TCR

Team Biem – Utrecht University

Thomas van der Plas & Jippe Hoogeveen

Contents

Hash script

```
# Last 6 chars of hash should match; var names
↪ matter
gcc -E -P -w -nostdinc file.cpp | tr -d '[:space:]'
↪ | md5sum
```

Test script

```
g++ -g -Wall -fsanitize=address,undefined
↪ -Wfatal-errors -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.cc

Hash: b33a1b

```
#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;
typedef vector<vii> vvii;
typedef vector<bool> vb;
typedef vector<vb> vvb;
typedef vector<ld> vd;
typedef vector<vd> vvd;
typedef vector<string> vs;

#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) begin(v), end(v)
#define sz(v) ((int) (v).size())
#define rs resize

namespace std { template<class T1, class T2>
struct hash<pair<T1,T2>> { public:
  size_t operator()(const pair<T1,T2> &p) const {
    size_t x = hash<T1>()(p.x), y = hash<T2>()(p.y);
    return x ^ (y + 0x9e3779b9 + (x<<6) + (x>>2));
  }
}; }

void run() {

}
```

```cpp
signed main() {
  // DON'T MIX "scanf" and "cin"!
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  cout << fixed << setprecision(20);
  run();
  return 0;
}
```

template.py

```python
# reading input:
from sys import *
n,m = [ int(x) for x in
↪  stdin.readline().rstrip().split() ]
stdout.write( str(n*m)+"\n" )
# set operations:
from itertools import *
for (x,y) in product(range(3),repeat=2):
  stdout.write( str(3*x+y)+" " )
print()
for L in combinations(range(4),2):
  stdout.write( str(L)+" " )
print()
# fancy lambdas:
from functools import *
y = reduce( lambda x,y: x+y, map( lambda x: x*x,
↪  range(4) ), -3 )
print(y)
# formatting:
from math import *
stdout.write( "{0:.2f}\n".format(pi) )
```

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.

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

Basics Hash: 6b2dcb

```cpp
auto comp = [](T i, T j){return i < j;}; //strictly
↪  smaller!
sort(v.begin(), v.end(), comp); // Sort vector v
priority_queue<T, vector<T>, decltype(comp)>
↪  q(comp); // Max heap.
set<T, decltype(comp)> s(comp); // Balanced binary
↪  search tree.
map<T, S, decltype(comp)> m(comp); // Balanced
↪  binary search tree map with key T and value S.

random_device rd; mt19937 gen(rd());
uniform_int_distribution<ll> uid(1, 9); // Generate
↪  integers between 1 and 9 inclusive.
uniform_real_distribution<ld> urd(1.0, 2.0); //
↪  Generate real numbers between 1 and 2
↪  left-inclusive.
ll x = uid(gen); ld y = urd(gen);
```

Detecting overflow: This GNU builtin checks for over- and underflow. Result is in `res` if successful: Hash: d5a5b6

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

1. Mathematics

XOR sum: $\bigoplus_{x=0}^{a-1} x = \{0, a-1, 1, a\}[a \bmod 4]$.
Hash: d7a1d4

```cpp
int sign(ll x) { return (x > 0) - (x < 0); }

ll lcm(ll a, ll b) { return a/gcd(a, b)*b; }
ll mod(ll a, ll b) { return (a % b + b) % b; }

// 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 mod(r,m);
}

// 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;
  if(a < 0) a *= -1, x = -1;
  if(b < 0) b *= -1, yy = -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) {
  ll s, t, d = egcd(n, m, s, t); //n,m<=1e9
  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);
  REP(i, N) phi[i] = i;
  rep(i,2,N) if (phi[i] == i)
    for (int j = i; j < N; j+=i) phi[j] -= phi[j]/i;
  return phi;
}

//Calculate (nCK % m) in O(k)
//Assert gcd(i, m) = 1 for i <= k
ll binom(ll n, ll k, ll m) {
  ll ans = 1, inv, y;
  REP(i, k) {
    ans = mod(ans * (n - i), m);
    egcd(i + 1, m, inv, y);
    ans = mod(ans * inv, m);
  }
  return ans;
}

// calculate nCk % p (p prime!) O(p log_p(n))
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), p);
    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. Hash: 370e04

```
ll primitive_root(ll m) {
  vi div; ll phi = m - 1;
  for (ll i = 2; i*i <= phi; i++)
    if (phi % i == 0) {
      div.pb(i);
      div.pb(phi/i);
    }
  rep(x,2,m) {//skip if gcd(x, m) != 1
    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. Hash: 248c3b

```
ll legendre(ll a, ll p) {
  if (a % p == 0) return 0;
  return p == 2 || 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 (with $O(EPS^4)$ error). Hash: f08ec9

```
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 XOR-convolutions in $O(k2^k)$ on k bits.
For AND-convolution, use $(x+y,\, y), \quad (x-y,\, y)$.
For OR-convolution, use $(x,\, x+y), \quad (x,\, -x+y)$.
Note: The array size must be a power of 2. Hash: 60e7b5

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

vi conv(vi A, vi B) {
  int n = sz(A);
  fht(A,false,0,n); fht(B,false,0,n);
  vi res = vi(n); REP(i,n) res[i] = A[i] * B[i];
  fht(res,true,0,n);
  return res;
}
```

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. Hash: c97ec8

```
void solve(int n, vd& a, vd& b, vd& c, vd& d, vd& x)
↪  {
  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. Number of Integer Points under Line.

Count the number of integer solutions to $Ax+By \leq C$, $0 \leq x \leq n$, $0 \leq y$. In other words, evaluate the sum $\sum_{x=0}^{n} \max(0, \lfloor \frac{C-Ax}{B} + 1 \rfloor)$. Be very careful about overflows. Hash: 9111c1

```
ll floor_sum(ll n, ll a, ll b, ll c) {
    if(a < 0) c -= n * a, a *= -1;
    if (c == 0) return 1;
    if (c < 0) return 0;
    n = min(n, c / a);
    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); }
```

1.7. Solving linear recurrences.

Given some brute-forced sequence $s[0], s[1], \ldots, s[2n-1]$, Berlekamp-Massey finds the shortest possible recurrence relation in $\mathcal{O}(n^2)$. After that, lin_rec finds $s[k]$ in $\mathcal{O}(n^2 \log k)$. Hash: abc4ad

```
// Given a sequence s[0], ..., s[2n-1] finds the
↪  smallest linear recurrence
// of size <= n compatible with s.
vi BerlekampMassey(const vi &s, ll mod) {
  int n = sz(s), L = 0, m = 0;
  vi C(n), B(n), T;
  C[0] = B[0] = 1;
  ll b = 1;
  REP(i, n) {
    ++m;
    ll d = s[i] % mod;
    rep(j, 1, L+1) d = (d + C[j] * s[i - j]) % mod;
    if (!d) continue;
    T = C;
    ll coef = d * modpow(b, mod-2, mod) % mod;
    rep(j,m,n) C[j] = (C[j] - coef * B[j-m]) % mod;
    if (2 * L > i) continue;
    L = i + 1 - L;
    B = T; b = d; m = 0;
  }
  C.rs(L + 1);
  C.erase(C.begin());
  for (auto &x : C) x = (mod - x) % mod;
  return C;
}

// Input: A[0,...,n-1], C[0,...,n-1] satisfying
//     A[i] = \sum_{j=1}^{n} C[j-1] A[i-j],
// Outputs A[k]
ll lin_rec(const vi &A, const vi &C, ll k, ll mod){
  int n = sz(A);
  auto combine = [&](vi a, vi b) {
    vi res(sz(a) + sz(b) - 1, 0);
    REP(i, sz(a)) REP(j, sz(b))
      res[i+j] = (res[i+j] + a[i]*b[j]) % mod;
    for (int i = 2*n; i > n; --i) REP(j,n)
```

```
    res[i-1-j] = (res[i-1-j] + res[i]*C[j]) % mod;
  res.rs(n + 1);
  return res;
};
vi pol(n + 1), e(pol);
pol[0] = e[1] = 1;
for (++k; k; k /= 2) {
  if (k % 2) pol = combine(pol, e);
  e = combine(e, e);
}
ll res = 0;
REP(i, n) res = (res + pol[i + 1] * A[i]) % mod;
return res;
}
```

1.8. Misc.

1.8.1. *Josephus problem.* Last man standing out of n if every kth is killed. Zero-based, and does not kill 0 on first pass. Hash: 42afc7

```
int J(int n, int k) {
  if (n == 1 || 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); }
```

- **Prime numbers**: 1031, 32771, 1048583, 33554467, 998244353, 9982451653, 1073741827, 34359738421, 1099511627791, 35184372088891, 1125899906842679, 36028797018963971, $10^{18}+7$.
  $10^3 + \{-9,-3,9,13\}$, $10^6 + \{-17,3,33\}$, $10^9 + \{7,9,21,33,87\}$.
- **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 $b_i = \sum_{j\&i=j} a_j$. Hash: 33fc1b

```
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). Examples: $998244352 = 2^{23}\cdot7\cdot17+1 : 2$, $2.013.265.921 = 2^{27}\cdot3\cdot5+1 : 31$, $10^9+7 : 5$, $10^9+9 : 13$, $10^9+21 : 2$, $10^9+33 : 5$, $10^9+87 : 3$, $36028797018963971 : 2$, $10^{18}+7 : 5$, $(10^9+7)^2 : 5$
- **Maximum number of divisors:**

$\leq N$	10^3	10^6	10^9	10^{12}	10^{18}
m	840	720720	735134400	963761198400	
$\sigma_0(m)$	32	240	1344	6270	103680

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

2. Datastructures

2.1. Order tree. Hash: 36a167

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

order_set<ll> s;
s.insert(5);
s.insert(2);
s.insert(3);
s.find_by_order(2);//5
s.order_of_key(3);//1
s.order_of_key(4);//2
```

2.2. Segment tree $\mathcal{O}(\log n)$. All intervals are right closed $[\ell, r]$.

2.2.1. *Lazy segment tree.* Allows for efficient range updates.
Hash: c39abc

```
struct node {
  int k, r, x, lazy;
  node() {}
  node(int _l, int _r) : k(_l), r(_r), x(INT_MAX),
  ↪  lazy(0){}
  node(int _l, int _r, int _x) : node(_l,_r){x=_x;}
  node(node a,node b):node(a.k,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 vi &a) : n(sz(a)), arr(4*n) {
    mk(a,0,0,n-1); }
  node mk(const vi &a, int i, int k, int r) {
    int m = (k+r)/2;
    return arr[i] = k > r ? node(k,r) :
      k == r ? node(k,r,a[k]) :
      node(mk(a,2*i+1,k,m),mk(a,2*i+2,m+1,r));
  }
  node update(int at, ll v, int i=0) {
    propagate(i);
    int hl = arr[i].k, 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 k, int r, int i=0) {
    propagate(i);
    int hl = arr[i].k, hr = arr[i].r;
    if (r < hl || hr < k) return node(hl,hr);
    if (k <= hl && hr <= r) return arr[i];
    return node(query(k,r,2*i+1),query(k,r,2*i+2));
  }
  node range_update(int k, int r, ll v, int i=0) {
    propagate(i);
    int hl = arr[i].k, hr = arr[i].r;
    if (r < hl || hr < k) return arr[i];
    if (k <= hl && hr <= r) {
      arr[i].range_update(v);
      propagate(i);
      return arr[i];
    }
    return arr[i] = node(range_update(k,r,v,2*i+1),
        range_update(k,r,v,2*i+2));
  }
  void propagate(int i) {
    if (arr[i].k < arr[i].r) {
      arr[i].push(arr[2*i+1]);
      arr[i].push(arr[2*i+2]);
    }
    arr[i].apply();
  }
};
```

2.2.2. *Persistent segment tree.* Keeps track of older versions of segment tree by id.
Hash: 885699

```
typedef ll T;
T combine(T k, T r) { return k + r; }

struct segment {
  int k, r, lid, rid;
  T val;
  segment(int _l, int _r) : k(_l), r(_r), val(0){}
};
vector&lt;segment&gt; S;

int build(int k, int r) {
```

```
  if (k > r) return -1;
  int id = sz(S);
  S.pb(segment(k,r));
  if(k != r) {
    int m = (k + r) / 2;
    S[id].lid = build(k , m);
    S[id].rid = build(m + 1, r);
  }
  return id;
}
int update(int idx, T v, int id) {//Make a[idx] = v
  if (id == -1) return -1;
  if (idx < S[id].k || idx > S[id].r) return id;
  int nid = sz(S);
  S.pb(segment(S[id].k, S[id].r));
  if(S[nid].k == S[nid].r)
    S[nid].val = v;
  else{
    int k = S[nid].lid = update(idx, v, S[id].lid);
    int r = S[nid].rid = update(idx, v, S[id].rid);
    S[nid].val = combine(S[k].val, S[r].val);
  }
  return nid;
}
T query(int id, int k, int r) {
  if (r < S[id].k || S[id].r < k) return 0;
  if (k<=S[id].k && S[id].r<=r) return S[id].val;
  return
  ↪  combine(query(S[id].lid,k,r),query(S[id].rid,k,r));
}
```

2.3. **Binary Indexed Tree $\mathcal{O}(\log n)$.** Use one-based indices ($i > 0$)! Stores and updates prefix sums efficiently. Hash: 6d8827

