BRAC University

BRACU_Nilgiri

Team Members

Contest Name

2025-09-18

Contest (1)

cf.sh

3 lines

```
#!/bin/bash
code=$1
g++ ${code}.cpp -o $code -std=c++20 -g -DDeBuG
      -Wall -Wshadow -fsanitize=address,
     undefined && ./$code
```

hash.sh

1 lines

```
cpp -dD -P -fpreprocessed | tr -d '\r\n\t ' |
     md5sum | cut -c-6 || echo "000000"
```

stdc++.h

17496a, 42 lines

```
#include <bits/stdc++.h>
using namespace std;
#define TT template <typename T

TT,typename=void> struct cerrok:false_type {};
TT> struct cerrok <T, void_t<decltype(cerr <<
     declval<T>() )>> : true_type {};

TT> constexpr void p1 (const T &x);
TT, typename V> void p1(const pair<T, V> &x) {
  cerr << "{"; p1(x.first); cerr << ", ";
  p1(x.second); cerr << "}";
}
TT> constexpr void p1 (const T &x) {
  if constexpr (cerrok<T>::value) cerr << x;
  else if constexpr (requires { std::declval<T
       &>().pop(); }) {
    auto tmp = x;
    int f = 0;
    cerr << "{";
    while (!tmp.empty()) {
      cerr << (f++ ? ", " : "");
      if constexpr (requires { tmp.top(); })
           p1(tmp.top());
      else p1(tmp.front());
      tmp.pop();
    }
    cerr << "}";
  }
  else { int f = 0; cerr << '{';
    for (auto &i: x)
      cerr << (f++ ? ", " : ""), p1(i);
    cerr << "}";
} }
void p2() { cerr << "]\n"; }
TT, typename... V> void p2(T t, V... v) {
  p1(t);
  if (sizeof...(v)) cerr << ", ";
  p2(v...);
}
#ifdef LOCAL
#define debug(x...) {cerr <<__func__<<":"<<
     __LINE__<<" [" << #x << "] = ["; p2(x);}
#define dbg(x, len) {cerr << __func__ << ":"
     << __LINE__<< " [" << #x << "] = {";for (
     int i = 0; i < (len); ++i) {if (i) cerr <<
      ", ";p1((x)[i]);}cerr << "}\n";}
#endif
```

template.cpp

"bits/stdc++.h" d01068, 36 lines

```
#ifndef LOCAL
#define dbg(...)
#define debug(...)
#endif

// #include <ext/pb_ds/assoc_container.hpp>
// #include <ext/pb_ds/tree_policy.hpp>
// using namespace __gnu_pbds;

using namespace std;
#define int long long
#define sz(v) (int)(v).size()
#define all(v) begin(v),end(v)
#define rep(i,a,b) for(int i=a;i<(b);++i)
#define cinv(v) for (auto &it:v) cin>>it;
#define coutv(v) for (auto &it:v) cout<< it<<'
     '; cout<<'\n';
#define unique(v) sort(v.begin(), v.end()); v.
     erase(unique(v.begin(), v.end()), v.end())
     ;
mt19937_64 rng(chrono::steady_clock::now().
     time_since_epoch().count());
#define rand(l, r) uniform_int_distribution<ll
     >(l, r)(rng)
using ll = long long;
using pii = pair<int, int>;
using pll = pair<ll, ll>;
using vi = vector<int>;
template<class T> using V = vector<T>;
// template<typename T> using ordered_set =
     tree<T, null_type, less<T>, rb_tree_tag,
     tree_order_statistics_node_update>;

void solve(int tc) {

}

signed main() {
    cin.tie(0)->sync_with_stdio(0);
    int t = 1;
    cin >> t;
    for (int _ = 1; _ <= t; ++_) shelby(_);
}
```

stress.sh

14 lines

```
#!/bin/bash
cf gen > in             # input generator
cf bf < in > exp        # bruteforce
cf code < in > out      # buggy code name

for ((i = 1; ; ++i)) do
   echo $i
   ./gen > in
   ./bf < in > exp
   ./code < in > out    # buggy code name
   diff -w exp out || break
done
# Shows expected first, then user
notify-send "bug found!!!!"
```

cf.bat

5 lines

```
@echo off
setlocal
set prog=%1
g++ %prog%.cpp -o %prog% -DDeBuG -std=c++17 -g
      -Wall -Wshadow && .\%prog%
endlocal
```

Mathematics (2)

2.1 Equations

The extremum of a quadtratic is given by $x = -b/2a$.

Cramer: Given an equation $Ax = b$, the solution to a variable x_i is given by

$$x_i = \frac{\det A_i'}{\det A} \quad \text{[where } A_i' \text{ is } A \text{ with the } i\text{'th column replaced by } b.]$$

Vieta: Let $P(x) = a_n x^n + ... + a_0$, be a polynomial with complex coefficients and degree n, having complex roots $r_n, ..., r_1$. Then for any integer $0 \le k \le n$,

$$\sum_{1 \le i_1 < i_2 < ... < i_k \le n} r_{i_1} r_{i_2} ... r_{i_k} = (-1)^k \frac{a_{n-k}}{a_n}$$

Rational Root Theorem: If $\frac{p}{q}$ is a reduced rational root of a polynomial with **integer coeffs**, then $p \mid a_0$ and $q \mid a_n$

2.2 Ceils and Floors

For $x, y \in \mathbb{R}$, $m, n \in \mathbb{Z}$:

- $\lfloor x \rfloor \le x < \lfloor x \rfloor + 1$; $\lceil x \rceil - 1 < x \le \lceil x \rceil$
- $-\lfloor x \rfloor = \lceil -x \rceil$; $-\lceil x \rceil = \lfloor -x \rfloor$
- $\lfloor x + n \rfloor = \lfloor x \rfloor + n$, $\lceil x + n \rceil = \lceil x \rceil + n$
- $\lfloor x \rfloor = m \Leftrightarrow x - 1 < m \le x < m + 1$
- $\lceil x \rceil = n \Leftrightarrow n - 1 < x \le n < x + 1$
- If $n > 0$, $\lfloor \frac{\lfloor x \rfloor + m}{n} \rfloor = \lfloor \frac{x+m}{n} \rfloor$
- If $n > 0$, $\lceil \frac{\lceil x \rceil + m}{n} \rceil = \lceil \frac{x+m}{n} \rceil$
- If $n > 0$, $\lfloor \frac{\lfloor \frac{x}{m} \rfloor}{n} \rfloor = \lfloor \frac{x}{mn} \rfloor$
- If $n > 0$, $\lceil \frac{\lceil \frac{x}{m} \rceil}{n} \rceil = \lceil \frac{x}{mn} \rceil$
- For $m, n > 0$, $\sum_{k=1}^{n-1} \lfloor \frac{km}{n} \rfloor = \frac{(m-1)(n-1)+\gcd(m,n)-1}{2}$

2.3 Recurrences

If $a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k}$, and $r_1, \ldots, r_k$ are distinct roots of $x^k - c_1 x^{k-1} - \cdots - c_k$, there are $d_1, \ldots, d_k$ s.t.

$$a_n = d_1 r_1^n + \cdots + d_k r_k^n.$$

Non-distinct roots r become polynomial factors, e.g. $a_n = (d_1 n + d_2) r^n$.

2.4 Trigonometry

$$\sin(v+w) = \sin v \cos w + \cos v \sin w$$
$$\cos(v+w) = \cos v \cos w - \sin v \sin w$$
$$\tan(v+w) = \frac{\tan v + \tan w}{1 - \tan v \tan w}$$
$$\sin v + \sin w = 2 \sin \frac{v+w}{2} \cos \frac{v-w}{2}$$
$$\cos v + \cos w = 2 \cos \frac{v+w}{2} \cos \frac{v-w}{2}$$

$$(V+W)\tan(\frac{v-w}{2}) = (V-W)\tan(\frac{v+w}{2})$$

V, W are sides opposite to angles v, w.
$a \cos x + b \sin x = r \cos(x - \phi)$
$a \sin x + b \cos x = r \sin(x + \phi)$
where $r = \sqrt{a^2+b^2}, \phi = \operatorname{atan2}(b, a)$.

2.5 Geometry

2.5.1 Triangles

Side lengths: a, b, c

Semiperimeter: $p = \dfrac{a+b+c}{2}$

Area: $A = \sqrt{p(p-a)(p-b)(p-c)}$

Circumradius: $R = \dfrac{abc}{4A}$

Inradius: $r = \dfrac{A}{p}$

Length of median (divides triangle into two equal-area triangles):
$m_a = \frac{1}{2}\sqrt{2b^2 + 2c^2 - a^2}$

Length of bisector (divides angles in two):
$s_a = \sqrt{bc\left[1 - (a/(b+c))^2\right]}$

Law of sines, cosines & tangents:
$\dfrac{\sin\alpha}{a} = \dfrac{\sin\beta}{b} = \dfrac{\sin\gamma}{c} = \dfrac{1}{2R}$.....(1)
$a^2 = b^2 + c^2 - 2bc\cos\alpha$.....(2)
$\dfrac{a+b}{a-b} = \dfrac{\tan((\alpha+\beta)/2)}{\tan((\alpha-\beta)/2)}$.....(3)

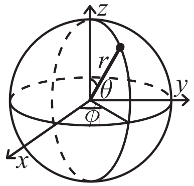
2.5.2 Quadrilaterals

With side lengths a, b, c, d, diagonals e, f, diagonals angle θ, area A and magic flux $F = b^2 + d^2 - a^2 - c^2$:

$$4A = 2ef \cdot \sin\theta = F \tan\theta = \sqrt{4e^2 f^2 - F^2}$$

For cyclic quadrilaterals the sum of opposite angles is $180°$, $ef = ac + bd$, and $A = \sqrt{(p-a)(p-b)(p-c)(p-d)}$

2.5.3 Spherical coordinates



$$\begin{aligned} x &= r \sin\theta \cos\phi & r &= \sqrt{x^2+y^2+z^2} \\ y &= r \sin\theta \sin\phi & \theta &= \operatorname{acos}(z/\sqrt{x^2+y^2+z^2}) \\ z &= r\cos\theta & \phi &= \operatorname{atan2}(y, x) \end{aligned}$$

2.6 Derivatives/Integrals

$$\frac{d}{dx}\arcsin x = \frac{1}{\sqrt{1-x^2}} \qquad \frac{d}{dx}\arccos x = -\frac{1}{\sqrt{1-x^2}}$$
$$\frac{d}{dx}\tan x = 1 + \tan^2 x \qquad \frac{d}{dx}\arctan x = \frac{1}{1+x^2}$$
$$\int \tan ax = -\frac{\ln|\cos ax|}{a} \qquad \int x e^{ax} dx = \frac{e^{ax}}{a^2}(ax-1)$$
$$\int e^{-x^2} = \frac{\sqrt{\pi}}{2}\operatorname{erf}(x) \qquad \int x \sin ax = \frac{\sin ax - ax\cos ax}{a^2}$$

$$\int_a^b f(x)g(x)dx = [F(x)g(x)]_a^b - \int_a^b F(x)g'(x)dx$$

2.7 Sums

$$c^a + c^{a+1} + \cdots + c^b = \frac{c^{b+1} - c^a}{c-1}, c \neq 1$$

$$1^4 + 2^4 + 3^4 + \cdots + n^4 = S_2 \times \frac{3n^2+3n-1}{5} = S_4$$

$$b\sum_{k=0}^{n-1}(a+kd)r^k = \frac{ab - (a+nd)br^n}{1-r} + \frac{dbr(1-r^n)}{(1-r)^2}$$

To compute $1^k + ... + n^k$ in $\mathcal{O}(k \lg k + k \lg MOD)$ compute first $t = k+2$ sums $y_1, ..., y_t$, then interpolate. Let $P = \prod_{i=1}^t (n-i)$. Then answer for n is

$$\sum_{i=1}^t \frac{P}{n-i} \cdot \frac{(-1)^{t-i} y_i}{(i-1)!(t-i)!}$$

Also $S_k = \frac{1}{k+1}\sum_{j=0}^k (-1)^j \binom{k+1}{j} B_j n^{k+1-j}$ where B_i are Bernoulli numbers.

2.8 Series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots, (-\infty < x < \infty)$$
$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots, (-1 < x \leq 1)$$
$$\sqrt{1+x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \ldots, (-1 \leq x \leq 1)$$
$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots, (-\infty < x < \infty)$$
$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots, (-\infty < x < \infty)$$
$$(1-x)^{-r} = \sum_{i=0}^{\infty}\binom{r+i-1}{i}x^i, (r \in \mathbb{R})$$

2.9 Probability theory

Let X be a discrete random variable with probability $p_X(x)$ of assuming the value x. It will then have an expected value (mean) $\mu = \mathbb{E}(X) = \sum_x x p_X(x)$ and variance $\sigma^2 = V(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = \sum_x (x - \mathbb{E}(X))^2 p_X(x)$ where σ is the standard deviation. If X is instead continuous it will have a probability density function $f_X(x)$ and the sums above will instead be integrals with $p_X(x)$ replaced by $f_X(x)$.

Expectation is linear:

$$\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$$

For independent X and Y,

$$V(aX + bY) = a^2 V(X) + b^2 V(Y).$$

2.9.1 Discrete distributions

Binomial distribution

The number of successes in n independent yes/no experiments, each which yields success with probability p is $\operatorname{Bin}(n, p)$, $n = 1, 2, \ldots$, $0 \leq p \leq 1$.

$$p(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

$$\mu = np, \sigma^2 = np(1-p)$$

$\operatorname{Bin}(n, p)$ is approximately $\operatorname{Po}(np)$ for small p.

First success distribution

The number of trials needed to get the first success in independent yes/no experiments, each which yields success with probability p is $\operatorname{Fs}(p)$, $0 \leq p \leq 1$.

$$p(k) = p(1-p)^{k-1}, k = 1, 2, \ldots$$

$$\mu = \frac{1}{p}, \sigma^2 = \frac{1-p}{p^2}$$

Poisson distribution

The number of events occurring in a fixed period of time t if these events occur with a known average rate κ and independently of the time since the last event is $\operatorname{Po}(\lambda)$, $\lambda = t\kappa$.

$$p(k) = e^{-\lambda}\frac{\lambda^k}{k!}, k = 0, 1, 2, \ldots$$

$$\mu = \lambda, \sigma^2 = \lambda$$

2.9.2 Continuous distributions

Uniform distribution

If the probability density function is constant between a and b and 0 elsewhere it is $\operatorname{U}(a, b)$, $a < b$.

$$f(x) = \begin{cases} \frac{1}{b-a} & a < x < b \\ 0 & \text{otherwise} \end{cases}$$

$$\mu = \frac{a+b}{2}, \sigma^2 = \frac{(b-a)^2}{12}$$

Exponential distribution

The time between events in a Poisson process is $\operatorname{Exp}(\lambda)$, $\lambda > 0$.

