-1} \right\rangle = 1, \left\langle {n \atop k} \right\rangle = (k+1) \left\langle {n-1 \atop k} \right\rangle + (n-k) \left\langle {n-1 \atop k-1} \right\rangle$.
- Euler 2^{nd} order (nr perms of $1, 1, 2, 2, \dots, n, n$ with exactly k ascents):
  $\left\langle {n \atop 0} \right\rangle = \left\langle {n \atop n-1} \right\rangle = 1, \left\langle {n \atop k} \right\rangle = (k+1) \left\langle {n-1 \atop k} \right\rangle + (n-k) \left\langle {n-1 \atop k-1} \right\rangle$.
- Rooted trees: n^{n-1}, unrooted: n^{n-2}.
- Forests of k rooted trees: $\binom{n}{k} k \cdot n^{n-k-1}$.
- $1^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}$, $\quad 1^3 + \dots + n^3 = \frac{n^2(n+1)^2}{4}$
- $\sum_{i=1}^{n} \binom{n}{i} F_i = F_{2n}$, $\quad \sum_i \binom{n-i}{i} = F_{n+1}$
- $\sum_{k=0}^{n} \binom{k}{m} = \binom{n+1}{m+1}$, $\quad x^k = \sum_{i=0}^{k} i! \left\{ {k \atop i} \right\} \binom{x}{i} = \sum_{i=0}^{k} \left\langle {k \atop i} \right\rangle \binom{x+i}{k}$
- $a \equiv b \pmod{x, y} \Leftrightarrow a \equiv b \pmod{\operatorname{lcm}(x, y)}$.
- $ac \equiv bc \pmod{m} \Leftrightarrow a \equiv b \pmod{m/\gcd(c, m)}$.
- $\gcd(n^a - 1, n^b - 1) = \gcd(a, b) - 1$.
- **Möbius inversion formula:** If $f(n) = \sum_{d|n} g(d)$, then $g(n) = \sum_{d|n} \mu(d) f(n/d)$. If $f(n) = \sum_{m=1}^{n} g(\lfloor n/m \rfloor)$, then $g(n) = \sum_{m=1}^{n} \mu(m) f(\lfloor \frac{n}{m} \rfloor)$.
- **Inclusion-Exclusion:** If $g(T) = \sum_{S \subseteq T} f(S)$, then

$$f(T) = \sum_{S \subseteq T} (-1)^{|T \setminus S|} g(T).$$

Corollary: $b_n = \sum_{k=0}^{n} \binom{n}{k} a_k \Longleftrightarrow a_n = \sum_{k=0}^{n} (-1)^{n-k} \binom{n}{k} b_k$.

8.1. The Twelvefold Way.

Putting n balls into k boxes. $\mathrm{p}(n, k)$ is $\#$ partitions of n in k parts, each > 0. $\mathrm{p}_k(n) = \sum_{i=0}^{k} \mathrm{p}(n, k)$.

Balls / Boxes	same / same	distinct / same	same / distinct	distinct / distinct
-	$\mathrm{p}_k(n)$	$\sum_{i=0}^{k} \left\{ {n \atop i} \right\}$	$\binom{n+k-1}{k-1}$	k^n
size ≥ 1	$\mathrm{p}(n, k)$	$\left\{ {n \atop k} \right\}$	$\binom{n-1}{k-1}$	$k! \left\{ {n \atop k} \right\}$
size ≤ 1	$[n \leq k]$	$[n \leq k]$	$\binom{k}{n}$	$n! \binom{k}{n}$