```
struct BIT {
  int n; vi A;
  BIT(int _n) : n(_n), A(_n+1, 0) {}
  BIT(vi &v) : n(sz(v)), A(1) {
    for (auto x:v) A.pb(x);
    for (int i=1, j; j=i&-i, i<=n; i++)
      if (i+j <= n) A[i+j] += A[i];
  }
  void update(int i, ll v) { // a[i] += v
    while (i <= n) A[i] += v, i += i&-i;
  }
  ll query(int i) { // sum_{j<=i} a[j]
    ll v = 0;
    while (i) v += A[i], i -= i&-i;
    return v;
  }
};

struct rangeBIT {
  int n; BIT b1, b2;
  rangeBIT(int _n) : n(_n), b1(_n), b2(_n+1) {}
  rangeBIT(vi &v) : n(sz(v)), b1(v), b2(sz(v)+1) {}
  void pupdate(int i, ll v) { b1.update(i, v); }
  void rupdate(int i, int j, ll v) { // a[i,..,j]+=v
    b2.update(i, v);
    b2.update(j+1, -v);
    b1.update(j+1, v*j);
    b1.update(i, (1-i)*v);
  }
  ll query(int i){return b1.query(i)+b2.query(i)*i;}
};
```

2.4. **Disjoint-Set / Union-Find $\mathcal{O}(\alpha(n))$.** Hash: 216404

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

2.5. **Cartesian tree/Treap.** Binary tree derived from a sequence of points. The root is the point with smallest y-coordinate and contains the points with smaller x in its left subtree (and larger x in right subtree). **Can be used as treap (= BST on x)**, by adding random y to each point. Additional information can be updated in augment. Can be made persistent by changing the update instructions in split, merge and erase to creating new nodes (**use keyword new**). Hash: 732435

```
struct node {
    ll x, y, size;
    node *l, *r;
    node(ll _x = 0, ll _y = 0)
      : x(_x), y(_y), size(1), l(NULL), r(NULL) { }
      ↪  };
ll tsize(node* t) { return t ? t->size : 0; }
ll tsum(node* t) { return t ? t->sum : 0; }
void augment(node *t) { //update all information
↪  here
    t->size = 1 + tsize(t->l) + tsize(t->r);
}
pair<node*,node*> split(node *t, ll x) {
    if (!t) return {NULL,NULL};
    if (t->x < x) {
        pair<node*,node*> res = split(t->r, x);
        t->r = res.x; augment(t);
        return make_pair(t, res.y); }
    pair<node*,node*> res = split(t->l, x);
    t->l = res.y; augment(t);
    return make_pair(res.x, 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, ll 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, ll x, ll y) {
    pair<node*,node*> res = split(t, x);
    return merge(res.x, merge(new node(x, y),
    ↪  res.y));
}
node* erase(node *t, ll 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;
}
ll kth(node *t, ll k) {
    if (k < tsize(t->l)) return kth(t->l, k);
    if (k == tsize(t->l)) return t->x;
    return kth(t->r, k - tsize(t->l) - 1);
}
```

Construction based on arrays $\mathcal{O}(n)$. Hash: 6d6756

```
//if points, sort(vii a) and compare .y in walk
vi a, l, r; ll root;
vi walk(vi& a, bool eq) {
    vi stack, res;
    REP(i, sz(a)) {
        ll b = -1;
        while(sz(stack) > 0) {
            if(a[i] > a[stack.back()] ||
              (eq && a[i] == a[stack.back()]))
              ↪  break;
            b = stack.back(); stack.pop_back();}
        stack.pb(i); res.pb(b);}
    return res; }
void constructtree() {
    l = walk(a, true);
    reverse(all(a));
    r = walk(a, false);
    reverse(all(r)); reverse(all(a));
    REP(i, sz(r)) if(r[i] != -1)
        r[i] = sz(a) - 1 - r[i];
    root = 0;
    REP(i,sz(a)) if(a[i] < a[root]) root = i; }
```

2.6. **Heap.** An implementation of a binary heap. Hash: a583e0

```
//heap stores keys, not values
//Use values in compare function
struct heap {
  vi q, loc;
  bool (*less) (ll, ll);
```

```
  heap(bool (*_less) (ll, ll)) : less(_less) {}
  bool cmp(int i, int j) { return less(q[i],q[j]); }
  void swp(int i, int j) {
    swap(q[i], q[j]), swap(loc[q[i]], loc[q[j]]);
  }
  void swim(int i) {
    for (int p; i; swp(i, p), i = p)
      if (!cmp(i, p=(i-1)/2)) break;
  }
  void sink(int i) {
    for (int j; (j=2*i+1)<sz(q); swp(j, i), i=j) {
      if (j+1 < sz(q) && cmp(j+1, j)) ++j;
      if (!cmp(j, i)) break;
    }
  }
  void push(int n) {
    while (n >= sz(loc)) loc.pb(-1);
    loc[n] = sz(q), q.pb(n);
    swim(sz(q) - 1);
  }
  int top() { return q[0]; }
  int pop() {
    int res = top();
    q[0] = q.back(), q.pop_back();
    loc[q[0]]=0, loc[res] = -1;
    sink(0); return res;
  }
  void heapify() {
    for (int i=sz(q); --i; )
      if (cmp(i, (i-1)/2)) swp(i, (i-1)/2);
  }
  void update_key(int n) {
    swim(loc[n]), sink(loc[n]);
  }
  int size() { return sz(q); }
  bool empty() { return !size(); }
  void clear() { q.clear(), loc.clear(); }
};
```

2.7. **Misof Tree.** A simple tree data structure for inserting, erasing, and querying the nth largest element. Hash: 1d305e

```
struct misof_tree {
  vvi cnt;
  int bits;
  misof_tree(int _bits) : bits(_bits) {
    cnt = vvi(bits, vi(1 << bits, 0)); }
  void change(int x, int d) {
    for (int i=0; i<bits; cnt[i++][x] += d, x >>=
    ↪ 1); }
  void insert(int x) { change(x,1); }
  void erase(int x) { change(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++;
    return res; } };
```

2.8. ***k*-d Tree.** A k-dimensional tree supporting fast construction, adding points, and nearest neighbor queries. NOTE: Not completely stable, occasionally segfaults. Hash: 955938

```
#define INC(c) ((c) == K - 1 ? 0 : (c) + 1)
const ld EPS = 1e-9;
template <int K> struct kd_tree {
  struct pt {
    double coord[K];
    pt() {}
    pt(double c[K]) { REP(i,K) coord[i] = c[i]; }
    double dist(const pt &other) const {
      double sum = 0.0;
      REP(i,K) sum +=
      ↪ pow(coord[i]-other.coord[i],2);
      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,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, sz(pts) - 1, 0); }
  node* construct(vector<pt> &pts, int fr, int to,
  ↪ int c) {
    if (fr > to) return NULL;
    int mid = fr + (to-fr) / 2;
    nth_element(pts.begin() + fr, pts.begin() + mid,
          pts.begin() + to + 1, cmp(c));
    return new node(pts[mid],
            construct(pts, fr, 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 same=true)
  ↪ {
    double mn = INFINITY, cs[K];
    REP(i,K) cs[i] = -INFINITY;
    pt from(cs);
    REP(i,K) cs[i] = INFINITY;
    pt to(cs);
    return _nn(p, root, bb(from, to), mn, 0,
    ↪ same).x;
  }
  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,2) {
      if (i == 1 || cmp(c)(n->p, p))
        swap(n1, n2), swap(l1, l2);
      auto res = _nn(p, n1, b.bound(n->p.coord[c],
      ↪ c, l1), mn, INC(c), same);
      if (res.y && (!found || p.dist(res.x) <
      ↪ p.dist(resp)))
        resp = res.x, found = true;
    }
    return make_pair(resp, found); } };
```

2.9. **Range Tree.** A 2-dimensional range tree supporting range queries in $O(\log(n))$ time. Hash: bf0fb1

```
struct rangetree {
    vi xtop, ytop;
    vi l, r; vvi lind, rind;
    ll base;
```

```
    rangetree(vii p) {
        sort(all(p));
        for(base = 1; base < sz(p); base *= 2);
        l = r = vi(2 * base - 1);
        lind = rind = vvi(2 * base - 1);
        ytop = build(p, 0, sz(p) - 1, 0);
        for(ii pt : p) xtop.pb(pt.x); }
    vi build(vii& p, ll _l, ll _r, ll i) {
        l[i] = _l, r[i] = _r;
        if(_l == _r) { return {p[_l].y}; }
        ll m = (_l + _r) / 2;
        vi left = build(p, _l, m, 2 * i + 1), right
        ↪  = build(p, m + 1, _r, 2 * i + 2);
        ll il = 0, ir = 0; vi res;
        while(il < sz(left) || ir < sz(right)) {
            lind[i].pb(il); rind[i].pb(ir);
            if(il < sz(left) && (ir == sz(right) ||
            ↪  left[il] <= right[ir])) {
                res.pb(left[il]);
                il++; }
            else {
                res.pb(right[ir]);
                ir++; } }
        lind[i].pb(il); rind[i].pb(ir); return res;
        ↪  }
    ll nexti(vi& a, ll v) {//first i with a[i] >= v
        ll k = -1, r = sz(a), m;
        while(r - k > 1) {
            m = (k + r) / 2;
            if(a[m] < v) k = m;
            else r = m; }
        return r; }
    ll q(ll iy, ll _l, ll _r, ll i) {
        if(l[i] > _r || r[i] < _l) return 0;
        if(l[i] >= _l && r[i] <= _r) return iy;
        return q(lind[i][iy], _l, _r, 2 * i + 1) +
        ↪  q(rind[i][iy], _l, _r, 2 * i + 2); }
    //query #points in [xl, xr] x [yl, yr]
    ll query(ll xl, ll xr, ll yl, ll yr) {
        ll k = nexti(xtop, xl), r = nexti(xtop, xr +
        ↪  1) - 1;
        ll y1 = nexti(ytop, yl), y2 = nexti(ytop, yr
        ↪  + 1);
        return q(y2, k, r, 0) - q(y1, k, r, 0); }};
```

2.10. **Monotonic Queue.** A queue that supports querying for the minimum element. Useful for sliding window algorithms. Hash: 112812

```
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();
  ↪  }
};
```

2.11. **Line container à la 'Convex Hull Trick' $\mathcal{O}(n \log n)$.** Container where you can add lines of the form $y_i(x) = k_i x + m_i$ and query $\max_i y_i(x)$. Hash: 819ac5

```
bool Q;
struct Line {
  mutable ll k, m, p;
  bool operator<(const Line& o) const {
    return Q ? p < o.p : k < o.k;
  }
};
struct LineContainer : multiset<Line> {
  // (for doubles, use inf = 1/.0, div(a,b) = a/b)
  const ll inf = LLONG_MAX;
  ll div(ll a, ll b) { // floored division
    return a / b - ((a ^ b) < 0 && a % b); }
  bool isect(iterator x, iterator y) {
    if (y == end()) { x->p = inf; return false; }
    if (x->k == y->k)
      x->p = x->m > y->m ? inf : -inf;
    else
      x->p = div(y->m - x->m, x->k - y->k);
    return x->p >= y->p;
  }
  void add(ll k, ll m) {
    auto z = insert({k, m, 0}), y = z++, x = y;
    while (isect(y, z)) z = erase(z);
    if (x != begin() && isect(--x, y))
      isect(x, y = erase(y));
    while ((y = x) != begin() && (--x)->p >= y->p)
      isect(x, erase(y));
  }
  ll query(ll x) {
    Q=1; auto l = *lower_bound({0,0,x}); Q=0;
    return l.k * x + l.m;
  }
};
```

2.12. **Li-Chao tree.** Tree where you can add pseudolines in $O(\log(n))$ and query the maximum line for values $a_1 < a_2 < \cdots < a_n$ in $O(\log(n))$. 2 pseudolines can intersect at most once. Hash: 5b862b

```
struct line { //Can be any pseudoline
    ll a, b;
    line(): a(0), b(0) {}
    line(ll _a, ll _b): a(_a), b(_b) {}
    bool overtakes(line l) { return a > l.a; }
    ll value(ll i) { return a * i + b; }
};

struct LiChaoTree {
    ll width;
    vector<line> tree; vi v;
    LiChaoTree(vi a) { //any increasing sequence
        for(width = 1; width < sz(a); width *= 2) ;
        v = vi(2 * width - 1);
        tree = vector<line>(2 * width - 1);
        REP(i, width)
          v[i + width - 1] = a[min(i, sz(a) - 1)];
        for(ll i = width - 2; i >= 0; i--)
          v[i] = v[2 * i + 2];
        for(ll i = 0; i < width - 1; i++)
          v[i] = v[2 * i + 1]; }
    void insert(line& l, ll i = 0) {
        if(i >= 2 * width - 1) return;
        line cur = tree[i];
        if(l.value(v[i]) > cur.value(v[i])) {
            tree[i] = l;
            swap(l,cur); }
        if(l.overtakes(cur)) insert(l, 2 * i + 2);
        else insert(l, 2 * i + 1); }
    ll query(ll i) { //query maximum value at a[i]
        ll k = (i + width - 1);
        ll res = tree[k].value(i);
        while(k > 0) {
            k = (k - 1) / 2;
            res = max(res, tree[k].value(i)); }
        return res; }
};
```

2.13. **Sparse Table $O(\log n)$ per query.** Static range queries. Hash: 5284dd