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

$$\mu = \frac{1}{\lambda}, \sigma^2 = \frac{1}{\lambda^2}$$

Normal distribution

Most real random values with mean μ and variance σ^2 are well described by $\mathcal{N}(\mu, \sigma^2)$, $\sigma > 0$.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

If $X_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$ and $X_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$ then

$$aX_1 + bX_2 + c \sim \mathcal{N}(\mu_1 + \mu_2 + c, a^2\sigma_1^2 + b^2\sigma_2^2)$$

2.10 Markov chains

A *Markov chain* is a discrete random process with the property that the next state depends only on the current state. Let $X_1, X_2, \ldots$ be a sequence of random variables generated by the Markov process. Then there is a transition matrix $\mathbf{P} = (p_{ij})$, with $p_{ij} = \Pr(X_n = i | X_{n-1} = j)$, and $\mathbf{p}^{(n)} = \mathbf{P}^n \mathbf{p}^{(0)}$ is the probability distribution for X_n (i.e., $p_i^{(n)} = \Pr(X_n = i)$), where $\mathbf{p}^{(0)}$ is the initial distribution.

π is a stationary distribution if $\pi = \pi\mathbf{P}$. If the Markov chain is *irreducible* (it is possible to get to any state from any state), then $\pi_i = \frac{1}{\mathbb{E}(T_i)}$ where $\mathbb{E}(T_i)$ is the expected time between two visits in state i. π_j/π_i is the expected number of visits in state j between two visits in state i.

For a connected, undirected and non-bipartite graph, where the transition probability is uniform among all neighbors, π_i is proportional to node i's degree.

A Markov chain is *ergodic* if the asymptotic distribution is independent of the initial distribution. A finite Markov chain is ergodic iff it is irreducible and *aperiodic* (i.e., the gcd of cycle lengths is 1). $\lim_{k\to\infty} \mathbf{P}^k = \mathbf{1}\pi$.

A Markov chain is an A-chain if the states can be partitioned into two sets $\mathbf{A}$ and $\mathbf{G}$, such that all states in $\mathbf{A}$ are absorbing ($p_{ii} = 1$), and all states in $\mathbf{G}$ leads to an absorbing state in $\mathbf{A}$. The probability for absorption in state $i \in \mathbf{A}$, when the initial state is j, is $a_{ij} = p_{ij} + \sum_{k\in\mathbf{G}} a_{ik} p_{kj}$. The expected time until absorption, when the initial state is i, is $t_i = 1 + \sum_{k\in\mathbf{G}} p_{ki} t_k$.

2.11 Trivia

Pythagorean triples: The Pythagorean triples are uniquely generated by $a = k\cdot(m^2 - n^2)$, $b = k\cdot(2mn)$, $c = k\cdot(m^2+n^2)$ with $m > n > 0$, $k > 0$, $\gcd(m, n) = 1$, both m, n not odd.
Primes: $p = 962592769$ is such that $2^{21} \mid p - 1$, which may be useful. For hashing use 970592641 (31-bit number), 31443539979727 (45-bit), 3006703054056749 (52-bit). There are 78498 primes less than 1 000 000.
Primitive roots modulo n exists iff $n = 1, 2, 4$ or, $n = p^k, 2p^k$ where p is an odd prime. Furthermore, the number of roots are $\phi(\phi(n))$.
To Find Generator g of M, factor $M - 1$ and get the distinct primes p_i. If $g^{(M-1)/p_i} \neq 1 (MODM)$ for each p_i then g is a valid root. Try all g until a hit is found (usually found very quick).
Esitmates: $\sum_{d|n} d = O(n \log\log n)$.
Prime count: 5133 upto 5e4. 9592 upto 1e5. 17984 upto 2e5. 78498 upto 1e6. 5761455 upto 1e8.
max NOD $\leq n$: 100 for $n = 5e4$. 500 for $n = 1e7$. 2000 for $n = 1e10$. 200 000 for $n = 1e19$.
max Unique Prime Factors: 6 upto 5e5. 7 upto 9e6. 8 upto 2e8. 9 upto 6e9. 11 upto 7e12. 15 upto 3e19.
Quadratic Residue: $(\frac{a}{p})$ is 0 if $p|a$, 1 if a is a quadratic residue, -1 otherwise. Euler: $(\frac{a}{p}) = a^{(p-1)/2} (\mod p)$ (prime). Jacobi: if $n = p_1^{e_1} \cdots p_k^{e_k}$ then $(\frac{a}{n}) = \prod (\frac{a}{p_i})^{e_i}$.
Chicken McNugget. If a, b coprime, there are $\frac{1}{2}(a-1)(b-1)$ numbers not of form $ax + by$ $(x, y \geq 0)$, the largest being $ab - a - b$.

Data structures (3)

OrderStatisticTree.h

Description: A set (not multiset!) with support for finding the n'th element, and finding the index of an element. To get a map, change `null_type`.
Time: $\mathcal{O}(\log N)$
<ext/pb_ds/assoc_container.hpp>, <ext/pb_ds/tree_policy.hpp>
87dec6, 23 lines

```
using namespace __gnu_pbds;
template <typename T>
using ordered_set = tree<T, null_type, less<T
    >, rb_tree_tag,
    tree_order_statistics_node_update>;
ordered_set<T> o;
template <typename T, typename R>
using o_map = tree<T, R, less<T>, rb_tree_tag,
     tree_order_statistics_node_update>;

//for ordered multiset "upper_bound" and "
    lower_bound" are reversed
//o.erase(--(o.lower_bound(v[i])))
//order_of_key: The number of items in a set
    that are strictly smaller than k
//find_by_order: It returns an iterator to the
    ith largest element

template<class T> using Tree=tree<T, null_type
    , less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;

void example() {
  Tree<int> t, t2; t.insert(8);
  auto it = t.insert(10).first;
  assert(it == t.lower_bound(9));
  assert(t.order_of_key(10) == 1);
  assert(t.order_of_key(11) == 2);
  assert(*t.find_by_order(0) == 8);
  t.join(t2); // assuming T < T2 or T > T2,
      merge t2 into t
}
```

HashMap.h

Description: Hash map with mostly the same API as unordered_map, but ~3x faster. Uses 1.5x memory. Initial capacity must be a power of 2 (if provided).
d77092, 7 lines

```
#include <bits/extc++.h>
// To use most bits rather than just the
    lowest ones:
struct chash { // large odd number for C
  const uint64_t C = ll(4e18 * acos(0)) | 71;
  ll operator()(ll x) const { return
      __builtin_bswap64(x*C); }
};
__gnu_pbds::gp_hash_table<ll,int,chash> h({},{
    },{},{},{1<<16});
```

CustomHash.h

Description: Custom hash function for unordered maps/sets to avoid hacking. Uses splitmix64 algorithm and supports both integer and vector keys.
Usage: See code examples below
e5729a, 25 lines

```
struct custom_hash {
  static uint64_t splitmix64(uint64_t x) {
    x += 0x9e3779b97f4a7c15;
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    return x ^ (x >> 31);
  }

  size_t operator()(uint64_t x) const {
    static const uint64_t FIXED_RANDOM =
        chrono::steady_clock::now().
        time_since_epoch().count();
    return splitmix64(x + FIXED_RANDOM);
  }

  size_t operator()(const vector<int> &v)
      const {
    uint64_t h = 0;
    for (int x: v) {
      h ^= splitmix64(x + 0x9e3779b97f4a7c15 +
          (h << 6) + (h >> 2));
    }
    return h;
  }
};

// Usage examples:
// unordered_map<int, int, custom_hash>
    safe_map;
// unordered_map<vector<int>, int, custom_hash
    > vector_map;
```

XORHashing.h

Description: XOR Hashing for array elements. Maps each unique value to a random number and computes prefix XOR sums. Useful for subarray XOR queries and finding subarrays with given XOR.
Time: $\mathcal{O}(nlogn)$ for preprocessing, $\mathcal{O}(1)$ for queries
435faa, 26 lines

```
void xorHashing(vector<int>& v) {
  map<int, int> hash;
  hash[0] = 0;
  set<int> used = {0};
  int n = v.size();
  vector<int> pref(n + 1);

  for (auto &it: v) {
    int random;
    if (!hash.count(it)) {
      do {
        random = rng();
      } while (used.count(random));
      used.insert(random);
      hash[it] = random;
    }
    else random = hash[it];
    it = random;
  }

  for (int i = 1; i <= n; ++i) {
    pref[i] = pref[i - 1] ^ v[i-1];
  }

  // Now pref[r+1] ^ pref[l] gives XOR of v[l
      ...r]
}
```

SegmentTree.h

Time: $\mathcal{O}(\log N)$
9f8f73, 61 lines

```
template<class S> struct segtree {
  int n; V<S> t;
  void init(int _) { n = _; t.assign(n+n-1, S
      ()); }
  void init(const V<S>& v) {
    n = sz(v); t.assign(n + n - 1, S());
    build(0,0,n-1,v);
  } template <typename... T>
  void upd(int l, int r, const T&... v) {
    assert(0 <= l && l <= r && r < n);
    upd(0, 0, n-1, l, r, v...);
  }
  S get(int l, int r) {
    assert(0 <= l && l <= r && r < n);
    return get(0, 0, n-1, l, r);
  }
private:
  inline void push(int u, int b, int e) {
    if (t[u].lazy == 0) return;
    int mid = (b+e)>>1, rc = u+((mid-b+1)<<1);
    t[u+1].upd(b, mid, t[u].lazy);
    t[rc].upd(mid+1, e, t[u].lazy);
    t[u].lazy = 0;
  }
  void build(int u,int b,int e,const V<S>&v) {
    if (b == e) return void(t[u] = v[b]);
    int mid = (b+e)>>1, rc = u+((mid-b+1)<<1);
    build(u+1, b,mid,v); build(rc, mid+1,e,v);
    t[u] = t[u+1] + t[rc];
  } template<typename... T>
  void upd(int u, int b, int e, int l, int r,
      const T&... v) {
    if (l <= b && e <= r) return t[u].upd(b, e
        , v...);
    push(u, b, e);
    int mid = (b+e)>>1, rc = u+((mid-b+1)<<1);
    if (l<=mid) upd(u+1, b, mid, l, r, v...);
    if (mid<r) upd(rc, mid+1, e, l, r, v...);
    t[u] = t[u+1] + t[rc];
  }
  S get(int u, int b, int e, int l, int r) {
    if (l <= b && e <= r) return t[u];
    push(u, b, e);
    S res; int mid = (b+e)>>1, rc = u+((mid-b
        +1)<<1);
    if (r<=mid) res = get(u+1, b, mid, l, r);
    else if (mid<l) res = get(rc,mid+1,e,l,r);
    else res = get(u+1, b, mid, l, r) + get(rc
        , mid+1, e, l, r);
    t[u] = t[u+1] + t[rc]; return res;
  }
}; // Hash upto here = 773c09
/* (1) Declaration:
Create a node class. Now, segtree<node> T;
T.init(10) creates everything as node()
Consider using V<node> leaves to build
(2) upd(l, r, ...v): update range [l, r]
order in ...v must be same as node.upd() fn */
struct node {
  ll sum = 0, lazy = 0;
  node () {} // write full constructor
  node operator+(const node &obj) {
    return {sum + obj.sum, 0};    }
  void upd(int b, int e, ll x) {
    sum += (e - b + 1) * x, lazy += x;
} };
```

SegmentTreeEasy.h

Description: Easy Segment Tree with point update and range query. Provides merge (combine results), build (construct tree), upd (point update), query (range query). May need to change merge function and identity elements based on problem.
Usage: `int t[4*N]; build(1, 1, n, arr); upd(1, 1, n, idx, val); query(1, 1, n, l, r);`

Time: Build: $\mathcal{O}(N)$, Update: $\mathcal{O}(\log N)$, Query: $\mathcal{O}(\log N)$ cc4113, 25 lines

```cpp
// Global array to store segment tree
int t[N << 2];

// modify as needed
int merge(int a, int b) { return a + b; }

int build(int node, int b, int e, vector<int> &v) {
  if (b == e) return t[node] = v[b];
  int m = (b + e) >> 1;
  return t[node] = merge(build(node << 1, b, m, v), build(node << 1 | 1, m + 1, e, v));
}

void upd(int node, int b, int e, int i, int v) {
  if (b == e) return void(t[node] = v);
  int m = (b + e) >> 1;
  i <= m ? upd(node << 1, b, m, i, v) : upd(node << 1 | 1, m + 1, e, i, v);
  t[node] = merge(t[node << 1], t[node << 1 | 1]);
}

int query(int node, int b, int e, int l, int r) {
  if (b > r || e < l) return 0; // Identity element - modify as needed
  if (b >= l && e <= r) return t[node];
  int m = (b + e) >> 1;
  return merge(query(node << 1, b, m, l, r), query(node << 1 | 1, m + 1, e, l, r));
}
```

SegmentTreeIterative.h

Description: Iterative Segment Tree implementation with point update and range query. Uses 1-based indexing internally but 0-based indexing for the interface. Define op and identity based on the problem.

Usage: `ST st(n); st.upd(idx, val); st.query(l, r);`

Time: Build: $\mathcal{O}(N)$, Update: $\mathcal{O}(\log N)$, Query: $\mathcal{O}(\log N)$ b62892, 28 lines

```cpp
// modify as needed
int op(int a, int b) { return a + b; }

struct ST {
  int n;
  vector<int> t;

  ST(int _n) {
    n = _n;
    t.assign(2 * n + 1, 0);
  }

  void upd(int p, int v) {
    p += n;
    t[p] = v;
    for (p >>= 1; p > 0; p >>= 1) t[p] = op(t[p << 1], t[p << 1 | 1]);
  }

  int query(int l, int r) {
    int res = Identity;
    l += n, r += n;
    for (; l<=r; l>>=1, r>>=1) {
      if (l & 1) res = op(res, t[l++]);
      if (!(r & 1)) res = op(res, t[r--]);
    }
    return res;
  }
};
```

LazySegmentTree.h

Description: Simple Lazy Segment Tree with range updates and range queries. Supports both range sum updates and range assignment updates (commented options). Modify merge function and identity elements based on the problem.