9. Formulas

- **Legendre symbol:** $\left(\frac{a}{b}\right) = a^{(b-1)/2} \pmod{b}$, b odd prime.
- **Heron's formula:** A triangle with side lengths a, b, c has area $\sqrt{s(s-a)(s-b)(s-c)}$ where $s = \frac{a+b+c}{2}$.
- **Pick's theorem:** A polygon on an integer grid strictly containing i lattice points and having b lattice points on the boundary has area $i + \frac{b}{2} - 1$. (Nothing similar in higher dimensions)
- **König's theorem:** In any bipartite graph $G = (L \cup R, E)$, the number of edges in a maximum matching is equal to the number of vertices in a minimum vertex cover. Let U be the set of unmatched vertices in L, and Z be the set of vertices that are either in U or are connected to U by an alternating path. Then $K = (L \setminus Z) \cup (R \cap Z)$ is the minimum vertex cover.
- A minumum Steiner tree for n vertices requires at most $n - 2$ additional Steiner vertices.
- The number of vertices of a graph is equal to its minimum vertex cover number plus the size of a maximum independent set.
- **Lagrange polynomial** through points $(x_0, y_0), \dots, (x_k, y_k)$ is $L(x) = \sum_{j=0}^{k} y_j \prod_{\substack{0 \leq m \leq k \\ m \neq j}} \frac{x - x_m}{x_j - x_m}$
- **Hook length formula:** If λ is a Young diagram and $h_\lambda(i, j)$ is the hook-length of cell (i, j), then then the number of Young tableux $d_\lambda = n! / \prod h_\lambda(i, j)$.
- #primitive pythagorean triples with hypotenuse $< n$ approx $n/(2\pi)$.
- **Frobenius Number:** largest number which can't be expressed as a linear combination of numbers $a_1, \dots, a_n$ with non-negative coefficients. $g(a_1, a_2) = a_1 a_2 - a_1 - a_2$, $N(a_1, a_2) = (a_1 - 1)(a_2 - 1)/2$. $g(d \cdot a_1, d \cdot a_2, a_3) = d \cdot g(a_1, a_2, a_3) + a_3(d - 1)$. An integer $x > (\max_i a_i)^2$ can be expressed in such a way iff. $x \mid \gcd(a_1, \dots, a_n)$.
- **Snell's law:** $v_2 \sin \theta_1 = v_1 \sin \theta_2$ gives the shortest path between two media.
- **BEST theorem:** The number of Eulerian cycles in a *directed* graph G is:

$$t_w(G) \prod_{v \in G} (\deg v - 1)!,$$

where $t_w(G)$ is the number of arborescences ("directed spanning" tree) rooted at w: $t_w(G) = \det(q_{ij})_{i,j \neq w}$, with $q_{ij} = [i = j] \operatorname{indeg}(i) - \#\{(i, j) \in E\}$.

9.1. Burnside's Lemma.

Let a finite group G act on a set X. Denote $X^g = \{x \in X \mid gx = x\}$. For each g in G let X^g denote the set of elements in X that are fixed by g. Then the number of orbits is:

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g|.$$

9.2. Bézout's identity.

If (x, y) is a solution to $ax + by = d$ (x, y can be found with EGCD), then all solutions are given by

$$(x + k \cdot \operatorname{lcm}(a, b)/a,\ y - k \cdot \operatorname{lcm}(a, b)/b), \quad k \in \mathbb{Z}$$

10. Game Theory

A game can be reduced to Nim if it is a finite impartial game. Nim and its variants include:

- **Nim:** Let $X = \bigoplus_{i=1}^{n} x_i$, then $(x_i)_{i=1}^{n}$ is a winning position iff $X \neq 0$. Find a move by picking k such that $x_k > x_k \oplus X$.
- **Misère Nim:** Regular Nim, except that the last player to move *loses*. Play regular Nim until there is only one pile of size larger than 1, reduce it to 0 or 1 such that there is an odd number of piles. The second player wins $(a_1, \dots, a_n)$ if 1) there is a pile $a_i > 1$ and $\oplus_{i=1}^{n} a_i = 0$ or 2) all $a_i \leq 1$ and $\oplus_{i=1}^{n} a_i = 1$.
- **Staircase Nim:** Stones are moved down a staircase and only removed from the last pile. $(x_i)_{i=1}^{n}$ is an L-position if $(x_{2i-1})_{i=1}^{n/2}$ is (i.e. only look at odd-numbered piles).
- **Moore's Nim$_k$:** The player may remove from at most k piles ($\text{Nim} = \text{Nim}_1$). Expand the piles in base 2, do a carry-less addition in base $k + 1$ (i.e. the number of ones in each column should be divisible by $k + 1$).
- **Dim$^+$:** The number of removed stones must be a divisor of the pile size. The Sprague-Grundy function is $k + 1$ where 2^k is the largest power of 2 dividing the pile size.
- **Aliquot game:** Same as above, except the divisor should be proper (hence 1 is also a terminal state, but watch out for size 0 piles). Now the Sprague-Grundy function is just k.
- **Nim (at most half):** Write $n + 1 = 2^m y$ with m maximal, then the Sprague-Grundy function of n is $(y - 1)/2$.
- **Lasker's Nim:** Players may alternatively split a pile into two new non-empty piles. $g(4k + 1) = 4k + 1$, $g(4k + 2) = 4k + 2$, $g(4k + 3) = 4k + 4$, $g(4k + 4) = 4k + 3$ ($k \geq 0$).
- **Hackenbush on trees:** A tree with stalks $(x_i)_{i=1}^{n}$ may be replaced with a single stalk with length $\bigoplus_{i=1}^{n} x_i$.