```
struct sparse_table {
  vvi m;
  sparse_table(vi arr) {
    m.pb(arr);
    for (int b=0; (1<<(++b)) <= sz(arr); ) {
      int w = (1<<b), hw = w/2;
      m.pb(vi(sz(arr) - w + 1));
      for (int i = 0; i+w <= sz(arr); i++) {
        m[b][i] = min(m[b-1][i], m[b-1][i+hw]);
      }
    }
  }
  int query(int k, int r) { // query min in [l,r]
    int b = 31 - __builtin_clz(r-k);
    // for (b = 0; 1<<(b+1) <= r-k+1; b++);
    return min(m[b][k], m[b][r-(1<<b)+1]);
```

```
  }
};
```

3. Graph Algorithms

3.1. Shortest path.

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

3.1.2. *Floyd-Warshall* $\mathcal{O}(V^3)$ *all pairs.*
Be careful with negative edges! Note: |d[i][j]| can grow exponentially, and INFTY + negative < INFTY. Hash: dc6ed3

```
const ll INF = 1LL << 61;
void floyd_warshall( vvi& d ) {
  ll n = sz(d);
  REP(i,n) REP(j,n) REP(k,n)
    if(d[j][i] < INF && d[i][k] < INF) // neg edges!
      d[j][k] = max(-INF,
        min(d[j][k], d[j][i] + d[i][k]));
}
```

3.1.3. *Bellman Ford* $\mathcal{O}(VE)$.
This is only useful if there are edges with weight $w_{ij} < 0$ in the graph.

3.2. Maximum Matching.

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 of vertices such that each edge in the graph is incident to at least one vertex of the set.

Minimum Edge Cover: A set of edges 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.

Minimum edge cover $\iff$ Maximum independent set.

König's theorem: In any bipartite graph $G = (L \cup R, E)$, the number of edges in a maximum matching equals 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.

In any bipartite graph,

$$\text{maxmatch} = \text{MVC} = V - \text{MIS}.$$

See 3.2.2.

3.2.1. *Standard bipartite matching* $\mathcal{O}(|L|\cdot|R|)$. Hash: 9687e2

```
vb vis; vi L, R; vvi G; // L->{R,...}
void addedge(int a, int b) { G[a].pb(b);}
bool match(int u) {
  for (int v : G[u]) {
    if (vis[v]) continue;
    vis[v] = true;
    if (R[v] == -1 || match(R[v]))
    { R[v] = u, L[u] = v; return true; }
  }
  return false; }
// perfect matching iff ret == n == m
int maxmatch(int n, int m) {
  L.assign(n, -1);
  R.assign(m, -1);
  int ret = 0;
  REP(i, n) vis.assign(m, false), ret += match(i);
  return ret; }
```

3.2.2. *Hopcroft-Karp bipartite matching* $\mathcal{O}(E\sqrt{V})$. Hash: 2ad7e5

```
struct bigraph {
  int n, m, s; vvi G; vi L, R, d;
  bigraph(int _n, int _m) : n(_n), m(_m), s(0),
      G(n), L(n,-1), R(m,n), d(n+1) {}
  void addedge(int a, int b) { G[a].pb(b); }
  bool bfs() {
    queue<int> q; d[n] = LLONG_MAX;
    REP(v, n)
      if (L[v] < 0) d[v] = 0, q.push(v);
      else d[v] = LLONG_MAX;
    while (!q.empty()) {
      int v = q.front(); q.pop();
      if (d[v] >= d[n]) continue;
      for (int u : G[v]) if (d[R[u]] == LLONG_MAX)
        d[R[u]] = d[v]+1, q.push(R[u]);
    }
    return d[n] != LLONG_MAX;
  }
  bool dfs(int v) {
    if (v == n) return true;
    for (int u : G[v])
      if (d[R[u]] == d[v]+1 && dfs(R[u])) {
        R[u] = v; L[v] = u; return true;
      }
    d[v] = LLONG_MAX; return false;
  }
  int maxmatch() {
    while (bfs()) REP(i,n) s += L[i]<0 && dfs(i);
    return s;
  }

  void dfs2(int v, vb &alt) {
    alt[v] = true;
    for (int u : G[v]) {
      alt[u+n] = true;
      if (R[u] != n && !alt[R[u]]) dfs2(R[u], alt);
    }
  }

  vi minvertexcover() {
    vb alt(n+m, false); vi res;
    maxmatch();
    REP(i, n) if (L[i] < 0) dfs2(i, alt);
    // !alt[i] (i<n) OR alt[i] (i >= n)
    REP(i, n+m) if (alt[i] != (i<n)) res.pb(i);
    return res;
  }
};
```

3.2.3. *Blossom matching* $\mathcal{O}(EV^2)$. Hash: 3ce0fb

```
vb marked;
vvb emarked;
vi S;
ll n;
vi find_augmenting_path(const vvi &adj,const vi &m){
  int n = sz(adj), s = 0;
  vi par(n,-1), height(n), root(n,-1), q, a, b;
  marked = vb(n,false);
  emarked = vvb(n,vb(n,false));
  REP(i,n) if (m[i] >= 0) emarked[i][m[i]] = true;
           else root[i] = i, S[s++] = i;
  while (s) {
    int v = S[--s];
    for(ll w:adj[v]) {
      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.pb(v), v = par[v];
          reverse(all(q));
          while(w != -1) q.pb(w), w = par[w];
          return q;
        } else {
          int c = v;
          while(c != -1) a.pb(c), c = par[c];
          c = w;
          while(c != -1) b.pb(c), c = par[c];
          while(!a.empty() && !b.empty() && a.back()
          ↪ == b.back())
            c = a.back(), a.pop_back(),
            ↪ b.pop_back();
          marked = vb(n,false);
          fill(par.begin(), par.end(), 0);
          for(ll it : a) par[it] = 1; for(ll it : b)
          ↪ par[it] = 1;
          par[c] = s = 1;
          REP(i,n) root[par[i] = par[i] ? 0 : s++] =
          ↪ i;
```

```
          vvi adj2(s);
          REP(i,n) for(ll it : adj[i]) {
            if (par[it] == 0) continue;
            if (par[i] == 0) {
              if (!marked[par[it]])  {
                adj2[par[i]].pb(par[it]);
                adj2[par[it]].pb(par[i]);
                marked[par[it]] = true; }
            } else adj2[par[i]].pb(par[it]); }
          vi m2(s, -1);
          if (m[c] != -1) m2[m2[par[m[c]]] = 0] =
          ↪  par[m[c]];
          REP(i,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 < sz(p) && p[t]) t++;
          if (t == sz(p)) {
            REP(i,sz(p)) p[i] = root[p[i]];
            return p; }
          if (!p[0] || (m[c] != -1 && p[t+1] !=
          ↪  par[m[c]]))
            reverse(all(p)), t = sz(p)-t-1;
          rep(i,0,t) q.pb(root[p[i]]);
          for(ll it : adj[root[p[t-1]]]) {
            if (par[it] != (s = 0)) continue;
            a.pb(c), reverse(all(a));
            for(ll jt : b) a.pb(jt);
            while (a[s] != it) s++;
            if ((height[it] & 1) ^ (s < sz(a) -
            ↪  sz(b)))
              reverse(all(a)), s = sz(a)-s-1;
            while(a[s]!=c) q.pb(a[s]), s=(s+1) %
            ↪  sz(a);
            q.pb(c);
            rep(i,t+1,sz(p)) q.pb(root[p[i]]);
            return q; } } }
      emarked[v][w] = emarked[w][v] = true; }
    marked[v] = true; } return q; }
vii max_matching(const vvi &adj) {
  n = sz(adj);
  marked = vb(n);
  emarked = vvb(n,vb(n));
  S = vi(n);
  vi m(sz(adj), -1), ap; vii res, es;
  REP(i,sz(adj)) for(ll it:adj[i]) es.eb(i,it);
  random_shuffle(all(es));
  for(ii it: es) if (m[it.x] == -1 && m[it.y] == -1)
    m[it.x] = it.y, m[it.y] = it.x;
  do { ap = find_augmenting_path(adj, m);
       REP(i,sz(ap)) m[m[ap[i^1]] = ap[i]] =
       ↪  ap[i^1];
  } while (!ap.empty());
  REP(i,sz(m)) if (i < m[i]) res.eb(i, m[i]);
  return res; }
```

3.2.4. *Stable marriage.* With n men, $m \geq n$ women, n preference lists of women for each men, and for every woman j an preference of men defined by `pref[][j]` (lower is better) find for every man a women such that no pair of a men and a woman want to run off together. Hash: a74574

```
// n = aantal mannen, m = aantal vrouwen
// voor een man i, is order[i] de prefere
vi stable(int n, int m, vvi order, vvi pref) {
  queue<int> q;
  REP(i, n) q.push(i);
  vi mas(m,-1), mak(n,-1), p(n,0);
  while (!q.empty()) {
    int k = q.front();
    q.pop();
    int s = order[k][p[k]], k2 = mas[s];
    if (mas[s] == -1) {
      mas[s] = k;
      mak[k] = s;
    } else if (pref[k][s] < pref[k2][s]) {
      mas[s] = k;
      mak[k] = s;
      mak[k2] = -1;
      q.push(k2);
    } else {
      q.push(k);
    }
    p[k]++;
  }
  return mak;
}
```

3.3. Depth first searches.

3.3.1. *Topological Sort* $O(V+E)$. Hash: be0f5e

```
vi topo(vvi &adj) { // requires C++14
  int n=sz(adj); vb vis(n,0); vi ans;
  auto dfs = [&](int v, const auto& f)->void {
    vis[v] = true;
    for (int w : adj[v]) if (!vis[w]) f(w, f);
    ans.pb(v);
  };
  REP(i, n) if (!vis[i]) dfs(i, dfs);
  reverse(all(ans));
  return ans;
}
```

3.4. Cycle Detection $\mathcal{O}(V+E)$. Hash: 566bb0

```
vvi G;
vb vis, done;
vi p;
ii backedge(ll i) {
  vis[i] = true;
  for(ll j : G[i])
    if(!vis[j]) {
      p[j] = i;
      ii antw = backedge(j);
      if(antw.x != -1) return antw;
    }
    else if(!done[j]) return {i,j};
  done[i] = true;
  return {-1,-1}; }
//directed
vi findcycledir() {
  ll n = sz(G);
  vis = vb(n,false), done = vb(n,false);
  p = vi(n,-1);
  REP(i,n) if(!vis[i]) {
    ii antw = backedge(i);
    if(antw.x != -1) {
      vi c; ll v = antw.x, w = antw.y;
      c.pb(v);
      while(v != w) c.pb(v = p[v]);
      reverse(all(c));
      return c;
    }
  }
  return {}; }

//undirected
vi findcycleundir(const vvi &G, int v0) {
  vi p(sz(G), -1), s{v0};
  while (!s.empty()) {
    int v = s.back(); s.pop_back();
    for (int w : G[v])
      if (p[w] == -1) s.pb(w), p[w] = v;
      else if (w != p[v]) {
        vi c;
        while (v != w) c.pb(v = p[v]);
        return c;
      }
  }
  return {}; }
```

3.4.1. *Cut Points and Bridges* $O(V+E)$. Vertices/edges that when removed split their connected component in two. Hash: d21429

```
vi low, num;
int curnum;

void dfs(vvi &adj, vi &cp, vii &bs, int u, int p) {
  low[u] = num[u] = curnum++;
  int cnt = 0; bool found = false;
  REP(i, sz(adj[u])) {
    int v = adj[u][i];
    if (num[v] == -1) {
      dfs(adj, cp, bs, v, u);
      low[u] = min(low[u], low[v]);
      cnt++;
      found = found || low[v] >= num[u];
      if (low[v] > num[u]) bs.eb(u, v);
    } else if (p != v) low[u] = min(low[u], num[v]);
  }
```

```
  if (found && (p != -1 || cnt > 1)) cp.pb(u);
}

pair<vi,vii> cut_points_and_bridges(vvi &adj) {
  int n = sz(adj);
  vi cp; vii bs;
  num = vi(n, -1); low = vi(n);
  curnum = 0;
  REP(i,n) if (num[i] < 0) dfs(adj, cp, bs, i, -1);
  return make_pair(cp, bs);
}
```

3.4.2. *Strongly Connected Components* $\mathcal{O}(V + E)$. Hash: d35c19

```
struct SCC {
  int n, age=0, ncomps=0; vvi adj, comps;
  vi tidx, lnk, cnr, st; vb vis;
  SCC(vvi &_adj) : n(sz(_adj)), adj(_adj),
      tidx(n, 0), lnk(n), cnr(n), vis(n, false) {
    REP(i, n) if (!tidx[i]) dfs(i);
  }

  void dfs(int v) {
    tidx[v] = lnk[v] = ++age;
    vis[v] = true; st.pb(v);
    for (int w : adj[v]) {
      if (!tidx[w])
        dfs(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++;
  }
};
```

3.4.3. *2-SAT* $\mathcal{O}(V + E)$. Uses SCC. Hash: 943a72