Usage: `ST st(n); st.upd(l, r, val); st.query(l, r);`

Time: Build: $\mathcal{O}(N)$, Update: $\mathcal{O}(\log N)$, Query: $\mathcal{O}(\log N)$ 39a2d5, 53 lines

```cpp
// modify as needed
int merge(int a, int b) { return a + b; }

struct ST {
  int n;
  vector<int> t, lazy;
  const int LAZY_DEFAULT = -1; // may need to change
  const int IDENTITY = 0; // may need to change

  ST(int _n) {
    n = _n;
    t.assign(4 * n, IDENTITY);
    lazy.assign(4 * n, LAZY_DEFAULT);
  }

  void push(int node, int b, int e) {
    if (lazy[node] == LAZY_DEFAULT) return;
    t[node] += (e - b + 1) * lazy[node];
    // t[node] = (e - b + 1) * lazy[node];
    if (b != e) {
      lazy[node << 1] += lazy[node];
      lazy[(node << 1) | 1] += lazy[node];
      // lazy[node << 1] = lazy[(node << 1) | 1] = lazy[node];
    }
    lazy[node] = LAZY_DEFAULT;
  }

  void upd(int node, int b, int e, int l, int r, int v) {
    push(node, b, e);
    if (b > r || e < l) return;
    if (b >= l && e <= r) {
      lazy[node] += v;
      // lazy[node] = v;
      push(node, b, e);
      return;
    }
    int m = (b + e) >> 1;
    upd(node << 1, b, m, l, r, v);
    upd((node << 1) | 1, m + 1, e, l, r, v);
    t[node] = merge(t[node << 1], t[(node << 1) | 1]);
  }

  int query(int node, int b, int e, int l, int r) {
    push(node, b, e);
    if (b > r || e < l) return IDENTITY; // Identity element
    if (b >= l && e <= r) return t[node];
    int m = (b + e) >> 1;
    return merge(query(node << 1, b, m, l, r), query((node << 1) | 1, m + 1, e, l, r));
  }

  void upd(int l, int r, int val) { upd(1, 1, n, l, r, val); }
  int query(int l, int r) { return query(1, 1, n, l, r); }
};
```

DynamicSegmentTree.h

Description: Dynamic Segment Tree

Usage: `NODE *root = new NODE(); upd(root, 0, N-1, pos, delta); query(root, 0, N-1, L, R);` d45cb8, 27 lines

```cpp
//Dynamic ST
struct NODE {
  int v;
  NODE *l, *r;
};

void upd(NODE *node, int b, int e, int i, int v) {
  if (b > i || e < i) return;
  if (b == e) return void(node->v += v);
  int mid = (b + e) / 2;
  if (i <= mid) {
    if (!node->l) node->l = new NODE();
    upd(node->l, b, mid, i, v);
  }
  else {
    if (!node->r) node->r = new NODE();
    upd(node->r, mid + 1, e, i, v);
  }
  node->v = (node->l ? node->l->v : 0) + (node->r ? node->r->v : 0);
}

int query(NODE *node, int b, int e, int l, int r) {
  if (b > r || e < l || node == nullptr)
    return 0;
  if (b >= l && e <= r) return node->v;
  int mid = (b + e) / 2;
  return query(node->l, b, mid, l, r) + query(node->r, mid + 1, e, l, r);
}
```

PersistentSegmentTree.h

Description: Persistent segment tree implementation. Each update creates a new version of the tree while preserving the previous versions. (0 based indexing)

Time: $\mathcal{O}(logn)$ per query/update aab0aa, 70 lines

```cpp
const int N = 2e5 + 9;

struct Node {
  Node *left, *right;
  ll sum;
  Node() : left(nullptr), right(nullptr), sum(0) {}
  Node(Node* l, Node* r, ll val) : left(l), right(r), sum(val) {}
};

// Build initial tree from array
Node* build(int l, int r, vector<int>& arr) {
  if (l == r)
    return new Node(nullptr, nullptr, arr[l]);

  int mid = (l + r) / 2;
  Node* left_child = build(l, mid, arr);
  Node* right_child = build(mid + 1, r, arr);

  return new Node(left_child, right_child, left_child->sum + right_child->sum);
}

// Create a new version with updated value at position
Node* update(Node* node, int l, int r, int pos, int val) {
  if (l == r)
    return new Node(nullptr, nullptr, val);

  int mid = (l + r) / 2;
  if (pos <= mid) {
    Node* new_left = update(node->left, l, mid, pos, val);
    return new Node(new_left, node->right, new_left->sum + node->right->sum);
  } else {
    Node* new_right = update(node->right, mid + 1, r, pos, val);
    return new Node(node->left, new_right, node->left->sum + new_right->sum);
  }
}

// Query sum in range [ql, qr] on a specific version of the tree
ll query(Node* node, int l, int r, int ql, int qr) {
  if (ql > r || qr < l)
    return 0;
  if (ql <= l && r <= qr)
    return node->sum;

  int mid = (l + r) / 2;
  return query(node->left, l, mid, ql, qr) +
         query(node->right, mid + 1, r, ql, qr);
}

/*
Example usage:
int main() {
  int n = 5;
  vector<int> arr = {1, 2, 3, 4, 5};

  // Build initial tree (version 0)
  Node* version0 = build(0, n-1, arr);

  // Create version 1 by updating arr[2] to 10
  Node* version1 = update(version0, 0, n-1, 2, 10);

  // Query both versions
  ll sum_v0 = query(version0, 0, n-1, 1, 3); // Sum of elements [1,2,3] in version 0
  ll sum_v1 = query(version1, 0, n-1, 1, 3); // Sum of elements [1,2,3] in version 1
```

```
  cout << "Sum in version 0: " << sum_v0 << '\n'; // Output: 9
  cout << "Sum in version 1: " << sum_v1 << '\n'; // Output: 16

  return 0;
}
*/
```

MergeSortTree.h

Description: Merge Sort Tree for range queries counting elements smaller/greater than k. Find the number of elements smaller and greater than k in a given range $[l, r]$. In case of updates, use ordered_multiset instead of vector.
Usage: build(1, 1, n); query(1, 1, n, l, r, k);
Time: Build: $\mathcal{O}(N \log N)$, Query: $\mathcal{O}(\log^2 N)$

ff0da0, 32 lines

```
vector<int> t[N << 2];

vector<int> merge(const vector<int> &a, const
    vector<int> &b) {
  vector<int> ret;
  int i = 0, j = 0;
  while (i < (int)a.size() && j < (int)b.size
      ()) {
    if (a[i] < b[j]) ret.push_back(a[i++]);
    else ret.push_back(b[j++]);
  }
  while (i < (int)a.size()) ret.push_back(a[i
      ++]);
  while (j < (int)b.size()) ret.push_back(b[j
      ++]);
  return ret;
}

vector<int> build(int node, int b, int e) {
  if (b == e) return t[node] = { a[b] };
  int m = (b + e) >> 1;
  return t[node] = merge(build(node << 1, b, m
      ), build((node << 1) | 1, m + 1, e));
}

pii query(int node, int b, int e, int l, int r
    , int k) {
  if (b > r || e < l) return {0, 0};
  if (b >= l && e <= r) {
    int smaller = lower_bound(t[node].begin(),
        t[node].end(), k) - t[node].begin();
    int greater = (int)t[node].size() - (
        upper_bound(t[node].begin(), t[node].
        end(), k) - t[node].begin());
    return {smaller, greater};
  }
  int m = (b + e) >> 1;
  auto q1 = query(node << 1, b, m, l, r, k);
  auto q2 = query((node << 1) | 1, m + 1, e, l
      , r, k);
  return {q1.first + q2.first, q1.second + q2.
      second};
}
```

DSU.h

Description: Disjoint Set Union (DSU) implementation
Time: $\mathcal{O}(n)$ amortized

2ab1de, 11 lines

```
vector<int> parent(N), siz(N, 1);
iota(parent.begin(),parent.end(),0);
int find(int v) { return v == parent[v] ? v :
    parent[v] = find(parent[v]); }
void unite(int a, int b) {
  a = find(a), b = find(b);
  if (a != b) {
    if (siz[a] < siz[b]) swap(a, b);
    parent[b] = a;
    siz[a] += siz[b];
  }
}
```

UnionFindRollback.h

Description: Disjoint-set data structure with undo. If undo is not needed, skip st, time() and rollback().
Usage: int t = uf.time(); ...; uf.rollback(t);
Time: $\mathcal{O}(\log(N))$

de4ad0, 21 lines

```
struct RollbackUF {
  vi e; vector<pii> st;
  RollbackUF(int n) : e(n, -1) {}
  int size(int x) { return -e[find(x)]; }
  int find(int x) { return e[x] < 0 ? x : find
      (e[x]); }
  int time() { return sz(st); }
  void rollback(int t) {
    for (int i = time(); i --> t;)
      e[st[i].first] = st[i].second;
    st.resize(t);
  }
  bool join(int a, int b) {
    a = find(a), b = find(b);
    if (a == b) return false;
    if (e[a] > e[b]) swap(a, b);
    st.push_back({a, e[a]});
    st.push_back({b, e[b]});
    e[a] += e[b]; e[b] = a;
    return true;
  }
};
```

LineContainer.h

Description: Container where you can add lines of the form kx+m, and query maximum values at points x. Useful for dynamic programming ("convex hull trick").
Time: $\mathcal{O}(\log N)$

8ec1c7, 30 lines

```
struct Line {
  mutable ll k, m, p;
  bool operator<(const Line& o) const { return
      k < o.k; }
  bool operator<(ll x) const { return p < x; }
};

struct LineContainer : multiset<Line, less<>>
    {
  // (for doubles, use inf = 1/.0, div(a,b) =
      a/b)
  static 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()) return x->p = inf, 0;
    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) {
    assert(!empty());
    auto l = *lower_bound(x);
    return l.k * x + l.m;
  }
};
```

Lichao.h

Description: Add line segment, query minimum y at some x. Provide list of all query x points to constructor (offline solution). Use add_segment(line, l, r) to add a line segment $y = ax + b$ defined by $x \in [l, r)$. Use query(x) to get min at x.
Time: Both operations are $\mathcal{O}(\log \text{maxn})$.

566134, 43 lines

```
struct LiChaoTree {
  using Line = pair <ll, ll>;
  const ll linf = numeric_limits<ll>::max();
  int n; vector<ll> xl; vector<Line> dat;
  LiChaoTree(const vector<ll>& _xl):xl(_xl){
    n = 1; while(n < xl.size())n <<= 1;
    xl.resize(n,xl.back());
    dat = vector<Line>(2*n-1, Line(0,linf));
  }
  ll eval(Line f,ll x){return f.first * x + f.
      second;}
  void _add_line(Line f,int k,int l,int r){
    while (l != r) {
      int m = (l + r) / 2;
      ll lx = xl[l],mx = xl[m],rx = xl[r - 1];
      Line &g = dat[k];
      if(eval(f,lx) < eval(g,lx) && eval(f,rx)
          < eval(g,rx)) {
        g = f; return;
      }
      if(eval(f,lx) >= eval(g,lx) && eval(f,rx
          ) >= eval(g,rx))
        return;
      if(eval(f,mx) < eval(g,mx))swap(f,g);
      if(eval(f,lx) < eval(g,lx)) k = k * 2 +
          1, r = m;
      else k = k * 2 + 2, l = m;
    }
  }
  void add_line(Line f){_add_line(f,0,0,n);}
  void add_segment(Line f,ll lx,ll rx){
    int l = lower_bound(xl.begin(), xl.end(),
        lx) - xl.begin();
    int r = lower_bound(xl.begin(), xl.end(),
        rx) - xl.begin();
    int a0 = l, b0 = r, sz = 1; l += n;r += n;
    while(l < r){
      if(r & 1) r--, b0 -= sz, _add_line(f,r -
          1,b0,b0 + sz);
      if(l & 1) _add_line(f,l - 1,a0,a0 + sz),
          l++, a0 += sz;
      l >>= 1, r >>= 1, sz <<= 1;
    }
  }
  ll query(ll x) {
    int i = lower_bound(xl.begin(), xl.end(),x
        ) - xl.begin();
    i += n - 1; ll res = eval(dat[i],x);
    while (i) i = (i - 1) / 2, res = min(res,
        eval(dat[i], x));
    return res;
  }
};
```

Treap.h

Description: A short self-balancing tree. It acts as a sequential container with log-time splits/joins, and is easy to augment with additional data.
Time: $\mathcal{O}(\log N)$

1754b4, 53 lines

```
struct Node {
  Node *l = 0, *r = 0;
  int val, y, c = 1;
  Node(int val) : val(val), y(rand()) {}
  void recalc();
};

int cnt(Node* n) { return n ? n->c : 0; }
void Node::recalc() { c = cnt(l) + cnt(r) + 1;
    }

template<class F> void each(Node* n, F f) {
  if (n) { each(n->l, f); f(n->val); each(n->r
      , f); }
}

pair<Node*, Node*> split(Node* n, int k) {
  if (!n) return {};
  if (cnt(n->l) >= k) { // "n->val >= k" for
      lower_bound(k)
    auto [L,R] = split(n->l, k);
    n->l = R;
    n->recalc();
    return {L, n};
  } else {
    auto [L,R] = split(n->r,k - cnt(n->l) - 1)
        ; // and just "k"
    n->r = L;
    n->recalc();
    return {n, R};
  }
}

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);
    return l->recalc(), l;
  } else {
    r->l = merge(l, r->l);
    return r->recalc(), r;
  }
}

Node* ins(Node* t, Node* n, int pos) {
  auto [l,r] = split(t, pos);
  return merge(merge(l, n), r);
}

// Example application: move the range [l, r)
    to index k
void move(Node*& t, int l, int r, int k) {
  Node *a, *b, *c;
```

```
  tie(a,b) = split(t, l); tie(b,c) = split(b,
       r - l);
  if (k <= l) t = merge(ins(a, b, k), c);
  else t = merge(a, ins(c, b, k - r));
}
```

FenwickTree.h

Description: `update(i,x): a[i] += x;` `query(i): sum in [0, i);` `lower_bound(sum): min pos st sum of [0, pos] >= sum, returns n if all < sum, or -1 if empty sum.`

Time: Both operations are $\mathcal{O}(\log N)$. f74d01, 16 lines

```
struct FT {
  int n; V<ll> s;
  FT(int _n) : n(_n), s(_n) {}
  void update(int i, ll x) {
    for (; i < n; i |= i + 1) s[i] += x; }
  ll query(int i, ll r = 0) {
    for (; i > 0; i &= i - 1) r += s[i-1];
        return r; }
  int lower_bound(ll sum) {
    if (sum <= 0) return -1; int pos = 0;
    for (int pw = 1 << __lg(n); pw; pw >>= 1){
      if (pos+pw <= n && s[pos + pw-1] < sum)
        pos += pw, sum -= s[pos-1];
    }
    return pos;
  }
}; // Hash = d05c4f without lower_bound
```

FenwickTree2.h

Description: 1-indexed Fenwick Tree with range query and range update capabilities. `query(i): sum in [1, i];` `query(l, r): sum in [l, r];` `upd(i, val): a[i] += val;` `upd(l, r, val): a[l..r] += val;`

Time: All operations are $\mathcal{O}(\log N)$. c980d1, 26 lines

```
template<class T> struct BIT { //1-indexed
  int n;
  vector<T> t;
  BIT() {
  }
  BIT(int _n) {
    n = _n;
    t.assign(n + 1, 0);
  }
  T query(int i) {
    T ans = 0;
    for (; i >= 1; i -= (i & -i)) ans += t[i];
    return ans;
  }
  void upd(int i, T val) {
    if (i <= 0) return;
    for (; i <= n; i += (i & -i)) t[i] += val;
  }
  void upd(int l, int r, T val) {
    upd(l, val);
    upd(r + 1, -val);
  }
  T query(int l, int r) {
    return query(r) - query(l - 1);
  }
};
```

FenwickTreeRange.h

Description: Range add Range sum with FT.