11. Debugging Tips

- Stack overflow? Recursive DFS on tree that is actually a long path?
- Floating-point numbers
  - Getting `NaN`? Make sure `acos` etc. are not getting values out of their range (perhaps `1+eps`).
  - Rounding negative numbers?
  - Outputting in scientific notation?
- Wrong Answer?
  - Read the problem statement again!
  - Are multiple test cases being handled correctly? Try repeating the same test case many times.
  - Integer overflow?
  - Think very carefully about boundaries of all input parameters
  - Try out possible edge cases:
    - $n = 0, n = -1, n = 1, n = 2^{31} - 1$ or $n = -2^{31}$
    - List is empty, or contains a single element
    - n is even, n is odd
    - Graph is empty, or contains a single vertex
    - Graph is a multigraph (loops or multiple edges)
    - Polygon is concave or non-simple
  - Is initial condition wrong for small cases?
  - Are you sure the algorithm is correct?
  - Explain your solution to someone.
  - Are you using any functions that you don't completely understand? Maybe STL functions?
  - Maybe you (or someone else) should rewrite the solution?
  - Can the input line be empty?
- Run-Time Error?
  - Is it actually Memory Limit Exceeded?

11.1. Solution Ideas.

- Dynamic Programming
  - Parsing CFGs: CYK Algorithm
  - Drop a parameter, recover from others

- Swap answer and a parameter
- When grouping: try splitting in two
- 2^k trick
- When optimizing
  * Convex hull optimization
    · $\text{dp}[i] = \min_{j<i}\{\text{dp}[j] + b[j] \times a[i]\}$
    · $b[j] \geq b[j+1]$
    · optionally $a[i] \leq a[i+1]$
    · $O(n^2)$ to $O(n)$
  * Divide and conquer optimization
    · $\text{dp}[i][j] = \min_{k<j}\{\text{dp}[i-1][k] + C[k][j]\}$
    · $A[i][j] \leq A[i][j+1]$
    · $O(kn^2)$ to $O(kn \log n)$
    · sufficient: $C[a][c] + C[b][d] \leq C[a][d] + C[b][c]$, $a \leq b \leq c \leq d$ (QI)

```
vvi A; // A[i][j] is voor [i,j]

void divco(ll ls,ll rs,ll lt,ll rt,vi &t,
↪  vi &s){ // berekent t|_{[lt,rt)}
  if(lt >= rt)
    return;
  ll ms = ls, mt = (lt + rt)/2;
  t[mt] = -INF;
  rep(i,ls,rs){
    if(i >= mt){
      break;
    }
    if(s[i] + A[i][mt] > t[mt]){
      t[mt] = s[i] + A[i][mt];
      ms = i;
    }
  }
  divco(ls,ms+1,lt,mt,t,s);
  divco(ms,rs,mt+1,rt,t,s);
}
```