```
struct TwoSat {
  int n; SCC *scc = nullptr; vvi adj;
  TwoSat(int _n) : n(_n), adj(_n*2, vi()) {}
  ~TwoSat() { delete scc; }

  // a => b, i.e. b is true or ~a
  void imply(int a, int b) {
    adj[n+a].pb(n+b); adj[n+(~b)].pb(n+(~a)); }
  void OR(int a, int b) { imply(~a, b); }
  void CONST(int a) { OR(a, a); }
  void IFF(int a, int b) { imply(a,b); imply(b,a); }

  bool solve(vb &sol) {
    delete scc; scc = new SCC(adj);
    REP(i, n) if (scc->cnr[n+i] == scc->cnr[n+(~i)])
      return false;
    vb seen(n, false);
    sol.assign(n, false);
    for (vi &cc : scc->comps) for (int v : cc) {
      int i = v<n ? n + (~v) : v - n;
      if (!seen[i]) seen[i]=true, sol[i] = v>=n;
    }
    return true;
  }
};
```

3.4.4. *Dominator graph.*

- A node d dominates a node n if every path from the entry node to n must go through d.
- The immediate dominator (idom) of a node n is the unique node that strictly dominates n but does not strictly dominate any other node that strictly dominates n.

Hash: f1cfea

```
vvi g, grev, bucket;
vi pos, order, par, sdom, p, best, idom, lnk;
int cnt;

void create(ll n) {
  g = vvi(n), grev = vvi(n), bucket = vvi(n);
  pos = vi(n, -1), order = vi(n), par = vi(n), sdom
   ↪  = vi(n);
  p = vi(n), best = vi(n), idom = vi(n), lnk =
   ↪  vi(n);
}

void addedge(int a, int b) {
  g[a].pb(b), grev[b].pb(a);
}

void dfs(int v) {
  pos[v] = cnt;
  order[cnt++] = v;
  for (int u : g[v])
    if (pos[u] < 0) par[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) {
  pos = vi(n,-1);
  cnt = 0;
  dfs(root);
  REP(i, n) p[i] = best[i] = sdom[i] = i;

  for (int it = cnt - 1; it >= 1; it--) {
    int w = order[it];
    for (int u : grev[w]) {
      if(pos[u] == -1) continue;
      int t = find_best(u);
      if (pos[sdom[t]] < pos[sdom[w]])
        sdom[w] = sdom[t];
    }
    bucket[sdom[w]].pb(w);
    idom[w] = sdom[w];
    for (int u : bucket[par[w]])
      lnk[u] = find_best(u);
    bucket[par[w]].clear();
    p[w] = par[w];
  }

  for (int it = 1; it < cnt; it++) {
    int w = order[it];
    idom[w] = idom[lnk[w]];
  }
  REP(i,n) if(pos[i] == -1) idom[i] = -1;
  idom[root] = root;
}
```

3.5. Min Cut / Max Flow.

3.5.1. *Dinic's Algorithm* $\mathcal{O}(V^2E)$. Hash: 90ea92

```
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(sz(G)); G.pb({v, c, 0LL});
    E[v].pb(sz(G)); G.pb({u, 0LL, 0LL});
  }
  ll dfs(int t, int v, ll f) {
    if (v == t || !f) return f;
    for ( ; P[v] < sz(E[v]); 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;
  }
  void bfs(int s) {
    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);
    }
  }
  ll maxflow(int s, int t, ll f = 0) {
    while (1) {
      bfs(s);
      if (!H[t]) return f;
      fill(all(P), 0);
      while (ll df = dfs(t, s, LLONG_MAX)) f += df;
    }
  }
  vb mincut(int s, int t) {
    maxflow(s,t);
    bfs(s);
    vb antw(sz(H));
    REP(i, sz(H)) antw[i] = !H[i];
    return antw;
  }
  void resetflow() {
    REP(i, sz(G)) G[i].f = 0;
  }
};
```

3.5.2. *Min-cost max-flow* $O(n^2m^2)$. Find the cheapest possible way of sending a certain amount of flow through a flow network. Hash: 7558bd

```
struct edge { ll x, y, f, c, w; };
ll V; vi par, D; vector<edge> g;
vvi e;
void create(ll n)  {
  V = n; e = vvi(n);
  par = vi(n), D = vi(n);
}
inline void addEdge(int u, int v, ll c, ll w) {
  e[u].pb(sz(g));  g.pb({u, v, 0, c, w});
  e[v].pb(sz(g)); g.pb({v, u, 0, 0, -w});
}

void spBF(int s) {
  D = vi(V,LLONG_MAX); D[s] = 0;
  for (int ng = sz(g), _ = V; _--; ) {
    bool ok = false;
    for (int i = 0; i < ng; i++)
      if (D[g[i].x] != LLONG_MAX && 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;
  }
}

//Can be omitted if n small enough
void spDijk(int s) {
  vi ed(V,LLONG_MAX); ed[s] = 0;
  set<ii> front{ii(0,s)};
  while(sz(front) > 0) {
    ll v = front.begin()->y;
    front.erase(front.begin());
    for(ll i : e[v]) if(g[i].f < g[i].c) {
      ll y = g[i].y, now = g[i].w + ed[v] - D[y] +
      ↪  D[v];
      if(now < ed[y]) {
        front.erase(ii(ed[y],y));
        ed[y] = now;
        front.emplace(ii(now,y));
        par[y] = i;
      }
    }
  }
  REP(i,V)
    if(ed[i] < LLONG_MAX) D[i] += ed[i];
    else D[i] = LLONG_MAX;
}

void minCostMaxFlow(int s, int t, ll &c, ll &f) {
  spBF(s);
  for (c = f = 0; spDijk(s), D[t] < LLONG_MAX; ) {
    ll df = LLONG_MAX, 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;
  }
}
```

3.5.3. *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 or produces the correct min cut. DOES NOT WORK FOR DIRECTED GRAPHS.** Hash: b2364a

```
struct GHTree {
  Dinic d; int n;
  vvii tree;
  GHTree(int _n) : n(_n), d(_n){ }
  void addEdge(int u, int v, int c) {
    d.addEdge(u, v, c); d.G.back().c = c;
  }
  ll build(vi& nodes) {
    if(sz(nodes) == 1) return nodes[0];
    d.resetflow(); ll f = d.maxflow(nodes[0],
    ↪  nodes[1]);
    vb cut = d.mincut(nodes[0], nodes[1]);
    vi n1, n2;
    for(ll i : nodes)
      if(cut[i]) n1.pb(i);
      else n2.pb(i);
    ll p1 = build(n1), p2 = build(n2);
    tree[p1].eb(p2, f);
    tree[p2].eb(p1, f);
    if(cut[0]) return p1;
    return p2;
  }
  void buildTree() {
    tree = vvii(n);
    vi nodes; REP(i, n) nodes.pb(i);
    build(nodes);
  }
  void dfsedge(int i, int p,
  ↪  vector<tuple<int,int,ll>>& prev) {
    for(ii e : tree[i])
      if(e.x != p) {
        prev[e.x] = {i, e.x, e.y};
        dfsedge(e.x, i, prev);
      }
  }
  tuple<int, int, ll> findtreecut(int s, int t) {
    vector<tuple<int,int,ll>> prev(n); dfsedge(s,
    ↪  -1, prev);
    int small = t;
    for(int c = t; c != s; c = get<0>(prev[c]))
      if(get<2>(prev[c]) < get<2>(prev[small]))
        small = c;
    return prev[small];
  }
  ll maxflow(int s, int t) {
    auto e = findtreecut(s, t); return get<2>(e);
  }
  void dfscut(int i, int p, vb& cut) {
    cut[i] = true;
    for(ii e : tree[i])
      if(e.x != p) dfscut(e.x, i, cut);
  }
  vb mincut(int s, int t) {
    auto e = findtreecut(s, t);
    vb cut = vb(n, false); dfscut(get<0>(e),
    ↪  get<1>(e), cut);
    return cut;
  }
};
```

3.6. **Minimal Spanning Tree** $\mathcal{O}(E \log V)$**.** Hash: 17c710

```
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;
}
```

3.7. Euler Path $O(V+E)$ hopefully. Finds an Euler Path (or circuit) in a *directed* graph iff one exists. Hash: 2447dd

```
vvi adj;
int n, m;
vi indeg, outdeg, res;
ii start_end() {
  int start = -1, end = -1, any = 0, c = 0;
  REP(i, 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))
    return ii(-1,-1);
  if (start == -1) start = end = any;
  return ii(start, end); }
void makepath(ll i) {
  while(outdeg[i] > 0)
  ↪  makepath(adj[i][--outdeg[i]]);
  res.pb(i);
}
bool euler_path() {
  ii se = start_end();
  if (se.x == -1) return false;
  makepath(se.x); reverse(all(res));
  return (sz(res) == m + 1);
}
```

Finds an Euler Path (or circuit) in a *undirected* graph: Hash: b1890e

```
vector<multiset<int>> adj;
int n, m;
vi res;
ii start_end() {
  vi odd; int any = 0;
  REP(i,n) {
    if(sz(adj[i]) % 2 == 1) odd.pb(i);
    if(sz(adj[i]) > 0) any = i;
  }
  if(sz(odd) == 2) return ii(odd[0],odd[1]);
  if(sz(odd) == 0) return ii(any,any);
  return ii(-1,-1); }
void makepath(ll i) {
  while(sz(adj[i]) > 0) {
    ll j = *adj[i].begin();
    adj[i].erase(adj[i].find(j));
    adj[j].erase(adj[j].find(i));
    makepath(j); }
  res.pb(i);
}
bool euler_path() {
  ii se = start_end();
  if (se.x == -1) return false;
  makepath(se.x); reverse(all(res));
  return (sz(res) == m + 1); }
```

3.8. Heavy-Light Decomposition. Hash: ad4690

```
struct HLD {
  vvi adj; int cur_pos = 0;
  vi par, dep, hvy, head, pos;
  segmenttree st;

  HLD(int n, const vvi &A) : adj(all(A)), par(n),
    dep(n), hvy(n,-1), head(n), pos(n), st(n) {
    cur_pos = 0; dfs(0); decomp(0, 0);
  }

  int dfs(int v) { // determine parent/depth/sizes
    int wei = 1, mw = 0;
    for (int c : adj[v]) if (c != par[v]) {
      par[c] = v, dep[c] = dep[v]+1;
      int w = dfs(c);
      wei += w;
      if (w > mw) mw = w, hvy[v] = c;
    }
    return wei;
  }

  // pos: index in SegmentTree, head: root of path
  void decomp(int v, int h) {
    head[v] = h, pos[v] = cur_pos++;
    if (hvy[v] != -1) decomp(hvy[v], h);
    for (int c : adj[v])
      if (c != par[v] && c != hvy[v]) decomp(c, c);
  }

  void update(int i, ll v){ st.update(pos[i], v); }

  // requires queryST(a, b) = SUM{A[i] | a<=i<=b }.
  ll query(int a, int b) {
    ll res = 0;
    for (; head[a] != head[b]; b = par[head[b]]) {
      if (dep[head[a]] > dep[head[b]]) swap(a, b);
      res += st.query(pos[head[b]],pos[b]);
    }
    if (dep[a] > dep[b]) swap(a, b);
    return res + st.query(pos[a], pos[b]);
  }
};
```

3.9. Centroid Decomposition. Hash: 168da7

```
struct centroid_decomposition {
  int n; vvi adj;
  vvi parent, dist;
  vi sz, sepdepth;
  vvi children, tree;
  int logn, center;
  centroid_decomposition(int _n): n(_n), adj(n) {
    for(logn = 0; (1 << logn) < n; logn++);
    logn++;
    parent = dist = children = tree = vvi(n,
    ↪  vi(logn));
    sz = sepdepth = vi(n);
  }
  void add_edge(int a, int b) {
    adj[a].pb(b); adj[b].pb(a);
  }
  int dfs(int u, int p) {
    sz[u] = 1;
    for(int v : adj[u])
      if(v != p)
        sz[u] += dfs(v, u);
    return sz[u];
  }
  void makepaths(int sep, int u, int p, int len) {
    parent[u][sepdepth[sep]] = sep,
    ↪  dist[u][sepdepth[sep]] = len;
    int bad = -1;
    REP(i, sz(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();
  }
  int findcentroid(int u, int sep) {
    for(int v : adj[sep])
      if(sz[v] < sz[sep] && sz[v] > sz[u] / 2)
        return findcentroid(u, v);
    return sep;
  }
  int separate(int h, int u) {
    dfs(u, -1); int sep = findcentroid(u, u);
    sepdepth[sep] = h, makepaths(sep, sep, -1, 0);
    for(int v : adj[sep]) separate(h + 1, v);
    return sep;
  }
  void makeDecomp() {
    center = separate(0,0);
    REP(i,n) children[i].clear(), tree[i].clear();
    REP(i,n) {
      if(sepdepth[i] != 0)
        children[parent[i][sepdepth[i] - 1]].pb(i);
      REP(j, sepdepth[i] + 1)
          tree[parent[i][j]].pb(i); }
  }
};
```

3.10. Least Common Ancestors, Binary Jumping. Hash: d680d3