Time: Both operations are $\mathcal{O}(\log N)$. 8fc549, 11 lines

```
FT f1(n), f2(n);
// a[l...r] += v; 0 <= l <= r < n
auto upd = [&](int l, int r, ll v) {
  f1.update(l, v), f1.update(r + 1, -v);
  f2.update(l, v*(l-1)), f2.update(r+1, -v*r);
}; // a[l] + ... + a[r]; 0 <= l <= r < n
auto sum = [&](int l, int r) { ++r;
  ll sub = f1.query(l) * (l-1) - f2.query(l);
  ll add = f1.query(r) * (r-1) - f2.query(r);
  return add - sub;
};
```

FenwickTree2d.h

Description: Computes sums a[i,j] for all i<I, j<J, and increases single elements a[i,j]. Requires that the elements to be updated are known in advance (call fakeUpdate() before init()).

Time: $\mathcal{O}(\log^2 N)$. (Use persistent segment trees for $\mathcal{O}(\log N)$.)

"FenwickTree.h" d53ef2, 20 lines

```
struct FT2 {
  V<vi> ys; V<FT> ft;
  FT2(int limx) : ys(limx) {}
  void fakeUpdate(int x, int y) {
    for (;x<sz(ys);x|=x+1) ys[x].push_back(y);
  }
  void init() { for (vi& v : ys)
    sort(all(v)), ft.emplace_back(sz(v));
  }
  int ind(int x, int y) {
    return (int)(lower_bound(all(ys[x]), y) -
        ys[x].begin()); }
  void update(int x, int y, ll dif) {
    for (; x < sz(ys); x |= x + 1)
      ft[x].update(ind(x, y), dif);
  }
  ll query(int x, int y) { ll sum = 0;
    for (; x; x &= x - 1)
      sum += ft[x-1].query(ind(x-1, y));
    return sum;
  }
};
```

RMQ.h

Description: Range Minimum Queries on an array. Returns min(V[a], V[a + 1], ... V[b - 1]) in constant time.

Usage: `RMQ rmq(values);` `rmq.query(inclusive, exclusive);`

Time: $\mathcal{O}(|V|\log|V| + Q)$ 7d2211, 15 lines

```
template<class T>
struct RMQ {
  V<V<T>> jmp;
  RMQ(const V<T>& V) : jmp(1, V) {
    for (int pw = 1, k = 1; pw * 2 <= sz(V);
        pw *= 2, ++k) {
      jmp.emplace_back(sz(V) - pw * 2 + 1);
      rep(j,0,sz(jmp[k]))
        jmp[k][j] = min(jmp[k - 1][j], jmp[k -
            1][j + pw]);
    }
  }
  T query(int a, int b) {
    assert(a < b); // or return inf if a == b
    int dep = 31 - __builtin_clz(b - a);
    return min(jmp[dep][a], jmp[dep][b - (1 <<
        dep)]);
} };
```

SparseTable.h

Description: Sparse Table for efficient range queries with idempotent operations. Supports two query methods: general O(log n) and O(1) for idempotent operations like min/max.

Usage: `SparseTable<int, min> st(v);` `st.query(l, r);`

Time: Build: $\mathcal{O}(nlogn)$, Query: $\mathcal{O}(1)$ for idempotent ops, $\mathcal{O}(logn)$ for others. 94bcde, 31 lines

```
template<typename T, T (*op)(T, T)> struct
    SparseTable {
  vector<vector<T> > t;
  int K, N;

  SparseTable(const vector<T> &v): t(1, v) {
    K = __lg(sz(v)), N = sz(v);
    for (int i = 1; i <= K; ++i) {
      t.emplace_back(sz(v) - (1 << i) + 1);
      for (int j = 0; j + (1 << i) <= N; ++j)
        t[i][j] = op(t[i - 1][j], t[i - 1][j
          + (1 << (i - 1))]);
    }
  }

  // O(log n) query - for non-idempotent
       operations (e.g., sum)
  T query(int L, int R) {
    int ret = 0; // may need to change
    for (int i = K; i >= 0; --i) {
      if ((1 << i) <= R - L + 1) {
        ret += t[i][L]; // may need to change
        L += 1 << i;
      }
    }
    return ret;
  }

  // O(1) query - for idempotent operations (e
       .g., min, max, gcd)
  T query(int L, int R) {
    assert(L <= R);
    int i = __lg(R - L + 1);
    return op(t[i][L], t[i][R - (1 << i) + 1])
        ;
  }
};
```

MoQueries.h

Description: Answer interval or tree path queries by finding an approximate TSP through the queries, and moving from one query to the next by adding/removing points at the ends. If values are on tree edges, change `step` to add/remove the edge (a, c) and remove the initial `add` call (but keep `in`).

Time: $\mathcal{O}(N\sqrt{Q})$ a12ef4, 49 lines

```
void add(int ind, int end) { ... } // add a[
     ind] (end = 0 or 1)
void del(int ind, int end) { ... } // remove a
     [ind]
int calc() { ... } // compute current answer

vi mo(vector<pii> Q) {
  int L = 0, R = 0, blk = 350; // ~N/sqrt(Q)
  vi s(sz(Q)), res = s;
#define K(x) pii(x.first/blk, x.second ^ -(x.
     first/blk & 1))
  iota(all(s), 0);
  sort(all(s), [&](int s, int t){ return K(Q[s
       ]) < K(Q[t]); });
  for (int qi : s) {
    pii q = Q[qi];
    while (L > q.first) add(--L, 0);
    while (R < q.second) add(R++, 1);
    while (L < q.first) del(L++, 0);
    while (R > q.second) del(--R, 1);
    res[qi] = calc();
  }
  return res;
}

vi moTree(vector<array<int, 2>> Q, vector<vi>&
     ed, int root=0){
  int N = sz(ed), pos[2] = {}, blk = 350; // ~
       N/sqrt(Q)
  vi s(sz(Q)), res = s, I(N), L(N), R(N), in(N
       ), par(N);
  add(0, 0), in[0] = 1;
  auto dfs = [&](int x, int p, int dep, auto&
       f) -> void {
    par[x] = p;
    L[x] = N;
    if (dep) I[x] = N++;
    for (int y : ed[x]) if (y != p) f(y, x, !
         dep, f);
    if (!dep) I[x] = N++;
    R[x] = N;
  };
  dfs(root, -1, 0, dfs);
#define K(x) pii(I[x[0]] / blk, I[x[1]] ^ -(I[
     x[0]] / blk & 1))
  iota(all(s), 0);
  sort(all(s), [&](int s, int t){ return K(Q[s
       ]) < K(Q[t]); });
  for (int qi : s) rep(end,0,2) {
    int &a = pos[end], b = Q[qi][end], i = 0;
#define step(c) { if (in[c]) { del(a, end); in
     [a] = 0; } \
                  else { add(c, end); in[c] =
                      1; } a = c; }
    while (!(L[b] <= L[a] && R[a] <= R[b]))
      I[i++] = b, b = par[b];
    while (a != b) step(par[a]);
    while (i--) step(I[i]);
    if (end) res[qi] = calc();
  }
  return res;
}
```

MoAlgorithm2.h

Description: Mo's Algorithm with block sorting.

Time: $\mathcal{O}(N\sqrt{Q})$ a166e8, 17 lines

```
void mo_s_algorithm() {
  int n = /* array size */;
  int block_sz = sqrt(n);
  sort(queries.begin(), queries.end(), [&](
      auto i, auto j) -> bool {
    return make_pair(i.first.first / block_sz,
         i.first.second) <
          make_pair(j.first.first / block_sz,
              j.first.second);
  });
```

```
  int curr_l = 0, curr_r = -1;
  for (auto query : queries) {
    int l = query.first.first, r = query.first
        .second;
    while (l < curr_l) curr_l--;
    while (l > curr_l) curr_l++;
    while (r > curr_r) curr_r++;
    while (r < curr_r) curr_r--;
    // answers[query.second] = current_result;
  }
}
```

MoTree.h

Description: Build Euler tour of $2N$ size - write node at first enter and last exit. Now, Path(u, v) with $in[u] < in[v]$ is a segment. If lca$(u, v) = u$ then it is $[in[u], in[v]]$. Otherwise it is $[out[u], in[v]]$ + LCA node. Nodes that appear exactly once in each segment are relevant, ignore others, handle LCA separately.
Time: $\mathcal{O}\left(Q\sqrt{N}\right)$

MoTreeImplementation.h

Description: Mo's Algorithm on Trees (using Euler Tour + Mo's on array) for querying properties of paths between nodes such as counting distinct colors.
Time: $\mathcal{O}((N+Q) * sqrt(2*N))$ e72eb1, 123 lines

```
const int N = 2e5 + 9;
vector<int> adj[N];
int color[N];
int st[N], en[N], euler[2 * N], timer;
int depth[N], up[20][N];

// For Mo
int BLOCK;

struct Query {
  int l, r, idx, lca;
};

vector<Query> queries;
long long answer[N];
int freq[N], cntDistinct;
bool inPath[N];

// Euler Tour: record entry and exit, flatten
    tree
void dfs(int u, int p) {
  up[0][u] = p;
  depth[u] = (p == -1 ? 0 : depth[p] + 1);
  st[u] = timer;
  euler[timer++] = u;
  for (int v: adj[u]) if (v != p) dfs(v, u);
  en[u] = timer;
  euler[timer++] = u;
}

int lca(int u, int v) {
  if (depth[u] < depth[v]) swap(u, v);
  int diff = depth[u] - depth[v];
  for (int i = 0; i < 20; i++) if (diff >> i &
      1) u = up[i][u];
  if (u == v) return u;
  for (int i = 19; i >= 0; i--)
    if (up[i][u] != up[i][v]) {
      u = up[i][u];
      v = up[i][v];
    }
  return up[0][u];
}

// Toggle node in current window
void add(int u) {
  if (inPath[u]) {
    // remove
    freq[color[u]]--;
    if (freq[color[u]] == 0) cntDistinct--;
  }
  else {
    // add
    if (freq[color[u]] == 0) cntDistinct++;
    freq[color[u]]++;
  }
  inPath[u] = !inPath[u];
}

/*
Example usage:
int main() {
  int n, q;
  cin >> n >> q;
  for (int i = 1; i <= n; i++) cin >> color[i
      ];
  for (int i = 1, u, v; i < n; i++) {
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
  }

  // Initialize data structures
  timer = 0;
  memset(up, -1, sizeof up);

  // Build Euler tour and LCA table
  dfs(1, -1);

  // Build ancestor table
  for (int i = 1; i < 20; i++)
    for (int v = 1; v <= n; v++)
      up[i][v] = (up[i-1][v] == -1 ? -1 : up[i
          -1][up[i-1][v]]);

  // Prepare queries
  BLOCK = sqrt(timer);
  for (int i = 0, u, v; i < q; i++) {
    cin >> u >> v;
    if (st[u] > st[v]) swap(u, v);
    int w = lca(u, v);
    if (w == u) {
      queries.push_back({st[u], st[v], i, -1})
          ;
    }
    else {
      queries.push_back({en[u], st[v], i, w});
    }
  }

  // Sort queries using Mo's algorithm
      ordering
  sort(queries.begin(), queries.end(), [&](
      const Query &a, const Query &b) {
    if (a.l / BLOCK != b.l / BLOCK) return a.l
        < b.l;
    return a.r < b.r;
  });

  // Process queries
  int curL = 0, curR = -1;
  cntDistinct = 0;
  memset(freq, 0, sizeof freq);
  memset(inPath, 0, sizeof inPath);

  for (auto &qr: queries) {
    while (curL > qr.l) add(euler[--curL]);
    while (curR < qr.r) add(euler[++curR]);
    while (curL < qr.l) add(euler[curL++]);
    while (curR > qr.r) add(euler[curR--]);
    if (qr.lca != -1) add(qr.lca);
    answer[qr.idx] = cntDistinct;
    if (qr.lca != -1) add(qr.lca);
  }

  // Output results
  for (int i = 0; i < q; i++) cout << answer[i
      ] << '\n';

  return 0;
}
*/
```

MoUpdate.h

Description: Set block size $B = (2n^2)^{1/3}$. Sort queries by $(\lfloor \frac{L}{B} \rfloor, \lfloor \frac{R}{B} \rfloor, t)$, where t = number of updates before this query. Then process queries in sorted order, modify L, R and then apply/undo the updates to answer.
Time: $\mathcal{O}\left(Bq + qn^2/B^2\right)$ or $\mathcal{O}\left(qn^{2/3}\right)$ with that B.