  * Knuth optimization
    · $\text{dp}[i][j] = \min_{i<k<j}\{\text{dp}[i][k] + \text{dp}[k][j] + C[i][j]\}$
    · $A[i][j-1] \leq A[i][j] \leq A[i+1][j]$
    · $O(n^3)$ to $O(n^2)$
    · sufficient: QI and $C[b][c] \leq C[a][d]$, $a \leq b \leq c \leq d$
- Greedy
- Randomized
- Optimizations
  - Use bitset (/64)
  - Switch order of loops (cache locality)
- Process queries offline
  - Mo's algorithm
- Square-root decomposition
- Precomputation
- Efficient simulation
  - Mo's algorithm
  - Sqrt decomposition
  - Store 2^k jump pointers
- Data structure techniques
  - Sqrt buckets
  - Store 2^k jump pointers
  - 2^k merging trick
- Counting
  - Inclusion-exclusion principle
  - Generating functions
- Graphs
  - Can we model the problem as a graph?
  - Can we use any properties of the graph?
  - Strongly connected components
  - Cycles (or odd cycles)
  - Bipartite (no odd cycles)
    * Bipartite matching
    * Hall's marriage theorem
    * Stable Marriage
  - Cut vertex/bridge
  - Biconnected components
  - Degrees of vertices (odd/even)
  - Trees
    * Heavy-light decomposition
    * Centroid decomposition
    * Least common ancestor
    * Centers of the tree
  - Eulerian path/circuit
  - Chinese postman problem
  - Topological sort
  - (Min-Cost) Max Flow
  - Min Cut
    * Maximum Density Subgraph
  - Huffman Coding
  - Min-Cost Arborescence
  - Steiner Tree
  - Kirchoff's matrix tree theorem
  - Prüfer sequences
  - Lovász Toggle
  - Look at the DFS tree (which has no cross-edges)
  - Is the graph a DFA or NFA?
    * Is it the Synchronizing word problem?
- math
  - Is the function multiplicative?
  - Look for a pattern
  - Permutations
    * Consider the cycles of the permutation
  - Functions
    * Sum of piecewise-linear functions is a piecewise-linear function
    * Sum of convex (concave) functions is convex (concave)
  - Modular arithmetic
    * Chinese Remainder Theorem
    * Linear Congruence
  - Sieve
  - System of linear equations
  - Values too big to represent?
    * Compute using the logarithm
    * Divide everything by some large value
  - Linear programming
    * Is the dual problem easier to solve?
  - Can the problem be modeled as a different combinatorial problem? Does that simplify calculations?
- Logic
  - 2-SAT
  - XOR-SAT (Gauss elimination or Bipartite matching)
- Meet in the middle
- Only work with the smaller half ($\log(n)$)
- Strings
  - Trie (maybe over something weird, like bits)
  - Suffix array
  - Suffix automaton (+DP?)
  - Aho-Corasick
  - eerTree
  - Work with $S + S$
- Hashing
- Euler tour, tree to array
- Segment trees
  - Lazy propagation
  - Persistent
  - Implicit
  - Segment tree of X
- Geometry
  - Minkowski sum (of convex sets)
  - Rotating calipers
  - Sweep line (horizontally or vertically?)
  - Sweep angle
  - Convex hull
- Fix a parameter (possibly the answer).
- Are there few distinct values?
- Binary search
- Sliding Window (+ Monotonic Queue)
- Computing a Convolution? Fast Fourier Transform
- Computing a 2D Convolution? FFT on each row, and then on each column
- Exact Cover (+ Algorithm X)
- Cycle-Finding
- What is the smallest set of values that identify the solution? The cycle structure of the permutation? The powers of primes in the factorization?
- Look at the complement problem
  - Minimize something instead of maximizing
- Immediately enforce necessary conditions. (All values greater than 0? Initialize them all to 1)
- Add large constant to negative numbers to make them positive
- Counting/Bucket sort

Practice Contest Checklist

- How many operations per second? Compare to local machine.
- What is the stack size?
- How to use printf/scanf with long long/long double?
- Are `__int128` and `__float128` available?
- Does MLE give RTE or MLE as a verdict? What about stack overflow?
- What is `RAND_MAX`?
- How does the judge handle extra spaces (or missing newlines) in the output?
- Look at documentation for programming languages.
- Try different programming languages: C++, Java and Python.
- Try the submit script.
- Try local programs: i?python[23], factor.
- Try submitting with `assert(false)` and `assert(true)`.
- Omitting `return 0;` still works?
- Look for directory with sample test cases.
- Make sure printing works.