```
ll n, logn;
vi P; vvi BP; vi H;
//n, P, H input, assert p[root] = root
void initLCA() {
  for(logn = 0; (1 << (logn++)) < n; ) ;
  BP = vvi(n, vi(logn));
```

```
  REP(i, n) BP[i][0] = P[i];
  rep(j, 1, logn) REP(i, n)
    BP[i][j] = BP[BP[i][j-1]][j-1];
}
int query(int a, int b) {
  if (H[a] > H[b]) swap(a, b);
  int dh = H[b] - H[a], j = 0;
  REP(i, logn) 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];
}
```

3.11. Miscellaneous.

3.11.1. *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. Hash: 1cff2f

```
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, sz(G[v])});
    G[v].pb({u, -1, sz(G[u]) - 1});
  }

  void color(int v, int i) {
    vi fan = { i };
    vb used(sz(G[v]));
    used[i] = true;
    for (int j = 0; j < sz(G[v]); 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, sz(G[v]) + 1);
    F = vvi(N, vi(K, -1));
    REP(v, N) for (int i = sz(G[v]); i--; )
      if (G[v][i].col < 0) color(v, i);
  }
};
```

3.11.2. *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. $\mathcal{O}(EV)$ runtime. Hash: 9e9183

```
double min_mean_cycle(vector<vector<pair<int,ld>>>
↪  adj){
  int n = sz(adj); ld mn = INFINITY;
  vvd arr(n+1, vd(n, mn));
  arr[0][0] = 0;
  rep(k,1,n+1) REP(j,n) for(auto p : adj[j])
    arr[k][p.x] = min(arr[k][p.x],p.y +
    ↪  arr[k-1][j]);
  REP(k,n) {
    ld mx = -INFINITY;
    REP(i,n) mx = max(mx,
    ↪  (arr[n][i]-arr[k][i])/(n-k));
    mn = min(mn, mx); }
  return mn; }
```

3.11.3. *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!
$\mathcal{O}(V^2 \log V)$ **runtime and** $\mathcal{O}(E)$ **memory:** Hash: ea35fc

```
const ll oo = 1e9;
int N, R;
vvii g;
vi pred, label, node, helper;

int get_node(int n) {
  return node[n] == n ? n :
      (node[n] = get_node(node[n]));
}

ll update_node(int n) {
  ll m = oo;
  for (auto ed : g[n]) m = min(m, ed.y);
  REP(j, sz(g[n])) {
    g[n][j].y -= m;
    if (g[n][j].y == 0)
      pred[n] = g[n][j].x;
  }
  return m;
}

ll cycle(vi &active, int n, int &cend) {
  n = get_node(n);
  if (label[n] == 1) return false;
  if (label[n] == 2) { cend = n; return 0; }

  active.pb(n);
  label[n] = 2;
  auto res = cycle(active, pred[n], cend);
  if (cend == n) {
    int F = find(all(active), n)-active.begin();
    vi todo(active.begin() + F, active.end());
    active.rs(F);
    vii newg;
    for (auto i: todo) node[i] = n;
    for (auto i: todo) for(auto &ed : g[i])
      helper[ed.x] = get_node(ed.x)] = ed.y;
    for (auto i: todo) for(auto ed : g[i])
      helper[ed.x] = min(ed.y, helper[ed.x]);
    for (auto i: todo) for(auto ed: g[i]) {
      if (helper[ed.x] != oo && ed.x != n) {
        newg.eb(ed.x, helper[ed.x]);
        helper[ed.x] = oo;
      }
    }
    g[n] = newg;
    res += update_node(n);
    label[n] = 0;
    cend = -1;
    return cycle(active, n, cend) + res;
  }
  if (cend == -1) {
    active.pop_back();
    label[n] = 1;
  }
  return res;
}

// Calculates value of minimal arborescence from R,
// assuming it exists.
// adj[i] contains (j, v) with edge i -> j with
↪  value v
// pred[i] is parent in arborescence
ll min_arbor(vvii& adj, int r) {
  N = sz(adj); R = r; g = vvii(N);
  REP(i, N) for(ii p : adj[i]) g[p.x].eb(i,p.y);
  pred = label = node = helper = vi(N);
  ll res = 0;
  REP(i, N) {
    node[i] = i;
    if (i != R) res += update_node(i);
```

```
  }
  REP(i, N) label[i] = (i==R);
  REP(i, N) {
    if (label[i] == 1 || get_node(i) != i)
      continue;
    vi active;
    int cend = -1;
    res += cycle(active, i, cend);
  }
  return res;
}
```

3.11.4. *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).

3.11.5. *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.

3.11.6. *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.

3.11.7. *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.** Node content derived from position in tree, not in node. E.g. for each node a child per next character and then deriving the string from the path from the root. Hash: c1d464

```
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 $\mathcal{O}(n)$.** Hash: c038c2

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

4.3. **Manacher algorithm $\mathcal{O}(n)$.** Returns longest palindrome centered at letter i (index $2 \cdot i$) or between letters i and $i+1$ (index $2 \cdot i + 1$). Hash: 42bcf9

```
vi manacher(const string& s) {
    ll n = sz(s); vi res(2 * n - 1);
    ll k = 0, r = 0;
    REP(i, 2 * n - 1) {
        res[i] = i % 2;
        if(r > i) res[i] = res[r + k - i];
        if(i + res[i] >= r) {
            r = i + res[i];
            k = i - res[i];
            while(k >= 0 && r <= 2 * n - 2 && s[k /
            ↪  2] == s[r / 2])
                r += 2, k -= 2;
            r -= 2; k += 2;
            res[i] = r - i; } }
    REP(i, 2 * n - 1) res[i]++;
    return res; }
```

4.4. **Suffix array $\mathcal{O}(n \log n)$.** Lexicographically sorts the cyclic shifts of S where $p[0]$ is the index of the smallest string, etc. Efficient lookup of all indices where a substring occurs. Hash: 2b9060

```
vi sort_cyclic_shifts(const string &s) {
  const int alphabet = 256, n = sz(s);

  vi p(n), c(n), cnt(max(alphabet, n), 0);
  REP(i, n) cnt[s[i]]++;
  partial_sum(all(cnt), cnt.begin());
  REP(i, n) p[--cnt[s[i]]] = i;
  c[p[0]] = 0;
  int cl = 1;
  rep(i,1,n) {
    if (s[p[i]] != s[p[i-1]]) cl++;
    c[p[i]] = cl - 1;
  }

  vi pn(n), cn(n);
  for (int h = 0, l = 1; l < n; l*=2, ++h) {
    REP(i, n) {
      pn[i] = p[i] - (1 << h);
      if (pn[i] < 0) pn[i] += n;
    }
    fill(cnt.begin(), cnt.begin() + cl, 0);
    REP(i, n) cnt[c[pn[i]]]++;
    rep(i,1,cl) cnt[i] += cnt[i-1];
    for (int i = n-1; i >= 0; i--)
      p[--cnt[c[pn[i]]]] = pn[i];
    cn[p[0]] = 0;
    cl = 1;
    rep(i, 1, n) {
      if (c[p[i]] != c[p[i-1]] || c[(p[i]+l)%n]
          != c[(p[i-1]+l)%n]) cl++;
      cn[p[i]] = cl - 1;
    }
    c.swap(cn);
  }
  return p;
}

vi suffix_array(string s) {
  s += '\0';
  vi v = sort_cyclic_shifts(s);
  v.erase(v.begin());
  return v;
}
```

4.5. Levenshtein Distance $\mathcal{O}(n^2)$. Minimal number of insertions, removals and edits required to transform one string in the other. Hash: 1b7ea8

```
int levDist(const string &w1, const string &w2) {
  int n1 = sz(w1)+1, n2 = sz(w2)+1;
  vvi dp(n1, vi(n2));
  REP(i, n1) dp[i][0] = i; // removal
  REP(j, n2) dp[0][j] = j; // insertion
  rep(i,1,n1) rep(j,1,n2)
    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[sz(w1)][sz(w2)];
}
```

4.6. Knuth-Morris-Pratt algorithm $\mathcal{O}(N + M)$. Finds all occurrences of a word in a longer string. Hash: a9684c

```
int kmp(const string &word, const string &text) {
  int n = sz(word);
  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 < sz(text); ) {
    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! Matches all words in a dictionary simultaneously. Hash: ec6c33

```
const int sigma = 26;
const char base = 'a';
vi pnr, ploc, sLink, dLink;
vvi to;
vs P;
void makeNode() {
  pnr.pb(-1); sLink.pb(0);
  dLink.pb(0); to.pb(vi(sigma,0));
}
void makeTrie(vs& p) {
  // STEP 1: MAKE A TREE
  P = p;
  pnr.clear(),sLink.clear(),dLink.clear();
    to.clear(),ploc.clear();
  makeNode();
  for (int i = 0; i < sz(p); i++) {
    int cur = 0;
    for (char c : p[i]) {
      int i = c - base;
      if (to[cur][i] == 0) {
        makeNode();
        to[cur][i] = sz(to) - 1;
      }
      cur = to[cur][i];
    }
    pnr[cur] = i; ploc.pb(cur);
  }
  // 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];
    }
  }
}
void traverse(string& s) {
  for (int cur = 0, i = 0, n = sz(s); i < n; i++) {
    cur = to[cur][s[i] - base];
    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. Allows for fast access to all palindromes contained in a string. They can be used to solve the longest palindromic substring, the k-factorization problem[2] (can a given string be divided into exactly k palindromes), palindromic length of a string[3] (what is the minimum number of palindromes needed to construct the string), and finding and counting all distinct sub-palindromes. Hash: 555dcd

```
const int sigma = 26;
const char base = 'a';
struct state {
  int len, link, to[sigma];
};
struct eertree {
  int last, size, n;
  vector<state> nodes;
  string s;
  eertree() : last(1), size(2), n(0) {
    nodes.pb({-1,-1});
    nodes.pb({0,0}); }
  void extend(char c) {
    s.pb(c); n++; int p = last;
    while (n - nodes[p].len - 2 < 0 || c != s[n -
    ↪  nodes[p].len - 2])
      p = nodes[p].link;
    if (!nodes[p].to[c-base]) {
      int q = last = size++;
      nodes.pb({nodes[p].len + 2, 1});
      nodes[p].to[c-base] = q;
      do { p = nodes[p].link;
      } while (p != -1 && (n < nodes[p].len + 2 ||
            c != s[n - nodes[p].len - 2]));
      if(p != -1) nodes[q].link =
      ↪  nodes[p].to[c-base]; }
    else
      last = nodes[p].to[c-base];
  }};
```

4.9. Suffix Tree. Compressed suffix trie with $\leq 2n$ vertices. Works with characters in ASCII range $[64, 128)$. Preprocesses for fast substring queries. Also used to find longest substring that is prefix (return index in loop). Hash: 377cbd

```
const char base = 'A';
const int sigma = 26;
struct suffixtree {
  string a;
  vvi t; vi l, r, p, s; // p: parent, s: suffix link
  int tv, tp, ts, la; // edge p[v] -> v, contains
  ↪  a[l[v]..r[v]-1]
  suffixtree(const string& _a) : a(_a) {
    int n = sz(a) * 2; t = vvi(n, vi(sigma, -1));
    t[1] = vi(sigma, 0); r = vi(n, sz(a));
    l = p = s = vi(n); l[0]=l[1]=-1;
    la = tv = tp = r[0] = r[1] = 0; s[0] = 1; ts =
    ↪  2;
    for (; la < sz(a); la++) ukkadd(a[la] - base);
  }

  void ukkadd(int c) {
    if (r[tv] <= tp) {
      if (t[tv][c] == -1) {
        t[tv][c]=ts; l[ts]=la; p[ts++]=tv;
        tv=s[tv]; tp=r[tv]; ukkadd(c); return;
      }
      tv=t[tv][c]; tp=l[tv];
    }

    if (tp == -1 || c == a[tp]-base) { tp++; return;
    ↪  }

    l[ts+1]=la; p[ts+1]=ts;
```

```
    l[ts]=l[tv]; r[ts]=tp; p[ts]=p[tv];
    t[ts][c]=ts+1; t[ts][a[tp]-base]=tv;
    l[tv]=tp; p[tv]=ts; t[p[ts]][a[l[ts]]-base]=ts;
    tv=s[p[ts]]; tp=l[ts];
    while (tp < r[ts])
      tv = t[tv][a[tp]-base], tp += r[tv] - l[tv];
    if (tp == r[ts]) s[ts]=tv;
    else s[ts]=ts+2;
    tp = r[tv] - (tp - r[ts]); ts += 2; ukkadd(c);
  }
  ll max_substr(const string &S) { // O(|S|)
    int v = 0, it = 0, n = sz(S);
    while (it < n) {
      int c = S[it++] - base;
      if ((v = t[v][c]) < 0) return it - 1;
      for (int i = l[v]; it < n && ++i < r[v]; )
        if (S[it++] != a[i]) return it - 1;
    }
    return n;
  }
};
```