Numerical (4)

4.1 Polynomials and recurrences

BerlekampMassey.h

Description: Recovers any n-order linear recurrence relation from the first $2n$ terms of the recurrence. Useful for guessing linear recurrences after brute-forcing the first terms. Should work on any field, but numerical stability for floats is not guaranteed. Output will have size $\leq n$.
Usage: `berlekampMassey({0, 1, 1, 3, 5, 11}) // {1, 2} --> c[n] = c[n-1] + 2c[n-2]`
Time: $\mathcal{O}\left(N^2\right)$
"../number-theory/ModPow.h" 96548b, 20 lines

```
vector<ll> berlekampMassey(vector<ll> s) {
  int n = sz(s), L = 0, m = 0;
  vector<ll> C(n), B(n), T;
  C[0] = B[0] = 1;

  ll b = 1;
  rep(i,0,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;
    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.resize(L + 1); C.erase(C.begin());
  for (ll& x : C) x = (mod - x) % mod;
  return C;
}
```

LinearRecurrence.h

Description: Generates the k'th term of an n-order linear recurrence $S[i] = \sum_j S[i-j-1]tr[j]$, given $S[0 \ldots \geq n-1]$ and $tr[0 \ldots n-1]$. Faster than matrix multiplication. Useful together with Berlekamp–Massey.
Usage: `linearRec({0, 1}, {1, 1}, k) // k'th Fibonacci number`
Time: $\mathcal{O}\left(n^2 \log k\right)$ f4e444, 26 lines

```
typedef vector<ll> Poly;
ll linearRec(Poly S, Poly tr, ll k) {
  int n = sz(tr);

  auto combine = [&](Poly a, Poly b) {
    Poly res(n * 2 + 1);
    rep(i,0,n+1) rep(j,0,n+1)
      res[i + j] = (res[i + j] + a[i] * b[j])
          % mod;
    for (int i = 2 * n; i > n; --i) rep(j,0,n)
      res[i - 1 - j] = (res[i - 1 - j] + res[i
          ] * tr[j]) % mod;
    res.resize(n + 1);
    return res;
  };

  Poly 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,0,n) res = (res + pol[i + 1] * S[i]) %
      mod;
  return res;
}
```

Polynomial.h

c9b7b0, 17 lines

```
struct Poly {
  vector<double> a;
  double operator()(double x) const {
    double val = 0;
    for (int i=sz(a); i--;) (val*=x) += a[i];
    return val;
  }
  void diff() {
    rep(i,1,sz(a)) a[i-1] = i*a[i];
    a.pop_back();
  }
  void divroot(double x0) {
    double b = a.back(), c; a.back() = 0;
    for(int i=sz(a)-1; i--;) c = a[i], a[i] =
        a[i+1]*x0+b, b=c;
    a.pop_back();
  }
};
```

PolyRoots.h

Description: Finds the real roots to a polynomial.
Usage: `polyRoots({{2,-3,1}},-1e9,1e9) // solve x^2-3x+2 = 0`

Time: $\mathcal{O}\left(n^2 \log(1/\epsilon)\right)$

"Polynomial.h" b00bfe, 23 lines

```
vector<double> polyRoots(Poly p, double xmin,
    double xmax) {
  if (sz(p.a)==2) { return {-p.a[0]/p.a[1]}; }
  vector<double> ret;
  Poly der = p;
  der.diff();
  auto dr = polyRoots(der, xmin, xmax);
  dr.push_back(xmin-1);
  dr.push_back(xmax+1);
  sort(all(dr));
  rep(i,0,sz(dr)-1) {
    double l = dr[i], h = dr[i+1];
    bool sign = p(l) > 0;
    if (sign ^ (p(h) > 0)) {
      rep(it,0,60) { // while (h - l > 1e-8)
        double m = (l + h) / 2, f = p(m);
        if ((f <= 0) ^ sign) l = m;
        else h = m;
      }
      ret.push_back((l + h) / 2);
    }
  }
  return ret;
}
```

PolyInterpolate.h

Description: Given n points (x[i], y[i]), computes an n-1-degree polynomial p that passes through them: $p(x) = a[0] * x^0 + ... + a[n-1] * x^{n-1}$. For numerical precision, pick $x[k] = c * \cos(k/(n-1) * \pi), k = 0 \dots n-1$. For fast interpolation in $O(n \log^2 n)$ use Lagrange. $P(x) = \sum_{i=1}^{n} y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$. To compute values $\prod_{j \neq i}(x_i - x_j)$ fast, compute $A(x) = \prod_{i=1}^{n}(x - x_i)$ with divide and conquer. The required values are $A'(x_i)$, (values at derivative), compute fast with multipoint evaluation.

Time: $\mathcal{O}\left(n^2\right)$

08bf48, 13 lines

```
typedef vector<double> vd;
vd interpolate(vd x, vd y, int n) {
  vd res(n), temp(n);
  rep(k,0,n-1) rep(i,k+1,n)
    y[i] = (y[i] - y[k]) / (x[i] - x[k]);
  double last = 0; temp[0] = 1;
  rep(k,0,n) rep(i,0,n) {
    res[i] += y[k] * temp[i];
    swap(last, temp[i]);
    temp[i] -= last * x[k];
  }
  return res;
}
```

4.2 Fourier transforms

FastFourierTransform.h

Description: fft(a) computes $\hat{f}(k) = \sum_x a[x] \exp(2\pi i \cdot kx/N)$ for all k. N must be a power of 2. Useful for convolution: `conv(a, b) = c`, where $c[x] = \sum a[i]b[x-i]$. For convolution of complex numbers or more than two vectors: FFT each poly, multiply result pointwise, divide by n, reverse(out.begin()+1,end), FFT back, then round(out[i].real()) or truncate out[i].imag(0). Rounding is safe if $(\sum a_i^2 + \sum b_i^2) \log_2 N < 9 \cdot 10^{14}$ (in practice 10^{16}; higher for random inputs). Otherwise, use NTT/FFTMod.

Time: $\mathcal{O}(N \log N)$ with $N = |A|+|B|$ ($\sim$1s for $N = 2^{22}$)

3dd197, 36 lines

```
typedef complex<double> C;
typedef vector<double> vd;
void fft(vector<C>& a) {
  int n = sz(a), L = 31 - __builtin_clz(n);
  static vector<complex<long double>> R(2, 1);
  static vector<C> rt(2, 1);  // (^ 10% faster
      if double)
  for (static int k = 2; k < n; k *= 2) {
    R.resize(n); rt.resize(n);
    auto x = polar(1.0L, acos(-1.0L) / k);
    rep(i,k,2*k) rt[i] = R[i] = i&1 ? R[i/2] *
        x : R[i/2];
  }
  vi rev(n);
  rep(i,0,n) rev[i] = (rev[i / 2] | (i & 1) <<
      L) / 2;
  rep(i,0,n) if (i < rev[i]) swap(a[i], a[rev[
      i]]);
  for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k) rep(j
        ,0,k) {
      auto x = (double *)&rt[j+k], y = (double
          *)&a[i+j+k];
      C z(x[0]*y[0] - x[1]*y[1], x[0]*y[1] + x
          [1]*y[0]);
      a[i + j + k] = a[i + j] - z;
      a[i + j] += z;
    }
} // Use vector<C> when complex covolution
vd conv(const vd& a, const vd& b) {
  if (a.empty() || b.empty()) return {};
  vd res(sz(a) + sz(b) - 1);
  int L = 32 - __builtin_clz(sz(res)), n = 1
      << L;
  vector<C> in(n), out(n); // create in2
  copy(all(a), begin(in)); // copy(b) -> in2
  rep(i,0,sz(b)) in[i].imag(b[i]); // skip
  fft(in); // call extra fft for in2
  for (C& x : in) x *= x; // out_i=in_i*in2_i
  rep(i,0,n) out[i] = in[-i & (n - 1)] - conj(
      in[i]); // skip, rather divide by n
  fft(out);
  rep(i,0,sz(res)) res[i] = imag(out[i]) / (4
      * n); // do rounding on out instead
  return res;
}
```

FastFourierTransformMod.h

Description: Higher precision FFT, can be used for convolutions modulo arbitrary integers as long as $N \log_2 N \cdot \text{mod} < 8.6 \cdot 10^{14}$ (in practice 10^{16} or higher). Inputs must be in $[0, \text{mod})$.

Time: $\mathcal{O}(N \log N)$, where $N = |A| + |B|$ (twice as slow as NTT or FFT)

"FastFourierTransform.h" b82773, 22 lines

```
typedef vector<ll> vl;
template<int M> vl convMod(const vl &a, const
    vl &b) {
  if (a.empty() || b.empty()) return {};
  vl res(sz(a) + sz(b) - 1);
  int B=32-__builtin_clz(sz(res)), n=1<<B, cut
      =int(sqrt(M));
  vector<C> L(n), R(n), outs(n), outl(n);
  rep(i,0,sz(a)) L[i] = C((int)a[i] / cut, (
      int)a[i] % cut);
  rep(i,0,sz(b)) R[i] = C((int)b[i] / cut, (
      int)b[i] % cut);
  fft(L), fft(R);
  rep(i,0,n) {
    int j = -i & (n - 1);
    outl[j] = (L[i] + conj(L[j])) * R[i] /
        (2.0 * n);
    outs[j] = (L[i] - conj(L[j])) * R[i] /
        (2.0 * n) / 1i;
  }
  fft(outl), fft(outs);
  rep(i,0,sz(res)) {
    ll av = ll(real(outl[i])+.5), cv = ll(imag
        (outs[i])+.5);
    ll bv = ll(imag(outl[i])+.5) + ll(real(
        outs[i])+.5);
    res[i] = ((av % M * cut + bv) % M * cut +
        cv) % M;
  }
  return res;
}
```

NumberTheoreticTransform.h

Description: ntt(a) computes $\hat{f}(k) = \sum_x a[x]g^{xk}$ for all k, where $g = \text{root}^{(mod-1)/N}$. N must be a power of 2. Useful for convolution modulo specific nice primes of the form $2^a b + 1$, where the convolution result has size at most 2^a. For arbitrary modulo, see FFTMod. `conv(a, b) = c`, where $c[x] = \sum a[i]b[x-i]$. For manual convolution: NTT the inputs, multiply pointwise, divide by n, reverse(start+1, end), NTT back. Inputs must be in [0, mod).

Time: $\mathcal{O}(N \log N)$

"../number-theory/ModPow.h" ced03d, 35 lines

```
const ll mod = (119 << 23) + 1, root = 62; //
    = 998244353
// For p < 2^30 there is also e.g. 5 << 25, 7
    << 26, 479 << 21
// and 483 << 21 (same root). The last two are
    > 10^9.
typedef vector<ll> vl;
void ntt(vl &a) {
  int n = sz(a), L = 31 - __builtin_clz(n);
  static vl rt(2, 1);
  for (static int k = 2, s = 2; k < n; k *= 2,
      s++) {
    rt.resize(n);
    ll z[] = {1, modpow(root, mod >> s)};
    rep(i,k,2*k) rt[i] = rt[i / 2] * z[i & 1]
        % mod;
  }
  vi rev(n);
  rep(i,0,n) rev[i] = (rev[i / 2] | (i & 1) <<
      L) / 2;
  rep(i,0,n) if (i < rev[i]) swap(a[i], a[rev[
      i]]);
  for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k) rep(j
        ,0,k) {
      ll z = rt[j + k] * a[i + j + k] % mod, &
          ai = a[i + j];
      a[i + j + k] = ai - z + (z > ai ? mod :
          0);
      ai += (ai + z >= mod ? z - mod : z);
    }
}
vl conv(const vl &a, const vl &b) {
  if (a.empty() || b.empty()) return {};
  int s = sz(a) + sz(b) - 1, B = 32 -
      __builtin_clz(s),
      n = 1 << B;
  int inv = modpow(n, mod - 2);
  vl L(a), R(b), out(n);
  L.resize(n), R.resize(n);
  ntt(L), ntt(R);
  rep(i,0,n)
    out[-i & (n - 1)] = (ll)L[i] * R[i] % mod
        * inv % mod;
  ntt(out);
  return {out.begin(), out.begin() + s};
}
```

FastSubsetTransform.h

Description: (aka FWHT) Transform to a basis with fast convolutions of the form $c[z] = \sum_{z=x \oplus y} a[x] \cdot b[y]$, where $\oplus$ is one of AND, OR, XOR. The size of a must be a power of two.

Time: $\mathcal{O}(N \log N)$

464cf3, 16 lines

```
void FST(vi& a, bool inv) {
  for (int n = sz(a), step = 1; step < n; step
      *= 2) {
    for (int i = 0; i < n; i += 2 * step) rep(
        j,i,i+step) {
      int &u=a[j], &v=a[j+step]; tie(u, v) =
        inv ? pii(v-u,u) : pii(v,u+v); // AND
        inv ? pii(v,u-v) : pii(u+v,u); // OR
        pii(u+v, u-v);                 // XOR
    }
  }
  if(inv) for(int&x : a) x/=sz(a); //XOR only
}
vi conv(vi a, vi b) {
  FST(a, 0); FST(b, 0);
  rep(i,0,sz(a)) a[i] *= b[i];
  FST(a, 1); return a;
}
```

FastSubsetConvolution.h

Description: $\text{ans}[i] = \sum_{j \subseteq i} f_j g_{i \oplus j}$

Time: $\mathcal{O}\left(n^2 2^n\right)$ or, $\mathcal{O}\left(N \log^2 N\right)$

7571e4, 28 lines

```
int f[N], g[N], fh[LG][N], gh[LG][N], h[LG][N
    ], ans[N];
void conv() {
for (int mask = 0; mask < 1 << n; ++mask) {
 fh[__builtin_popcount(mask)][mask]=f[mask];
 gh[__builtin_popcount(mask)][mask]=g[mask];
}
for (int i = 0; i <= n; ++i) {
 for (int j = 0; j < n; ++j)
  for (int mask = 0; mask < 1 << n; ++mask)
   if (mask & 1 << j) {
    fh[i][mask] += fh[i][mask ^ 1 << j];
    gh[i][mask] += gh[i][mask ^ 1 << j];
   }
}
for (int mask = 0; mask < 1 << n; ++mask) {
 for (int i = 0; i <= n; ++i)
  for (int j = 0; j <= i; ++j)
   h[i][mask]+=fh[j][mask] * gh[i-j][mask];
}
for (int i = 0; i <= n; ++i) {
 for (int j = 0; j < n; ++j)
  for (int mask = 0; mask < 1 << n; ++mask)
   if (mask & 1 << j)
    h[i][mask] -= h[i][mask ^ 1 << j];
}
```

```
for (int mask = 0; mask < 1 << n; ++mask)
 ans[mask]=h[__builtin_popcount(mask)][mask];
}
```

GCDconvolution.h

Description: Computes $c_1, ..., c_n$, where $c_k = \sum_{\gcd(i,j)=k} a_i b_j$. Generate all primes upto n into pr first using sieve.
Time: $\mathcal{O}(N \log\log N)$ bc0c7a, 19 lines

```
void fw_mul_transform (V<ll> &a) {
  int n = sz(a) - 1;
  for (const auto p : pr) {
    if (p > n) break;
    for (int i = n/p; i>0; --i) a[i]+=a[i*p];
} } // A[i] = \sum_{j} a[i * j]
void bw_mul_transform (V<ll> &a) {
  int n = sz(a) - 1;
  for (const auto p : pr) {
    if (p > n) break;
    for (int i=1; i*p <= n; ++i) a[i]-=a[i*p];
} } // From A get a
V<ll>gcd_conv (const V<ll>&a, const V<ll>&b){
  assert(sz(a) == sz(b)); int n = sz(a);
  auto A = a, B = b;
  fw_mul_transform(A); fw_mul_transform(B);
  for (int i = 1; i < n; ++i) A[i] *= B[i];
  bw_mul_transform(A); return A;
}
```

LCMconvolution.h

Description: Computes $c_1, ..., c_n$, where $c_k = \sum_{lcm(i,j)=k} a_i b_j$. Generate all primes upto n into pr first using sieve.
Time: $\mathcal{O}(N \log\log N)$ 1c5704, 19 lines

```
void fw_div_transform (V<ll> &a) {
  int n = sz(a) - 1;
  for (const auto p : pr) {
    if (p > n) break;
    for (int i=1; i*p <= n; ++i) a[i*p]+=a[i];
} } // A[i] = \sum_{d | i} a[d]
void bw_div_transform (V<ll> &a) {
  int n = sz(a) - 1;
  for (const auto p : pr) {
    if (p > n) break;
    for (int i=n/p; i>0; --i) a[i*p]-=a[i];
} } // From A get a
V<ll>lcm_conv (const V<ll>&a, const V<ll>&b){
  assert(sz(a) == sz(b)); int n = sz(a);
  auto A = a, B = b;
  fw_div_transform(A); fw_div_transform(B);
  for (int i = 1; i < n; ++i) A[i] *= B[i];
  bw_div_transform(A); return A;
}
```

4.3 Matrices

Matrix.h

Description: Basic operations on square matrices.
Usage: Matrix<int, 3> A;
A.d = {{{{1,2,3}}, {{4,5,6}}, {{7,8,9}}}};
array<int, 3> vec = {1,2,3};
vec = (A^N) * vec; 6ab5db, 26 lines

```
template<class T, int N> struct Matrix {
  typedef Matrix M;
  array<array<T, N>, N> d{};
  M operator*(const M& m) const {
    M a;
    rep(i,0,N) rep(j,0,N)
      rep(k,0,N) a.d[i][j] += d[i][k]*m.d[k][j
          ];
    return a;
  }
  array<T, N> operator*(const array<T, N>& vec
      ) const {
    array<T, N> ret{};
    rep(i,0,N) rep(j,0,N) ret[i] += d[i][j] *
        vec[j];
    return ret;
  }
  M operator^(ll p) const {
    assert(p >= 0);
    M a, b(*this);
    rep(i,0,N) a.d[i][i] = 1;
    while (p) {
      if (p&1) a = a*b;
      b = b*b;
      p >>= 1;
    }
    return a;
  }
};
```

Determinant.h

Description: Calculates determinant of a matrix. Destroys the matrix.
Time: $\mathcal{O}(N^3)$ bd5cec, 15 lines

```
double det(vector<vector<double>>& a) {
  int n = sz(a); double res = 1;
  rep(i,0,n) {
    int b = i;
    rep(j,i+1,n) if (fabs(a[j][i]) > fabs(a[b
        ][i])) b = j;
    if (i != b) swap(a[i], a[b]), res *= -1;
    res *= a[i][i];
    if (res == 0) return 0;
    rep(j,i+1,n) {
      double v = a[j][i] / a[i][i];
      if (v != 0) rep(k,i+1,n) a[j][k] -= v *
          a[i][k];
    }
  }
  return res;
}
```

IntDeterminant.h

Description: Calculates determinant using modular arithmetics. Modulos can also be removed to get a pure-integer version.
Time: $\mathcal{O}(N^3)$ 3313dc, 18 lines

```
const ll mod = 12345;
ll det(vector<vector<ll>>& a) {
  int n = sz(a); ll ans = 1;
  rep(i,0,n) {
    rep(j,i+1,n) {
      while (a[j][i] != 0) { // gcd step
        ll t = a[i][i] / a[j][i];
        if (t) rep(k,i,n)
          a[i][k] = (a[i][k] - a[j][k] * t) %
              mod;
        swap(a[i], a[j]);
        ans *= -1;
      }
    }
    ans = ans * a[i][i] % mod;
    if (!ans) return 0;
  }
  return (ans + mod) % mod;
}
```

SolveLinear.h

Description: Solves $A * x = b$. If there are multiple solutions, an arbitrary one is returned. Returns rank, or -1 if no solutions. Data in A and b is lost.
Time: $\mathcal{O}(n^2 m)$ 44c9ab, 38 lines

```
typedef vector<double> vd;
const double eps = 1e-12;

int solveLinear(vector<vd>& A, vd& b, vd& x) {
  int n = sz(A), m = sz(x), rank = 0, br, bc;
  if (n) assert(sz(A[0]) == m);
  vi col(m); iota(all(col), 0);

  rep(i,0,n) {
    double v, bv = 0;
    rep(r,i,n) rep(c,i,m)
      if ((v = fabs(A[r][c])) > bv)
        br = r, bc = c, bv = v;
    if (bv <= eps) {
      rep(j,i,n) if (fabs(b[j]) > eps) return
          -1;
      break;
    }
    swap(A[i], A[br]);
    swap(b[i], b[br]);
    swap(col[i], col[bc]);
    rep(j,0,n) swap(A[j][i], A[j][bc]);
    bv = 1/A[i][i];
    rep(j,i+1,n) {
      double fac = A[j][i] * bv;
      b[j] -= fac * b[i];
      rep(k,i+1,m) A[j][k] -= fac*A[i][k];
    }
    rank++;
  }

  x.assign(m, 0);
  for (int i = rank; i--;) {
    b[i] /= A[i][i];
    x[col[i]] = b[i];
    rep(j,0,i) b[j] -= A[j][i] * b[i];
  }
  return rank; // (multiple solutions if rank
      < m)
}
```

SolveLinear2.h

Description: To get all uniquely determined values of x back from SolveLinear, make the following changes:
"SolveLinear.h" 08e495, 7 lines

```
rep(j,0,n) if (j != i) // instead of rep(j,i
    +1,n)
// ... then at the end:
x.assign(m, undefined);
rep(i,0,rank) {
  rep(j,rank,m) if (fabs(A[i][j]) > eps) goto
      fail;
  x[col[i]] = b[i] / A[i][i];
fail:; }
```

SolveLinearBinary.h

Description: Solves $Ax = b$ over $\mathbb{F}_2$. If there are multiple solutions, one is returned arbitrarily. Returns rank, or -1 if no solutions. Destroys A and b.
Time: $\mathcal{O}(n^2 m)$ fa2d7a, 34 lines

```
typedef bitset<1000> bs;

int solveLinear(vector<bs>& A, vi& b, bs& x,
    int m) {
  int n = sz(A), rank = 0, br;
  assert(m <= sz(x));
  vi col(m); iota(all(col), 0);
  rep(i,0,n) {
    for (br=i; br<n; ++br) if (A[br].any())
        break;
    if (br == n) {
      rep(j,i,n) if(b[j]) return -1;
      break;
    }
    int bc = (int)A[br]._Find_next(i-1);
    swap(A[i], A[br]);
    swap(b[i], b[br]);
    swap(col[i], col[bc]);
    rep(j,0,n) if (A[j][i] != A[j][bc]) {
      A[j].flip(i); A[j].flip(bc);
    }
    rep(j,i+1,n) if (A[j][i]) {
      b[j] ^= b[i];
      A[j] ^= A[i];
    }
    rank++;
  }

  x = bs();
  for (int i = rank; i--;) {
    if (!b[i]) continue;
    x[col[i]] = 1;
    rep(j,0,i) b[j] ^= A[j][i];
  }
  return rank; // (multiple solutions if rank
      < m)
}
```

XorBasis.h

Description: Maintain the basis of bit vectors.
Time: $\mathcal{O}(D^2/64)$ per insert 0daa2d, 19 lines

```
const int D = 1000; // use ll if < 64
struct Xor_Basis {
  V<int> who; V<bitset<D>> a;
  Xor_Basis () : who(D, -1) {}
  bool insert (bitset<D> x) {
    for (int i = 0; i < D; ++i)
      if (x[i] && who[i]!=-1) x^=a[who[i]];
    int pivot = -1;
    for (int i = 0; i < D; ++i)
      if (x[i]) { pivot = i; break; }
    if (pivot == -1) return false;
    // ^ null vector detected
    who[pivot] = sz(a);
    for (int i = 0; i < sz(a); ++i)
      if (a[i][pivot] == 1) a[i] ^= x;
    a.push_back(x);
    return true;
  }
};
```

MatrixInverse.h

Description: Invert matrix A. Returns rank; result is stored in A unless singular (rank < n). Can easily be extended to prime moduli; for prime powers, repeatedly set $A^{-1} = A^{-1}(2I - AA^{-1}) \pmod{p^k}$ where A^{-1} starts as the inverse of A mod p, and k is doubled in each step.
Time: $\mathcal{O}(n^3)$

ebfff6, 35 lines

```
int matInv(vector<vector<double>>& A) {
  int n = sz(A); vi col(n);
  vector<vector<double>> tmp(n, vector<double
      >(n));
  rep(i,0,n) tmp[i][i] = 1, col[i] = i;

  rep(i,0,n) {
    int r = i, c = i;
    rep(j,i,n) rep(k,i,n)
      if (fabs(A[j][k]) > fabs(A[r][c]))
        r = j, c = k;
    if (fabs(A[r][c]) < 1e-12) return i;
    A[i].swap(A[r]); tmp[i].swap(tmp[r]);
    rep(j,0,n)
      swap(A[j][i], A[j][c]), swap(tmp[j][i],
          tmp[j][c]);
    swap(col[i], col[c]);
    double v = A[i][i];
    rep(j,i+1,n) {
      double f = A[j][i] / v;
      A[j][i] = 0;
      rep(k,i+1,n) A[j][k] -= f*A[i][k];
      rep(k,0,n) tmp[j][k] -= f*tmp[i][k];
    }
    rep(j,i+1,n) A[i][j] /= v;
    rep(j,0,n) tmp[i][j] /= v;
    A[i][i] = 1;
  }

  for (int i = n-1; i > 0; --i) rep(j,0,i) {
    double v = A[j][i];
    rep(k,0,n) tmp[j][k] -= v*tmp[i][k];
  }

  rep(i,0,n) rep(j,0,n) A[col[i]][col[j]] =
      tmp[i][j];
  return n;
}
```

Tridiagonal.h

Description: x = tridiagonal(d, p, q, b) solves the equation system

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \end{pmatrix} = \begin{pmatrix} d_0 & p_0 & 0 & 0 & \cdots & 0 \\ q_0 & d_1 & p_1 & 0 & \cdots & 0 \\ 0 & q_1 & d_2 & p_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & q_{n-3} & d_{n-2} & p_{n-2} \\ 0 & 0 & \cdots & 0 & q_{n-2} & d_{n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

This is useful for solving problems on the type

$$a_i = b_i a_{i-1} + c_i a_{i+1} + d_i,\ 1 \le i \le n,$$

where a_0, a_{n+1}, b_i, c_i and d_i are known. a can then be obtained from

$$\{a_i\} = \text{tridiagonal}(\{1, -1, -1, ..., -1, 1\}, \{0, c_1, c_2, \ldots, c_n\}, \{b_1, b_2, \ldots, b_n, 0\}, \{a_0, d_1, d_2, \ldots, d_n, a_{n+1}\}).$$

Fails if the solution is not unique.
If $|d_i| > |p_i| + |q_{i-1}|$ for all i, or $|d_i| > |p_{i-1}| + |q_i|$, or the matrix is positive definite, the algorithm is numerically stable and neither `tr` nor the check for `diag[i] == 0` is needed.
Time: $\mathcal{O}(N)$

115ed4, 25 lines

```
typedef double T;
V<T> tridiagonal(V<T> diag, const V<T>& super,
    const V<T>& sub, V<T> b) {
  int n = sz(b); vi tr(n);
  rep(i,0,n-1) {
    if (abs(diag[i]) < 1e-9 * abs(super[i])) {
        // diag[i] == 0
      b[i+1] -= b[i] * diag[i+1] / super[i];
      if (i+2 < n) b[i+2] -= b[i] * sub[i+1] /
          super[i];
      diag[i+1] = sub[i]; tr[++i] = 1;
    } else {
      diag[i+1] -= super[i]*sub[i]/diag[i];
      b[i+1] -= b[i]*sub[i]/diag[i];
    }
  }
  for (int i = n; i--;) {
    if (tr[i]) {
      swap(b[i], b[i-1]);
      diag[i-1] = diag[i];
      b[i] /= super[i-1];
    } else {
      b[i] /= diag[i];
      if (i) b[i-1] -= b[i]*super[i-1];
    }
  }
  return b;
}
```

4.4 Optimization

GoldenSectionSearch.h

Description: Finds the argument minimizing the function f in the interval $[a, b]$ assuming f is unimodal on the interval, i.e. has only one local minimum and no local maximum. The maximum error in the result is *eps*. Works equally well for maximization with a small change in the code. See TernarySearch.h in the Various chapter for a discrete version.
Usage: `double func(double x) { return 4+x+.3*x*x; }`
`double xmin = gss(-1000,1000,func);`
Time: $\mathcal{O}(\log((b-a)/\epsilon))$

31d45b, 14 lines

```
double gss(double a, double b, double (*f)(
    double)) {
  double r = (sqrt(5)-1)/2, eps = 1e-7;
  double x1 = b - r*(b-a), x2 = a + r*(b-a);
  double f1 = f(x1), f2 = f(x2);
  while (b-a > eps)
    if (f1 < f2) { //change to > to find
        maximum
      b = x2; x2 = x1; f2 = f1;
      x1 = b - r*(b-a); f1 = f(x1);
    } else {
      a = x1; x1 = x2; f1 = f2;
      x2 = a + r*(b-a); f2 = f(x2);
    }
  return a;
}
```

HillClimbing.h

Description: Poor man's optimization for unimodal functions.