4.10. **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. Hash: 9f7b1d

```
typedef vector<char> vc;
struct suffix_automaton {
  vi len, link;
  vi first, topo;//reversed topological ordering
  vector<map<char,int>> next;
  vi previd; vc prevc;
  vvi linkinv;
  int sz, last;
  string s;
  suffix_automaton() : len(1), link(1),
  next(1), first(1), previd(1), prevc(1) {
    sz = 1; last = 0; }
  suffix_automaton(string s) : suffix_automaton() {
    for(char c : s) extend(c);
    maketopo(); makelinkinvs();
  }
  void extend(char c){ s.pb(c);
    int cur = sz++; len.pb(len[last]+1); link.pb(0);
    next.pb(map<char,int>()); first.pb(sz(s));
    previd.pb(last); prevc.pb(c);
    int p = last;
    for(; p != -1 && !next[p].count(c); p = link[p])
      next[p][c] = cur;
    if(p != -1) { int q = next[p][c];
      if(len[p] + 1 == len[q]){ link[cur] = q; }
      else { int clone = sz++;
        len.pb(len[p] + 1); first.pb(first[q]);
        link.pb(link[q]); next.pb(next[q]);
        previd.pb(p); prevc.pb(c);
        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 makelinkinvs() {
    linkinv = vvi(sz);
    rep(i,1,sz) linkinv[link[i]].pb(i); }
  void maketopo() {
    topo.clear();
    topo = vi(sz); REP(i,sz) topo[i] = i;
    sort(all(topo), [&](ll a, ll b)
      { return len[a] > len[b]; });
  }
  int locstr(string& other){//returns location of
  ↪  other (or -1)
    int cur = 0;
    for(int i = 0; i < sz(other); ++i){
      if(cur == -1) return -1;
      cur = next[cur][other[i]]; }
    return cur; }
  string maxstring(int loc) {
    string res;
    while(loc > 0) {
      res.pb(prevc[loc]); loc = previd[loc]; }
    reverse(all(res)); return res;
  }
};
```

DP examples Hash: 7f3fba

```
//cnt[sa.locstr(s)] = #distinct substrings of sa
↪  with prefix s
vi distinct(suffix_automaton& sa) {
    vi cnt = vi(sa.sz, 1); cnt[0] = 0;
    for(ll i : sa.topo)
        for(auto p : sa.next[i])
            cnt[i] += cnt[p.y];
    return cnt;
}

//cnt[sa.locstr(s)] = #locations of s in sa
vi occur(suffix_automaton& sa) {
    vi cnt = vi(sa.sz, 0);
    for(int cur = 0, i = 0; i < sz(sa.s); i++) {
        cnt[cur = sa.next[cur][sa.s[i]]]++; }
    for(ll i : sa.topo) cnt[sa.link[i]] += cnt[i];
    return cnt;
}

//return endpositions of occurrences of t
↪  (unsorted!)
vi location(suffix_automaton& sa, string& t) {
    int cur = sa.locstr(t);
    if(cur == -1) return vi();
    vi res, stack(1,cur);
    while(sz(stack) > 0) {
        cur = stack.back(), stack.pop_back();
        res.pb(sa.first[cur]);
        for(ll n : sa.linkinv[cur]) stack.pb(n); }
    return res; }

//find the longest common substring
string lcs(vs& s) {
    //Make the automaton
    vc extra;
    REP(i,sz(s))
      extra.pb(i + 256);//assert not in s!
    suffix_automaton sa;
    REP(i,sz(s)) {
        for(char c : s[i]) sa.extend(c);
        sa.extend(extra[i]);
    }
    sa.maketopo();
    sa.makelinkinvs();

    //Determine possible locations
    vvb pos; int cur;
    REP(i, sz(s)) {
        pos.pb(vb(sz(s[i]), false));
        vi stack; cur = sa.next[0][extra[i]];
        for(char c : s[i]) {
            cur = sa.next[cur][c]; stack.pb(cur);
        }
        while(sz(stack) > 0) {
            cur = stack.back(); stack.pop_back();
            if(!pos[i][cur]) {
                pos[i][cur] = true;
                for(ll p : sa.linkinv[cur])
                ↪  stack.pb(p);
            }}}

    //Determine the answer
    for(ll i : sa.topo) {
        bool can = true;
        REP(j, sz(s)) if(!pos[j][i]) {
            can = false;
            break;}
        if(can)
            return sa.maxstring(i);//sa.length[i]
    } return ""; }
```

4.11. **Hashing.** Modulus should be a large prime. Can also use multiple instances with different moduli to minimize chance of collision. Hash: 71ce96

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

```
  }
};
```

5. Geometry

Hash: 8169c8

```
const ld EPS = 1e-7, PI = acos(-1.0);
typedef ld NUM; // EITHER ld OR ll
typedef pair<NUM, NUM> pt;
typedef vector<pt> poly;

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

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

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; }

// square distance from p to q
NUM dist2(pt p, pt q){
  return (q - p) * (q - p);
 }

//Normal distance from p to q
ld dist(pt p, pt q) { return sqrt(dist2(p,q)); }

// distance from p to line ab
ld distPtLine(pt p, pt a, pt b) {
  p -= a; b -= a;
  return sqrt(ld(p^b) * (p^b) / (b*b));
}

// distance from p to linesegment ab
ld distPtSegment(pt p, pt a, pt b) {
  p -= a; b -= a;
  NUM dot = p*b, len = b*b;
  if (dot <= 0) return sqrt(p*p);
  if (dot >= len) return sqrt((p-b)*(p-b));
  return sqrt(p*p - ld(dot)*dot/len);
}

// projects p onto the line ab
// NUM has to be ld
pt proj(pt p, pt a, pt b) {
  p -= a; b -= a;
  return a + b*((b*p) / (b*b));
}

bool col(pt a, pt b, pt c) {
  return abs((a-b) ^ (a-c)) < EPS;
}

// note: to accept collinear points, change `> 0'
// returns true if r is on the left side of line pq
bool ccw(pt p, pt q, pt r) {
  return ((q - p) ^ (r - p)) > 0; }

// true => 1 intersection, false => parallel or same
bool linesIntersect(pt a, pt b, pt c, pt d) {
  return abs((a-b) ^ (c-d)) > EPS;
}

// Test if p is on line segment ab
bool segmentHasPoint(pt p, pt a, pt b) {
  pt u = p-a, v = p-b;
  return abs(u^v) < EPS && u*v <= 0;
}

// Check lines intersect!
// NUM has to be ld
pt lineLineIntersection(pt a, pt b, pt c, pt d) {
  ld det = (a-b) ^ (c-d);
  return ((c-d)*(a^b) - (a-b)*(c^d)) * (1.0/det);
}

// Check lines intersect!
// Num has to be ld
bool segmentIntersection(pt a, pt b, pt c, pt d, pt&
↪ res) {
  res = lineLineIntersection(a, b, c, d);
  return segmentHasPoint(res,a,b) &&
  ↪ segmentHasPoint(res, c, d);
}

// Lines can be parallel: segments overlap from
↪ seg(start, end)
// 0 = no intersect, 1 = 1 lines not parallel (res =
↪ start), 2 = lines parallel
// Num has to be ld
int segmentIntersection(pt a, pt b, pt c, pt d, pt&
↪ start, pt& end) {
  pt d1 = b - a, d2 = d - c;
  if(abs(d1 ^ d2) > EPS) return
  ↪ segmentIntersection(a, b, c, d, start);
  if(abs((c - a) ^ d1) > EPS) return 0;
  if(d1 * d2 < 0) swap(c,d), d2 = d - c;
  ld s = max((ld)0, (c - a) * d1), e = min(d1 * d1,
  ↪ (d - a) * d1);
  s /= (d1 * d1), e /= (d1 * d1);
  if(s > e + EPS) return 0;
  start = a + d1 * s; end = a + d1 * e; return 2;
}

// line segment p-q intersect with line A-B.
// NUM has to be ld!
bool lineIntersectSeg(pt p, pt q, pt a, pt b, pt&
↪ res) {
  res = lineLineIntersection(p,q,a,b);
  return segmentHasPoint(res,p,q);
}
```

5.1. Convex Hull $\mathcal{O}(n \log n)$. Hash: 706f63

```
// the convex hull consists of: { pts[ret[0]],
↪ pts[ret[1]], ... pts[ret.back()] } in
↪ counterclockwise order
vi convexHull(const poly &pts) {
  if (pts.empty()) return vi();
  vi ret, ord;
  int n = sz(pts), st = min_element(all(pts)) -
  ↪ pts.begin();
  REP(i, 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 : p * p > q *
    ↪ q;
  });
  ord.pb(st); ret.pb(st);
  for (int i : ord) {
    // use '>=' in ccw to include ALL points on the
    ↪ hull-line
    for(int s = sz(ret) - 1; s > 1 && !ccw(pts[ret[s
    ↪ - 1]], pts[ret[s]], pts[i]); s--)
      ret.pop_back();
    ret.pb(i);
  }
  ret.pop_back();
  return ret;
}
```

5.2. Closest points $\mathcal{O}(n \log n)$. Hash: 7ab1d9

```
poly pts;

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

inline NUM dist(ii p) { pt a = (pts[p.x] -
↪ pts[p.y]); return a * a; }

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

// closest pts (by index) inside pts[l ... r - 1],
↪ with sorted y values in ys
// check pts is sorted on x!
ii closest(int l, int r, vi &ys) {
  if (r - l == 2) { // don't assume 1 here.
    ys = { l, l + 1 };
    sort(all(ys), byY());
    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 = (pts[m-1].x +
  ↪  pts[m].x) / 2;
  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 (sz(q) > 8) q.pop_front(); // magic from
    ↪  Introduction to Algorithms.
  }
  return delta;
}
```

5.3. Great-Circle Distance.

Computes the distance between two points (given as latitude/longitude coordinates) on a sphere of radius r. Hash: d850b6

```
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)); }
```

5.4. Delaunay triangulation.

Hash: 742901

```
int sgn(const ll& a) { return (a > 0) - (a < 0);}

const pt inf_pt = make_pair(1e18, 1e18);

struct Quad { // `QuadEdge` originally
  pt O; // origin
  Quad *rot = nullptr, *onext = nullptr;
  bool used = false;
  Quad* rev() const { return rot->rot; }
  Quad* lnext() const {
    return rot->rev()->onext->rot; }
  Quad* oprev() const {
    return rot->onext->rot; }
  pt dest() const { return rev()->O; }
};

Quad* make_edge(pt from, pt to) {
  Quad* e1 = new Quad, *e2 = new Quad;
  Quad* e3 = new Quad, *e4 = new Quad;
  e1->O = from; e2->O = to;
  e3->O = e4->O = inf_pt;
  e1->rot = e3; e2->rot = e4;
  e3->rot = e2; e4->rot = e1;
  e1->onext = e1; e2->onext = e2;
  e3->onext = e4; e4->onext = e3;
  return e1;
}
void splice(Quad* a, Quad* b) {
  swap(a->onext->rot->onext, b->onext->rot->onext);
  swap(a->onext, b->onext);
}

void delete_edge(Quad* e) {
  splice(e, e->oprev());
  splice(e->rev(), e->rev()->oprev());
}

Quad* connect(Quad* a, Quad* b) {
  Quad* e = make_edge(a->dest(), b->O);
  splice(e, a->lnext());
  splice(e->rev(), b);
  return e;
}

bool left_of(pt p, Quad* e) {
  return ((e->O - p) ^ (e->dest() - p)) > 0; }
bool right_of(pt p, Quad* e) {
  return ((e->O - p) ^ (e->dest() - p)) < 0; }

template <class T> T det3(T a1, T a2, T a3,
    T b1, T b2, T b3, T c1, T c2, T c3) {
  return a1*(b2*c3 - c2*b3) - a2*(b1*c3 - c1*b3)
    + a3*(b1*c2 - c1*b2);
}

// Calculate directly with __int128, or with angles
bool in_circle(pt a, pt b, pt c, pt d) {
  __int128 det = 0;
  det -= det3<__int128>(b.x,b.y,b * b,
    c.x,c.y,c * c, d.x,d.y,d * d);
  det += det3<__int128>(a.x,a.y,a * a,
    c.x,c.y,c * c, d.x,d.y,d * d);
  det -= det3<__int128>(a.x,a.y,a * a,
    b.x,b.y,b * b, d.x,d.y,d * d);
  det += det3<__int128>(a.x,a.y,a * a,
    b.x,b.y,b * b, c.x,c.y,c * c);
  return det > 0;
}

pair<Quad*, Quad*> build_tr(int k, int r,
    poly& p) {
  if (r - k == 3) {
    Quad* res = make_edge(p[k], p[r]);
    return make_pair(res, res->rev());
  }
  if (r - k == 4) {
    Quad *a = make_edge(p[k], p[k+1]);
    Quad *b = make_edge(p[k+1], p[r]);
    splice(a->rev(), b);
    int sg = sgn((p[k + 1] - p[k]) ^(p[r] - p[k]));
    if (sg == 0) return make_pair(a, b->rev());
    Quad* c = connect(b, a);
    if (sg == 1) return make_pair(a, b->rev());
    return make_pair(c->rev(), c);
  }
  int mid = (k + r) / 2;
  Quad *ldo, *ldi, *rdo, *rdi;
  tie(ldo, ldi) = build_tr(k, mid, p);
  tie(rdi, rdo) = build_tr(mid + 1, r, p);
  while (true) {
    if (left_of(rdi->O, ldi)) {
      ldi = ldi->lnext(); continue; }
    if (right_of(ldi->O, rdi)) {
      rdi = rdi->rev()->onext; continue; }
    break;
  }
  Quad* B = connect(rdi->rev(), ldi);
  auto valid = [&B](Quad* e) {
    return right_of(e->dest(), B); };

  if (ldi->O == ldo->O) ldo = B->rev();
  if (rdi->O == rdo->O) rdo = B;
  while (true) {
    Quad* lc = B->rev()->onext; // left candidate
    if (valid(lc)) {
      while (in_circle(B->dest(), B->O,
          lc->dest(), lc->onext->dest())) {
        Quad* t = lc->onext;
        delete_edge(lc);
        lc = t;
      }
    }
    Quad* rc = B->oprev(); // right candidate
    if (valid(rc)) {
      while (in_circle(B->dest(), B->O,
          rc->dest(), rc->oprev()->dest())) {
        Quad* t = rc->oprev();
        delete_edge(rc);
        rc = t;
      }
    }
    if (!valid(lc) && !valid(rc)) break;
    if (!valid(lc) || (valid(rc) && in_circle(
        lc->dest(), lc->O, rc->O, rc->dest())))
      B = connect(rc, B->rev());
    else B = connect(B->rev(), lc->rev());
  }
  return make_pair(ldo, rdo);
}

vector<tuple<pt, pt, pt>> delaunay(poly p) {
  sort(all(p), [](const pt& a, const pt& b) {
    return a.x < b.x ||
        (a.x == b.x && a.y < b.y);
  });
  auto res = build_tr(0, sz(p) - 1, p);
  Quad* e = res.first;
  vector<Quad*> edges = {e};
  while(((e->dest() - e->onext->dest()) ^(e->O -
  ↪  e->onext->dest())) < 0)
    e = e->onext;
```

```
  auto add = [&p, &e, &edges]() {
    Quad* cur = e;
    do {
      cur->used = true;
      p.pb(cur->O);
      edges.pb(cur->rev());
      cur = cur->lnext();
    } while (cur != e);
  };
  add(); p.clear();

  int kek = 0;
  while (kek < sz(edges))
    if (!(e = edges[kek++])->used) add();
  vector<tuple<pt, pt, pt>> ans;
  for (int i = 0; i < sz(p); i += 3)
    ans.pb(make_tuple(p[i], p[i + 1], p[i + 2]));
  return ans;
}
```