8eeeaf, 14 lines

```
typedef array<double, 2> P;

template<class F> pair<double, P> hillClimb(P
    start, F f) {
  pair<double, P> cur(f(start), start);
  for (double jmp = 1e9; jmp > 1e-20; jmp /=
      2) {
    rep(j,0,100) rep(dx,-1,2) rep(dy,-1,2) {
      P p = cur.second;
      p[0] += dx*jmp;
      p[1] += dy*jmp;
      cur = min(cur, make_pair(f(p), p));
    }
  }
  return cur;
}
```

Integrate.h

Description: Simple integration of a function over an interval using Simpson's rule. The error should be proportional to h^4, although in practice you will want to verify that the result is stable to desired precision when epsilon changes.

4756fc, 7 lines

```
template<class F>
double quad(double a, double b, F f, const int
    n = 1000) {
  double h = (b - a) / 2 / n, v = f(a) + f(b);
  rep(i,1,n*2)
    v += f(a + i*h) * (i&1 ? 4 : 2);
  return v * h / 3;
}
```

IntegrateAdaptive.h

Description: Fast integration using an adaptive Simpson's rule.
Usage: `double sphereVolume = quad(-1, 1, [](double x) {`
`return quad(-1, 1, [&](double y) {`
`return quad(-1, 1, [&](double z) {`
`return x*x + y*y + z*z < 1; });});});`

92dd79, 15 lines

```
typedef double d;
#define S(a,b) (f(a) + 4*f((a+b) / 2) + f(b))
    * (b-a) / 6

template <class F>
d rec(F& f, d a, d b, d eps, d S) {
  d c = (a + b) / 2;
  d S1 = S(a, c), S2 = S(c, b), T = S1 + S2;
  if (abs(T - S) <= 15 * eps || b - a < 1e-10)
    return T + (T - S) / 15;
  return rec(f, a, c, eps / 2, S1) + rec(f, c,
      b, eps / 2, S2);
}
template<class F>
d quad(d a, d b, F f, d eps = 1e-8) {
  return rec(f, a, b, eps, S(a, b));
}
```

Simplex.h

Description: Solves a general linear maximization problem: maximize $c^T x$ subject to $Ax \le b$, $x \ge 0$. Returns -inf if there is no solution, inf if there are arbitrarily good solutions, or the maximum value of $c^T x$ otherwise. The input vector is set to an optimal x (or in the unbounded case, an arbitrary solution fulfilling the constraints). Numerical stability is not guaranteed. For better performance, define variables such that $x = 0$ is viable.
Usage: `vvd A = {{1,-1}, {-1,1}, {-1,-2}};`
`vd b = {1,1,-4}, c = {-1,-1}, x;`
`T val = LPSolver(A, b, c).solve(x);`
Time: $\mathcal{O}(NM * \#pivots)$, where a pivot may be e.g. an edge relaxation. $\mathcal{O}(2^n)$ in the general case.

aa8530, 68 lines

```
typedef double T; // long double, Rational,
    double + mod<P>...
typedef vector<T> vd;
typedef vector<vd> vvd;

const T eps = 1e-8, inf = 1/.0;
#define MP make_pair
#define ltj(X) if(s == -1 || MP(X[j],N[j]) <
    MP(X[s],N[s])) s=j

struct LPSolver {
  int m, n;
  vi N, B;
  vvd D;

  LPSolver(const vvd& A, const vd& b, const vd
      & c) :
    m(sz(b)), n(sz(c)), N(n+1), B(m), D(m+2,
        vd(n+2)) {
      rep(i,0,m) rep(j,0,n) D[i][j] = A[i][j];
      rep(i,0,m) { B[i] = n+i; D[i][n] = -1; D
          [i][n+1] = b[i];}
      rep(j,0,n) { N[j] = j; D[m][j] = -c[j];
          }
      N[n] = -1; D[m+1][n] = 1;
    }

  void pivot(int r, int s) {
    T *a = D[r].data(), inv = 1 / a[s];
    rep(i,0,m+2) if (i != r && abs(D[i][s]) >
        eps) {
      T *b = D[i].data(), inv2 = b[s] * inv;
      rep(j,0,n+2) b[j] -= a[j] * inv2;
      b[s] = a[s] * inv2;
    }
    rep(j,0,n+2) if (j != s) D[r][j] *= inv;
    rep(i,0,m+2) if (i != r) D[i][s] *= -inv;
    D[r][s] = inv;
    swap(B[r], N[s]);
  }

  bool simplex(int phase) {
    int x = m + phase - 1;
    for (;;) {
      int s = -1;
      rep(j,0,n+1) if (N[j] != -phase) ltj(D[x
          ]);
      if (D[x][s] >= -eps) return true;
      int r = -1;
      rep(i,0,m) {
        if (D[i][s] <= eps) continue;
        if (r == -1 || MP(D[i][n+1] / D[i][s],
            B[i])
                   < MP(D[r][n+1] / D[r][s],
                       B[r])) r = i;
      }
      if (r == -1) return false;
      pivot(r, s);
    }
  }

  T solve(vd &x) {
    int r = 0;
```

```
    rep(i,1,m) if (D[i][n+1] < D[r][n+1]) r =
        i;
    if (D[r][n+1] < -eps) {
      pivot(r, n);
      if (!simplex(2) || D[m+1][n+1] < -eps)
        return -inf;
      rep(i,0,m) if (B[i] == -1) {
        int s = 0;
        rep(j,1,n+1) ltj(D[i]);
        pivot(i, s);
      }
    }
    bool ok = simplex(1); x = vd(n);
    rep(i,0,m) if (B[i] < n) x[B[i]] = D[i][n
        +1];
    return ok ? D[m][n+1] : inf;
  }
};
```

Number theory (5)

5.1 Modular arithmetic

ModInverse.h

Description: Pre-computation of modular inverses. Assumes LIM $\leq$ mod and that mod is a prime.

6f684f, 3 lines

```
const ll mod = 1000000007, LIM = 200000;
ll* inv = new ll[LIM] - 1; inv[1] = 1;
rep(i,2,LIM) inv[i] = mod - (mod / i) * inv[
    mod % i] % mod;
```

ModPow.h

b83e45, 8 lines

```
const ll mod = 1000000007; // faster if const

ll modpow(ll b, ll e) {
  ll ans = 1;
  for (; e; b = b * b % mod, e /= 2)
    if (e & 1) ans = ans * b % mod;
  return ans;
}
```

ModLog.h

Description: Returns the smallest $x > 0$ s.t. $a^x = b \pmod{m}$, or -1 if no such x exists. modLog(a,1,m) can be used to calculate the order of a.

Time: $\mathcal{O}(\sqrt{m})$

c040b8, 11 lines

```
ll modLog(ll a, ll b, ll m) {
  ll n = (ll) sqrt(m) + 1, e = 1, f = 1, j =
      1;
  unordered_map<ll, ll> A;
  while (j <= n && (e = f = e * a % m) != b %
      m)
    A[e * b % m] = j++;
  if (e == b % m) return j;
  if (__gcd(m, e) == __gcd(m, b))
    rep(i,2,n+2) if (A.count(e = e * f % m))
      return n * i - A[e];
  return -1;
}
```

ModSum.h

Description: Sums of mod'ed arithmetic progressions.
`modsum(to, c, k, m)` $= \sum_{i=0}^{\mathrm{to}-1} (ki+c)\%m$. `divsum` is similar but for floored division.

Time: $\log(m)$, with a large constant.

5c5bc5, 16 lines

```
typedef unsigned long long ull;
ull sumsq(ull to) { return to / 2 * ((to-1) |
    1); }

ull divsum(ull to, ull c, ull k, ull m) {
  ull res = k / m * sumsq(to) + c / m * to;
  k %= m; c %= m;
  if (!k) return res;
  ull to2 = (to * k + c) / m;
  return res + (to - 1) * to2 - divsum(to2, m
      -1 - c, m, k);
}

ll modsum(ull to, ll c, ll k, ll m) {
  c = ((c % m) + m) % m;
  k = ((k % m) + m) % m;
  return to * c + k * sumsq(to) - m * divsum(
      to, c, k, m);
}
```

ModMulLL.h

Description: Calculate $a \cdot b \bmod c$ (or $a^b \bmod c$) for $0 \leq a, b \leq c \leq 7.2 \cdot 10^{18}$.

Time: $\mathcal{O}(1)$ for `modmul`, $\mathcal{O}(\log b)$ for `modpow`

bbbd8f, 11 lines

```
typedef unsigned long long ull;
ull modmul(ull a, ull b, ull M) {
  ll ret = a * b - M * ull(1.L / M * a * b);
  return ret + M * (ret < 0) - M * (ret >= (ll
      )M);
}
ull modpow(ull b, ull e, ull mod) {
  ull ans = 1;
  for (; e; b = modmul(b, b, mod), e /= 2)
    if (e & 1) ans = modmul(ans, b, mod);
  return ans;
}
```

ModSqrt.h

Description: Tonelli-Shanks algorithm for modular square roots. Finds x s.t. $x^2 = a \pmod{p}$ ($-x$ gives the other solution).

Time: $\mathcal{O}(\log^2 p)$ worst case, $\mathcal{O}(\log p)$ for most p

"ModPow.h" 19a793, 24 lines

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

5.2 Primality

LinearSieve.h

Description: Can be used to precompute multiplicative functions using $f(px) = f(p)f(x)$ when $p \nmid x$. We compute $f(px) = f(p^{e+1} \cdot x/p^e) = f(p^{e+1})f(x/p^e)$ by multiplicativity (bookkeeping e, the max power of p dividing x where p is the smallest prime dividing x). If $f(px)$ can be computed easily when $p \mid x$ then we can simplify the code.

Time: $\mathcal{O}(n)$

e696bd, 16 lines

```
int func[N],cnt[N]; bool isc[N]; V<int> prime;
void sieve (int n) {
  fill(isc, isc + n, false); func[1] = 1;
  for (int i = 2; i < n; ++i) {
    if (!isc[i]) {
      prime.push_back(i); func[i]=1; cnt[i]=1;
    }
    for (int j = 0; j < prime.size () && i *
        prime[j] < n; ++j) {
      isc[i * prime[j]] = true;
      if (i % prime[j] == 0) {
        func[i * prime[j]] = func[i] / cnt[i]
            * (cnt[i] + 1);
        cnt[i * prime[j]] = cnt[i] + 1; break;
      } else {
        func[i * prime[j]] = func[i] * func[
            prime[j]];
        cnt[i * prime[j]] = 1;
} } } }
```

LinearSieveMobius.h

Description: Linear sieve for computing Möbius function values. mu[i] = 0 if i has squared prime factor, 1 if even number of prime factors, -1 if odd number of prime factors. Also computes smallest prime factors.

Time: $\mathcal{O}(N)$

701180, 20 lines

```
vector<int> lp(N), mu(N), primes;
void pre() {
  mu[1] = 1;
  for (int i = 2; i < N; ++i) {
    if (lp[i] == 0) {
      lp[i] = i;
      mu[i] = -1;
      primes.push_back(i);
    }
    for (auto &j : primes) {
      int k = i * j;
      if (k >= N) break;
      if (lp[i] == j) {
        mu[k] = 0, lp[k] = j;
        break;
      }
      mu[k] = -mu[i], lp[k] = j;
    }
  }
}
```

SegmentedSieve.h

Description: Segmented sieve for finding primes in range [L, R]. First generates all primes up to sqrt(R), then sieves the segment [L, R]. Useful when L and R are large but R-L+1 is manageable.

Time: $\mathcal{O}((R-L+1)loglogR + sqrt(R)loglogsqrt(R))$

91fe31, 21 lines

```
vector<char> segmentedSieve(long long L, long
    long R) {
  // generate all primes up to sqrt(R)
  long long lim = sqrt(R);
  vector<char> mark(lim + 1, false);
  vector<long long> primes;
  for (long long i = 2; i <= lim; ++i) {
    if (!mark[i]) {
      primes.emplace_back(i);
      for (long long j = i * i; j <= lim; j +=
          i)
        mark[j] = true;
    }
  }

  vector<char> isPrime(R - L + 1, true);
  for (long long i : primes)
    for (long long j = max(i * i, (L + i - 1)
        / i * i); j <= R; j += i)
      isPrime[j - L] = false;
  if (L == 1)
    isPrime[0] = false;
  return isPrime;
}
```

phiFunction.h

Description: *Euler's* ϕ function is defined as $\phi(n) :=$ # of positive integers $\leq n$ that are coprime with n. $\phi(1) = 1$, p prime $\Rightarrow \phi(p^k) = (p-1)p^{k-1}$, m, n coprime $\Rightarrow \phi(mn) = \phi(m)\phi(n)$. If $n = p_1^{k_1} p_2^{k_2} ... p_r^{k_r}$ then $\phi(n) = (p_1 - 1)p_1^{k_1 - 1} ... (p_r - 1)p_r^{k_r - 1}$. $\phi(n) = n \cdot \prod_{p|n}(1 - 1/p)$. $\sum_{d|n} \phi(d) = n$, $\sum_{1 \leq k \leq n, \gcd(k,n)=1} k = n\phi(n)/2, n > 1$

Euler's thm: a, n coprime $\Rightarrow a^{\phi(n)} \equiv 1 \pmod{n}$.

Fermat's little thm: p prime $\Rightarrow a^{p-1} \equiv 1 \pmod{p}\ \forall a$.

cf7d6d, 8 lines

```
const int LIM = 5000000;
int phi[LIM];

void calculatePhi() {
  rep(i,0,LIM) phi[i] = i&1 ? i : i/2;
  for (int i = 3; i < LIM; i += 2) if(phi[i]
      == i)
    for (int j = i; j < LIM; j += i) phi[j] -=
        phi[j] / i;
}
```

FastEratosthenes.h

Description: Prime sieve for generating all primes smaller than LIM.