5.5. 3D Primitives. Hash: 84ea9e

```
#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. Hash: 3e3b82

```
struct RMST {
  struct point {
    int i; ll x, y;
    point() : i(-1) { }
    point(ll _x, ll _y, int _i) : x(_x), y(_y),
    ↪  i(_i){}
    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;
  }};
  vector<point> A, best, tmp;
  int n;
  RMST() : n(0) {}
  void add_point(int x, int y) {
    A.pb(point(x,y,n++));}
  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 && A[i].d1() < A[j].d1())){
        tmp[k] = A[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] = A[j++];
        if (bst.i == -1 || bst.d2() < tmp[k].d2())
          bst = tmp[k]; } }
```

```
    rep(i,l,r+1) A[i] = tmp[i]; }
  vector<pair<ll,ii> > candidates() {
    vector<pair<ll, ii> > es;
    tmp = best = vector<point>(n);
    REP(p, 2) {
      REP(q, 2) {
        sort(all(A));
        REP(i, n) best[i].i = -1;
        rec(0, n-1);
        REP(i, n) {
          if(best[A[i].i].i != -1)
            es.eb(A[i].dist(best[A[i].i]),
                  ii(A[i].i, best[A[i].i].i));
          swap(A[i].x, A[i].y);
          A[i].x *= -1, A[i].y *= -1; } }
      REP(i, n) A[i].x *= -1; }
    return es; } };
```

5.8. **Points and lines (CP3).** Hash: a7aab7

```
ld DEG_to_RAD(ld d) { return d*PI/180.0; }
ld RAD_to_DEG(ld r) { return r*180.0/PI; }

// rotate p by rad RADIANS CCW w.r.t origin (0, 0)
// NUM has to be ld
pt rotate(pt p, ld rad) {
  return make_pair(p.x*cos(rad) - p.y*sin(rad),
         p.x*sin(rad) + p.y*cos(rad));
}

// lines are (x,y) s.t. ax + by + c = 0 AND b=0,1.
struct line { ld a, b, c; };

// gives line through p1, p2
line pointsToLine(pt p1, pt p2) {
  if (fabs(p1.x - p2.x) < EPS) // vertical line
    return { 1.0, 0.0, -(ld)p1.x };
  else {
    ld a =-(ld)(p1.y - p2.y) / (p1.x - p2.x);
    return {
      a,
      1.0,
      -(ld)(a * p1.x) - p1.y
    };
  }
}

// returns the reflection of p on the line through a
↪ and b
//NUM has to be ld
pt reflectionPoint(pt p, pt a, pt b) {
  pt m = proj(p, a, b);
  return m * 2 - p; }

// returns angle aob in rad in [0, 2 PI)
ld angle(pt a, pt o, pt b) {
  pt oa = a - o, ob = b - o;
  ld antw = atan2(ob.y, ob.x) - atan2(oa.y, oa.x);
  if(antw < 0)
    antw += 2 * PI;
  return antw;
}
```

5.9. **Polygon (CP3).** Polygons have $P_0 = P_{n-1}$ here. Hash: 4b9bcb

```
// returns the perimeter: sum of Euclidean distances
// of consecutive line segments (polygon edges)
ld perimeter(const poly &P) {
  ld result = 0.0;
  REP(i, sz(P)-1)
    result += dist(P[i], P[i+1]);
  return result; }

// Returns TWICE the area of a polygon (for
↪ integers)
NUM polygonTwiceArea(const poly &P) {
  NUM area = 0;
  REP(i,sz(P) - 1)
    area += P[i] ^ P[i + 1];
  return abs(area); // area < 0 <=> p ccw
}

// returns true if we always make the same turn
// throughout the polygon (strict)
bool isConvex(const poly &P) {
  int n = sz(P);
  if (n <= 3) return false; // point=2; line=3
  bool isLeft = ccw(P[0], P[1], P[2]);
  REP(i, n-2) if (ccw(P[i], P[i+1],
      P[(i+2) == n ? 1 : i+2]) != isLeft ||
      col(P[i], P[i+1], P[(i+2) == n ? 1 : i+2]))
    return false; // different sign -> concave
  return true; } // convex


bool insidePolygon(const poly &P, pt p, bool strict
↪ = true) {
  int n = 0;
  REP(i, sz(P) - 1) {
    // if p is on edge of polygon
    if (segmentHasPoint(p, P[i], P[i + 1])) return
    ↪ !strict;
    // or: if(distPtSegmentSq(p, pts[i], pts[i + 1])
    ↪ <= EPS) return !strict;

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

// cuts polygon Q along the line formed by a -> b
// (note: Q[0] == Q[n-1] is assumed)
// NUM has to be ld
poly cutPolygon(pt a, pt b, const poly &Q) {
  poly P;
  REP(i, sz(Q)) {
    ld left1 = (b - a) ^ (Q[i] - a);
    ld left2 = 0;
    if (i != sz(Q)-1)
      left2 = (b - a) ^ (Q[i+1] - a);
    if (left1 > -EPS)
      P.pb(Q[i]); // Q[i] is left of ab
    if (left1 * left2 < -EPS)
      // edge Q[i]--Q[i+1] crosses line ab
        P.pb(pt()); lineIntersectSeg(Q[i], Q[i+1],
        ↪ a, b, P.back());
  }
  if (!P.empty() && !(P.back() == P.front()))
    P.pb(P.front()); // make P[0] == P[n-1]
  return P; }
```

5.10. **Triangle (CP3).** Hash: afa849

```
ld perimeter(pt a, pt b, pt c) {
  return dist(a, b) + dist(b, c) + dist(c, a); }

ld area(ld ab, ld bc, ld ca) {
 // Heron's formula
  ld s = 0.5 * (ab+bc+ca);
  return sqrt(s)*sqrt(s-ab)*sqrt(s-bc)*sqrt(s-ca);
}
ld area(pt a, pt b, pt c) {
  return area(dist(a, b), dist(b, c), dist(c, a));
}
ld rInCircle(ld ab, ld bc, ld ca) {
  return area(ab,bc,ca)*2.0 / (ab+bc+ca);
}

ld rInCircle(pt a, pt b, pt c) {
  return rInCircle(dist(a,b),dist(b,c),dist(c,a));
}
// assumption: the required points/lines functions
// have been written.
// Returns if there is an inCircle center
// if it returns TRUE, ctr will be the inCircle
// center and r is the same as rInCircle
bool inCircle(point p1, point p2, point p3, point
↪ &ctr, ld &r) {
  r = rInCircle(p1, p2, p3);
  if (fabs(r) < EPS) return false;

  ld ratio = dist(p1, p2) / dist(p1, p3);
  pt q1 = p2 + (p3 - p2) * (ratio / (1 + ratio));

  ratio = dist(p2, p1) / dist(p2, p3);
  pt q2 = p1 + (p3 - p1) * (ratio / (1 + ratio));

  // get their intersection point:
```

```
  ctr = lineLineIntersection(p1, q1, p2, q2);
  return true;
}

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

ld rCircumCircle(pt a, pt b, pt c) {
  return rCircumCircle(
      dist(a,b), dist(b,c), dist(c,a));
}

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

  ctr.x = (d*e - b*f) / g;
  ctr.y = (a*f - c*e) / g;
  r = dist(p1, ctr); // r = dist(center, p_i)
  return true;
}

// returns if pt d is strictly inside the
// circumCircle defined by a,b,c
// for non strict, change < to <=
bool inCircumCircle(pt a, pt b,  pt c, pt d) {
  pt va=(d - a), vb=(d - b), vc=(d - c);
  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(NUM a, NUM b, NUM c) {
  return a+b > c && a+c > b && b+c > a; }
```

5.11. Circle (CP3). Hash: e07ae5

```
//0 is in circle, 1 on the border and 2 outside the
↪  circle
int insideCircle(pt p, pt c, NUM r) {
    NUM d = dist2(p,c), r2 = r * r;
    return d < r2 ? 0 : d == r2 ? 1 : 2; }
    ↪  //inside/border/outside
//c becomes center of the circle through p1 and p2
↪  with radius r
//For other option, reverse p1 and p2
//Requires NUM = ld
bool circle2PtsRad(pt p1, pt p2, NUM r, pt &c) {
    ld d2 = dist2(p1,p2);
    ld det = r * r / d2 - 0.25;
    if (det < 0.0) return false;
    ld 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; }
```

5.12. 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 $(c_1r_2 + c_2r_1)/(r_1+r_2)$, external intersect at $(c_1r_2 - c_2r_1)/(r_1+r_2)$.

6. Miscellaneous

6.1. Fast Fourier Transform $\mathcal{O}(n \log n)$.

Given two polynomials $A(x) = a_0 + a_1x + \cdots + a_{n/2}x^{n/2}$ and $B(x) = b_0 + b_1x + \cdots + b_{n/2}x^{n/2}$, FFT calculates all coefficients of $C(x) = A(x) \cdot B(x) = c_0 + c_1x + \ldots c_nx^n$, with $c_i = \sum_{j=0}^{i} a_j b_{i-j}$. Hash: 34d39d

```
typedef complex<ld> cpx;
typedef vector<cpx> vc;
vi rev; vc rt;
void fft(vc& A) {
  REP(i, sz(A)) if (i < rev[i]) swap(A[i],
  ↪  A[rev[i]]);
  for (int k = 1; k < sz(A); k *= 2)
    for (int i = 0; i < sz(A); 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(vc& a, vc& b) { // a = a * b
  ll logn = 0; for(; (1 << logn) < sz(a); logn++);
  ll n = (1 << logn); a.rs(n); b.rs(n);
    const ld PI = acos(-1.0);
  rev = vi(n); rt = vc(n);
  rev[0] = 0; rt[1] = cpx(1, 0);
  REP(i, n) rev[i] = (rev[i/2] | (i&1)<<logn)/2;
  for (int k = 2; k < n; 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, n) a[i] *= b[i] / (ld)(n);
  REP(i, n) rt[i] = 1 / rt[i]; fft(a); }
```

NTT $\mathcal{O}(n\log(n))$. Requires $2^{e_2(mod-1)} \geq n$. Can be calculated exact by taking (multiple) large primes as modulus (and combining them with CRT). Hash: 70a119

```
//include mod_pow
const ll mod = 998244353, g = 2;//g is primitive
↪  root of mod

vi rev, rt;
ll inv(ll x) { return mod_pow(x, mod - 2, mod);}
void ntt(vi &A) {
  REP(i, sz(A)) if (i < rev[i]) swap(A[i],
  ↪  A[rev[i]]);
  for (int k = 1; k < sz(A); k *= 2)
    for (int i = 0; i < sz(A); i += 2*k) REP(j, k) {
      ll t = rt[j + k] * A[i + j + k];
      A[i + j + k] = (A[i + j] - t + mod * mod) %
      ↪  mod;
      A[i + j] = (A[i + j] + t) % mod;
    }
}
void multiply(vi &A, vi &B) { //A = A * B
  ll logn; for(logn = 1; (1 << logn) < sz(A);
  ↪  logn++);
  ll n = 1 << logn; rev = rt = vi(n);
  A.rs(n), B.rs(n);
  rev[0] = 0; rt[1] = 1;
  REP(i, n) rev[i] = (rev[i/2] | (i&1)<<logn)/2;
  for (int i = 1; (1 << i) < n; ++i) {
    int k = 1<<i;
    ll z = mod_pow(g,(mod - 1) >> (i + 2),mod);
    rep(i, k/2, k) rt[2*i] = rt[i], rt[2*i+1] =
    ↪  (rt[i] * z) % mod;
  }
  ntt(A); ntt(B);
  REP(i, n) A[i] = (((A[i] * B[i]) % mod) * inv(n))
  ↪  % mod;
  REP(i,n) rt[i] = inv(rt[i]);
  ntt(A); }
```

6.2. Minimum Assignment (Hungarian Algorithm) $\mathcal{O}(n^3)$. Hash: 1c5dea

```
//a[i][j] gives value for 1 <= i <= n and 1 <= j <=
↪  m (1-based!)
//pm[j] gives assigned i and pn[i] assigned j
ll minimum_assignment(int n, int m, vvi& a, vi& pm,
↪  vi& pn) {
  vi u(n + 1), v(m + 1), way(m + 1); pm = vi(m + 1);
  for (int i = 1; i <= n; i++) {
    pm[0] = i;
    int j0 = 0;
    vi mv(m + 1, INT_MAX);
    vb used(m + 1, false);
    do {
      used[j0] = true;
      int i0 = pm[j0], delta = INT_MAX, j1;
      for (int j = 1; j <= m; j++)
        if (!used[j]) {
          int cur = a[i0][j] - u[i0] - v[j];
          if (cur < mv[j]) mv[j] = cur, way[j] = j0;
          if (mv[j] < delta) delta = mv[j], j1 = j;
        }
      for (int j = 0; j <= m; j++) {
        if(used[j]) u[pm[j]] += delta, v[j] -=
        ↪  delta;
        else mv[j] -= delta;
      }
      j0 = j1;
    } while (pm[j0] != 0);
    do {
      int j1 = way[j0]; pm[j0] = pm[j1]; j0 = j1;
    } while (j0);
  }
  pn = vi(n + 1); rep(i, 1, m + 1) pn[pm[i]] = i;
  return -v[0];
}
```

6.3. Partial linear equation solver $\mathcal{O}(N^3)$. Hash: 79998e

```
typedef ld NUM;
const NUM EPS = 1e-5;
bool is0(NUM a) { return -EPS < a && a < EPS; }
// finds x such that Ax = b
// A_ij is M[i][j], b_i is M[i][m]
// 0 is no solution, 1 is unique, 2 is multiple
// peq is index of pivot equation
int solveM(int n, int m, vector<vector<NUM>> &M,
↪  vector<NUM> &val, vi &peq) {
  int pr = 0, pc = 0;
  while (pc < m) {
    //Pick first nonzero element (ld largest
    ↪  stabler)
    int r = pr, c;
    while (r < n && is0(M[r][pc])) r++;
    if (r == n) { pc++; continue; }
    for (c = 0; c <= m; c++)
      swap(M[pr][c], M[r][c]);
    r = pr++; c = pc++;
    NUM div = 1 / M[r][c]; //mult inv if mod
    for (int col = c; col <= m; col++)
      M[r][col] *= div;
    REP(row, n) {
      if (row == r) continue;
      // F2: if (M[row].test(c)) M[row] ^= M[r];
      NUM times = -M[row][c];
      for (int col = c; col <= m; col++)
        M[row][col] += times * M[r][col];
    }
  } // now M is in RREF

  for (int r = pr; r < n; r++)
    if (!is0(M[r][m])) return 0;
  peq = vi(m, -1);
  val = vector<NUM>(m, 0);
  for (int col = 0, row = 0; col < m && row < n;
  ↪  col++)
    if(!is0(M[row][col])) {
      peq[col] = row;
      val[col] = M[row][m];
      row++;
    }
  REP(i, m) if (peq[i] == -1) return 2;
  return 1; }
```

6.4. Cycle-Finding. Hash: a1f2e7

```
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); }
```

6.5. Longest Increasing Subsequence. Hash: 1bc3da

```
vi lis(vi arr) {
  vi seq, back(sz(arr)), ans;
  REP(i, sz(arr)) {
    int res = 0, lo = 1, hi = sz(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 < sz(seq)) seq[res] = i;
    else seq.pb(i);
    back[i] = res == 0 ? -1 : seq[res-1];
  }
  int at = seq.back();
  while (at != -1) ans.pb(at), at = back[at];
  reverse(all(ans));
  return ans;
}
```

6.6. Dates. Hash: 28e80c

```
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; }
```

6.7. Simplex. Hash: 7dcfea

```
const ld EPS = 1e-9;
struct LPSolver {
 int m, n; vi B, N; vvd D;
 LPSolver(const vvd &A, const vd &b, const vd &c) :
     m(sz(b)), n(sz(c)),
     N(n + 1), B(m), D(m + 2, vd(n + 2)) {
  REP(i, m) REP(j, n) D[i][j] = A[i][j];
  REP(i, m) { B[i] = n + i; D[i][n] = -1;
    D[i][n + 1] = b[i]; }
  REP(j, n) N[j] = j, D[m][j] = -c[j];
  N[n] = -1; D[m + 1][n] = 1;
 }
 void Pivot(int r, int s) {
  ld inv = 1.0 / D[r][s];
  REP(i, m+2) if (i != r) REP(j, n+2) if (j != s)
    D[i][j] -= D[r][j] * D[i][s] * inv;
  REP(j, n+2) if (j != s) D[r][j] *= inv;
  REP(i, m+2) 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;
   REP(i, m) {
    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); } }
 ld Solve(vd &x) {
  int r = 0;
  rep(i, 1, m) if (D[i][n+1] < D[r][n+1]) r = i;
  if (D[r][n + 1] < -EPS) {
   Pivot(r, n);
   if (!Simplex(1) || D[m + 1][n + 1] < -EPS)
     return -numeric_limits<ld>::infinity();
   REP(i, m) 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<ld>::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]; } };
// 2-phase simplex solves linear system:
//     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 -- optimal solution (by reference)
// OUTPUT: c^T x (inf. if unbounded above, nan if
↪ infeasible)
// *** Example ***
// const int m = 4, n = 3;
// ld _A[m][n] = {{6,-1,0}, {-1,-5,0},
//     {1,5,1}, {-1,-5,-1}};
// ld _b[m] = {10,-4,5,-5}, _c[n]= {1,-1,0};
// vvd A(m);
// vd b(_b, _b + m), c(_c, _c + n), x;
// REP(i, m) A[i] = vd(_A[i], _A[i] + n);
// LPSolver solver(A, b, c);
// ld value = solver.Solve(x);
// cerr << "VALUE: " << value << endl; // 1.29032
// cerr << "SOLUTION:"; // 1.74194 0.451613 1
// REP(i, sz(x)) cerr << " " << x[i];
// cerr << endl;
```

7. 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$^{\text{th}}$ 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$^{\text{nd}}$ 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$^{\text{nd}}$ order (nr perms of $1,1,2,2,\dots,n,n$ with exactly k ascents):
  $\left\langle\!\left\langle{n \atop k}\right\rangle\!\right\rangle = (k+1)\left\langle\!\left\langle{n-1 \atop k}\right\rangle\!\right\rangle + (2n-k-1)\left\langle\!\left\langle{n-1 \atop k-1}\right\rangle\!\right\rangle$.
- 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: $b_n = \sum_{k=0}^{n}\binom{n}{k}a_k \iff a_n = \sum_{k=0}^{n}(-1)^{n-k}\binom{n}{k}b_k$.
- **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	$[n \leq k]$	$[n \leq k]$	$\binom{k}{n}$	$n!\binom{k}{n}$

8. 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 $s = \frac{a+b+c}{2}$.
- **Shoelace formula:** $A = \frac{1}{2}|\sum_{i=0}^{n-1} x_i y_{i+1} - x_{i+1}y_i|$.
- **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)
- **Absorption probabilities** A random walk on $[0,n]$ with probability p to increase and q to decrease, starting at k has at n absorption probability $\frac{(q/p)^k-1}{(q/p)^n-1}$ if $q \neq p$, and k/n if $q = p$.
- A minimum Steiner tree for n vertices requires at most $n-2$ additional Steiner vertices.
- **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. $g(a_1,a_2) = a_1a_2 - a_1 - a_2$, $N(a_1,a_2) = (a_1-1)(a_2-1)/2$. $g(d\cdot a_1, d\cdot a_2, a_3) = d\cdot g(a_1,a_2,a_3) + a_3(d-1)$. 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 $q_{ij} = [i=j]\mathrm{indeg}(i) - \#\{(i,j)\in E\}$.
- **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:
  $$|X/G| = \frac{1}{|G|}\sum_{g\in G}|X^g|.$$
- **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
  $$(x + k\cdot\operatorname{lcm}(a,b)/a,\ y - k\cdot\operatorname{lcm}(a,b)/b)\,,\quad k\in\mathbb{Z}$$

9. 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 (Nim = 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 $\bigoplus_{i=1}^n x_i$.

10. Scheduling Theory

Let p_j be the time task j takes on a machine, d_j the deadline, C_j the time it is completed, $L_j = C_j - d_j$ the lateness, $T_j = \max(L_j, 0)$ the tardiness, $U_j = 1$ iff $T_j > 0$ and else 0.

- One machine, minimise $L_{\max}$: do the tasks in increasing deadline
- One machine, minimise $\sum_j w_j C_j$: do the task increasing in p_j/w_j
- One machine, minimise $\sum_{j=1}^n C_j$ under the condition that all tasks can be done on time:
  (1) Initialise $k = n, \tau = \sum_j p_j, J = [n]$
  (2) Take $i_k \in J$ with $d_{i_k} \geq \tau$ and $p_{i_k} \geq p_\ell$ for $\ell \in J$ with $d_\ell \geq \tau$
  (3) $\tau := \tau - p_{i_k}, k := k-1, J := J - \{i_k\}$. If $k \neq 0$, go to step 2.
  (4) The optimale schedule is $i_1, ..., i_n$.
- One machine, minimise $\sum_j U_j$. Add all tasks in order of increasing deadline; if adding a task makes it contrary with its deadline, remove the processed task with the highest processing time.
- Two machines (all tasks have to be done on both machines, in any order), minimise $C_{\max}$: a greedy algorithm, when a machine is free it picks a task that hasn't been done yet on either machine and has longest processing time on the other machine.
- Two machines (all tasks have to be done first on machine 1, then machine 2), minimise $C_{\max}$. There is an optimal schedule with on both machines the same order of tasks. Take $X = \{j : p_{1j} \leq p_{2j}\}$ and Y the complement. Sort X increasing in p_{1j} and Y decreasing in p_{2j}. Then X, Y is an optimal schedule.
- Two machines (all tasks have to be done first on machine 1, then on 2, or vice versa), minimise $C_{\max}$: let J_{12} be the tasks that have to be done first on machine 1, then on 2 and similar J_{21}. Use the above algorithm to find S_{12}, S_{21} optimal for J_{12}, J_{21}. Then optimal is S_{12}, S_{21} for M1 and S_{21}, S_{12} for M2. (If there are tasks that have to be done on only one machine, do them in the middle.)

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?
- TLE?
  - Replace endl with newline
  - Replace vector with const size array
  - #pragma GCC optimize("O3,unroll-loops") #pragma GCC target("avx2,bmi,bmi2,popcnt")
  - for loop with const upper bound $N \leq 1000 \rightarrow$ #pragma GCC unroll(N)
  - Replace conditional assignment with $a = c \cdot t + !c \cdot f$ / ternary when t/f computationally expensive
  - Reduce modulo usage / replace with with bit operations
  - Still not enough? Try to reduce asymptotic runtime

11.1. Dynamic programming optimizations.

- Convex Hull
  - $\text{dp}[i] = \min_{j<i}\{\text{dp}[j] + b[j] \times a[i]\}$
  - $b[j] \geq b[j+1]$
  - optionally $a[i] \leq a[i+1]$
  - $O(n^2)$ to $O(n)$ (see 2.11).
- Divide & Conquer
  - $\text{dp}[i][j] = \min_{k<j}\{\text{dp}[i-1][k] + C[k][j]\}$
  - $A[i][j] \leq A[i][j+1]$
  - sufficient:

$$C[a][c] + C[b][d] \leq C[a][d] + C[b][c], (a \leq b \leq c \leq d) \quad \text{(QI)}$$

  - $O(kn^2)$ to $O(kn \log n)$
- Knuth
  - $\text{dp}[i][j] = \min_{i<k<j}\{\text{dp}[i][k] + \text{dp}[k][j] + C[i][j]\}$
  - $A[i][j-1] \leq A[i][j] \leq A[i+1][j]$
  - $O(n^3)$ to $O(n^2)$
  - sufficient: QI and $C[b][c] \leq C[a][d]$, $a \leq b \leq c \leq d$

11.2. 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
- 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
  - Simulate in reverse order
- 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.