Time: LIM=1e9 $\approx$ 1.5s

6b2912, 20 lines

```
const int LIM = 1e6;
bitset<LIM> isPrime;
vi eratosthenes() {
  const int S = (int)round(sqrt(LIM)), R = LIM
      / 2;
  vi pr = {2}, sieve(S+1); pr.reserve(int(LIM/
      log(LIM)*1.1));
  vector<pii> cp;
  for (int i = 3; i <= S; i += 2) if (!sieve[i
      ]) {
    cp.push_back({i, i * i / 2});
    for (int j = i * i; j <= S; j += 2 * i)
      sieve[j] = 1;
  }
  for (int L = 1; L <= R; L += S) {
```

```
    array<bool, S> block{};
    for (auto &[p, idx] : cp)
      for (int i=idx; i < S+L; idx = (i+=p))
        block[i-L] = 1;
    rep(i,0,min(S, R - L))
      if (!block[i]) pr.push_back((L + i) * 2
          + 1);
  }
  for (int i : pr) isPrime[i] = 1;
  return pr;
}
```

MillerRabin.h

Description: Deterministic Miller-Rabin primality test. Guaranteed to work for numbers up to $7 \cdot 10^{18}$; for larger numbers, use Python and extend A randomly.
Time: 7 times the complexity of $a^b \mod c$.

"ModMulLL.h" 60dcd1, 12 lines

```
bool isPrime(ull n) {
  if (n < 2 || n % 6 % 4 != 1) return (n | 1)
      == 3;
  ull A[] = {2, 325, 9375, 28178, 450775,
      9780504, 1795265022},
      s = __builtin_ctzll(n-1), d = n >> s;
  for (ull a : A) {   // ^ count trailing
      zeroes
    ull p = modpow(a%n, d, n), i = s;
    while (p != 1 && p != n - 1 && a % n && i
        --)
      p = modmul(p, p, n);
    if (p != n-1 && i != s) return 0;
  }
  return 1;
}
```

Factor.h

Description: Pollard-rho randomized factorization algorithm. Returns prime factors of a number, in arbitrary order (e.g. 2299 -> {11, 19, 11}).
Time: $\mathcal{O}\left(n^{1/4}\right)$, less for numbers with small factors.

"ModMulLL.h", "MillerRabin.h" d8d98d, 18 lines

```
ull pollard(ull n) {
  ull x = 0, y = 0, t = 30, prd = 2, i = 1, q;
  auto f = [&](ull x) { return modmul(x, x, n)
      + i; };
  while (t++ % 40 || __gcd(prd, n) == 1) {
    if (x == y) x = ++i, y = f(x);
    if ((q = modmul(prd, max(x,y) - min(x,y),
        n))) prd = q;
    x = f(x), y = f(f(y));
  }
  return __gcd(prd, n);
}
vector<ull> factor(ull n) {
  if (n == 1) return {};
  if (isPrime(n)) return {n};
  ull x = pollard(n);
  auto l = factor(x), r = factor(n / x);
  l.insert(l.end(), all(r));
  return l;
}
```

5.3 Divisibility

euclid.h

Description: Finds two integers x and y, such that $ax + by = \gcd(a, b)$. If you just need gcd, use the built in __gcd instead. If a and b are coprime, then x is the inverse of $a \pmod b$.

33ba8f, 5 lines

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

CRT.h

Description: Chinese Remainder Theorem.
crt(a, m, b, n) computes x such that $x \equiv a \pmod m$, $x \equiv b \pmod n$. If $|a| < m$ and $|b| < n$, x will obey $0 \le x < \text{lcm}(m, n)$. Assumes $mn < 2^{62}$.
Time: $\log(n)$

"euclid.h" 04d93a, 7 lines

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

SPFAndDivisors.h

Description: Smallest prime factor (SPF) and divisor properties. SPF can be used for fast prime factorization in O(log n).
Time: $\mathcal{O}(NloglogN)$ preprocessing, $\mathcal{O}(logN)$ factorization

15611b, 12 lines

```
//smallest prime factor
vector<int> spf(N);
iota(spf.begin(), spf.end(), 0);
for (int i = 2; i < N; ++i) {
  if (spf[i] == i) {
    for (int j = i * i; j < N; j += i)
      if (spf[j] == j)spf[j] = i;
  }
}

//no. of divisors= (e1+1).(e2+1).(e3+1).....(
    ek+1)
//sum of divisors= ((p1^(e1+1)-1)/p1-1).((p2^(
    e2+1)-1)/p2-1)....((pk^(ek+1)-1)/pk-1)
```

5.3.1 Bézout's identity

For $a \neq$, $b \neq 0$, then $d = gcd(a, b)$ is the smallest positive integer for which there are integer solutions to $ax + by = d$. If (x, y) is one solution, then all solutions are given by $(x + kb/d, y - ka/d)$, $k \in \mathbb{Z}$. Find one solution using egcd.

5.4 Fractions

FracBinarySearch.h

Description: Given f and N, finds the smallest fraction $p/q \in [0, 1]$ such that $f(p/q)$ is true, and $p, q \le N$. You may want to throw an exception from f if it finds an exact solution, in which case N can be removed.
Usage: fracBS([](Frac f) { return f.p>=3*f.q; }, 10); // {1,3}
Time: $\mathcal{O}(\log(N))$

27ab3e, 25 lines

```
struct Frac { ll p, q; };

template<class F>
Frac fracBS(F f, ll N) {
  bool dir = 1, A = 1, B = 1;
  Frac lo{0, 1}, hi{1, 1}; // Set hi to 1/0 to
      search (0, N]
  if (f(lo)) return lo;
  assert(f(hi));
  while (A || B) {
    ll adv = 0, step = 1; // move hi if dir,
        else lo
    for (int si = 0; step; (step *= 2) >>= si)
        {
      adv += step;
      Frac mid{lo.p * adv + hi.p, lo.q * adv +
          hi.q};
      if (abs(mid.p) > N || mid.q > N || dir
          == !f(mid)) {
        adv -= step; si = 2;
      }
    }
    hi.p += lo.p * adv;
    hi.q += lo.q * adv;
    dir = !dir;
    swap(lo, hi);
    A = B; B = !!adv;
  }
  return dir ? hi : lo;
}
```

5.5 Mobius Function

$$\mu(n) = \begin{cases} 0 & n \text{ is not square free} \\ 1 & n \text{ has even number of prime factors} \\ -1 & n \text{ has odd number of prime factors} \end{cases}$$

Mobius Inversion:

$$g(n) = \sum_{d|n} f(d) \Leftrightarrow f(n) = \sum_{d|n} \mu(d) g(n/d)$$

Other useful formulas/forms:

$\sum_{d|n} \mu(d) = [n = 1]$, $\phi(n) = \sum_{d|n} \mu(d) \frac{n}{d}$

$g(n) = \sum_{n|d} f(d) \Leftrightarrow f(n) = \sum_{n|d} \mu(\frac{d}{n}) g(d)$

$g(n) = \sum_{1 \le m \le n} f(\lfloor \frac{n}{m} \rfloor) \Leftrightarrow f(n) = \sum_{1 \le m \le n} \mu(m) g(\lfloor \frac{n}{m} \rfloor)$

If f multiplicative,
$\sum_{d|n} \mu(d) f(d) = \prod_{\text{prime } p|n} (1 - f(p))$ and
$\sum_{d|n} \mu^2(d) f(d) = \prod_{\text{prime } p|n} (1 + f(p))$.

If $s_f(n) = \sum_{i=1}^n f(i)$ is a prefix sum of mulitplicative f then $s_{f*g}(n) = \sum_{1 \le xy \le n} f(x) g(y)$.
Then $s_f(n) = \{s_{f*g}(n) - \sum_{d=2}^n s_f(\lfloor n/d \rfloor) g(d)\} / g(1)$ where $f * g(n) = \sum_{d|n} f(d) g(n/d)$ (Dirichlet).

Precompute (linear sieve) $O(n^{2/3})$ first values of s_f for complexity $O(n^{2/3})$.

Useful sums and convolutions: $\epsilon = \mu * \mathbf{1}$, $\text{id} = \phi * \mathbf{1}$, $\text{id} = g * \text{id}_2$, where $\epsilon(n) = [n = 1]$, $\mathbf{1}(n) = 1$, $\text{id}(n) = n$, $\text{id}_k(n) = n^k$,
$g(n) = \sum_{d|n} \mu(d) nd$.
coprime pairs in $[1, n]$ is $\sum_{d=1}^n \mu(d) \lfloor n/d \rfloor^2$. Sum of GCD pairs in $[1, n]$ is $\sum_{d=1}^n \phi(d) \lfloor n/d \rfloor^2$. Sum of LCM pairs in $[1, n]$ is $\sum_{d=1}^n (\frac{\lfloor n/d \rfloor (1 + \lfloor n/d \rfloor)}{2})^2 g(d)$, where g is defined above with $g(p^k) = p^k - p^{k+1}$.

Combinatorial (6)

6.1 Permutations

IntPerm.h

Description: Permutation -> integer conversion. (Not order preserving.) Integer -> permutation can use a lookup table.
Time: $\mathcal{O}(n)$

044568, 6 lines

```
int permToInt(vi& v) {
  int use = 0, i = 0, r = 0;
  for(int x:v) r = r * ++i +
      __builtin_popcount(use & -(1<<x)),
    use |= 1 << x; // (note: minus, not ~!)
  return r;
}
```

multinomial.h

Description: Computes $\binom{v_0 + \cdots + v_{n-1}}{v_0, \ldots, v_{n-1}}$

a0a312, 6 lines

```
ll multinomial(vi& v) {
  ll c = 1, m = v.empty() ? 1 : v[0];
  rep(i,1,sz(v)) rep(j,0,v[i])
    c = c * ++m / (j+1);
  return c;
}
```

PrecomputedNCR.h

Description: Precomputed binomial coefficients with modular arithmetic. Calculates nCr mod MOD for n,r < N using precomputed factorials.
Time: $\mathcal{O}(N)$ preprocessing, $\mathcal{O}(1)$ per query

8757fe, 15 lines

```
int C(int n, int r) {
    if (r > n) return 0;
    return fact[n] * ifact[r] % MOD * ifact[n
        - r] % MOD;
}

void prepare() {
    fact[0] = 1;
    for (int i = 1; i < N; i++) {
        fact[i] = (fact[i - 1] * i) % mod;
    }
    ifact[N - 1] = inv(fact[N - 1]); //inv is
        a function that returns the modular
        inverse of a number
    for (int i = N - 2; i >= 0; i--) {
        ifact[i] = (ifact[i + 1] * (i + 1)) %
            mod;
    }
}
```

Permutations.h

Description: Permutation algorithms
Time: $\mathcal{O}(n!)$

615722, 11 lines

```
void permute(string &s, int l, int r) {
  if (l == r) { cout << s << '\n'; return; }
  for (int i = l; i <= r; ++i) {
    swap(s[l], s[i]);
```

```
    permute(s, l + 1, r);
    swap(s[l], s[i]);
  }
}

// STL: sort(all(v)); do{/*process*/}while(
     next_permutation(all(v)));
// For prev_permutation, start with reverse
     sorted
```

Cycles Let $g_S(n)$ be the number of n-permutations whose cycle lengths all belong to the set S. Then $\sum_{n\geq 0} g_S(n)\frac{x^n}{n!} = \exp\left(\sum_{n\in S}\frac{x^n}{n}\right)$

Derangements Permutations of a set such that none of the elements appear in their original position. $D(n) = (n-1)(D(n-1)+D(n-2)) = nD(n-1)+(-1)^n = \left\lfloor \frac{n!}{e} \right\rceil$

Burnside's Lemma Given a group G of symmetries and a set X, the number of elements of X *up to symmetry* equals $\frac{1}{|G|}\sum_{g\in G}|X^g|$, where X^g are the elements fixed by g ($g.x = x$). If $f(n)$ counts "configurations" (of some sort) of length n, we can ignore rotational symmetry using $G = \mathbb{Z}_n$ to get $g(n) = \frac{1}{n}\sum_{k=0}^{n-1} f(\gcd(n,k)) = \frac{1}{n}\sum_{k|n} f(k)\phi(n/k)$.

Partition function Number of ways of writing n as a sum of positive integers, disregarding the order of the summands. $p(0) = 1$, $p(n) = \sum_{k\in\mathbb{Z}\setminus\{0\}}(-1)^{k+1}p(n-k(3k-1)/2)$.
$p(n) \sim 0.145/n \cdot \exp(2.56\sqrt{n})$
First few values: 1, 1, 2, 3, 5, 7, 11, 15, 22, 30.
$p(20) = 627, p(50) \approx 2e5, p(100) \approx 2e8$.

Lucas' Theorem: Let n, m be non-negative integers and p a prime. Write $n = n_kp^k + ... + n_1p + n_0$ and $m = m_kp^k + ... + m_1p + m_0$. Then $\binom{n}{m} \equiv \prod_{i=0}^{k}\binom{n_i}{m_i} \pmod p$.

Bernoulli numbers EGF of Bernoulli numbers is $B(t) = \frac{t}{e^t-1}$ (FFT-able). $\sum\frac{B_i}{i!}x^i = \frac{x}{1-e^{-x}}$.
$B[0,\ldots] = [1, -\frac{1}{2}, \frac{1}{6}, 0, -\frac{1}{30}, 0, \frac{1}{42}, \ldots]$.
Sums of powers:

$$\sum_{i=1}^{n} n^m = \frac{1}{m+1}\sum_{k=0}^{m}\binom{m+1}{k}B_k\cdot(n+1)^{m+1-k}$$

Stirling numbers of the first kind Number of permutations on n items with k cycles.
$c(n,k) = c(n-1,k-1) + (n-1)c(n-1,k)$,
$c(0,0) = 1.\sum_{k=0}^{n} c(n,k)x^k = x(x+1)\ldots(x+n-1)$
$c(8,k) =$
$8, 0, 5040, 13068, 13132, 6769, 1960, 322, 28, 1$
$c(n,2) =$
$0, 0, 1, 3, 11, 50, 274, 1764, 13068, 109584, ..$

Stirling numbers of the second kind Partitions of n distinct elements into exactly k non-empty subsets. $S(n,k) = S(n-1,k-1) + kS(n-1,k)$.
$S(n,1) = S(n,n) = 1$.
$S(n,k) = \frac{1}{k!}\sum_{j=0}^{k}(-1)^{k-j}\binom{k}{j}j^n$.

Eulerian numbers Number of n-permutations with exactly k rises (positions i with $p_i > p_{i-1}$).
$E(n,k) = (n-k)E(n-1,k-1) + (k+1)E(n-1,k)$.
$E(n,0) = E(n,n-1) = 1$.
$E(n,k) = \sum_{j=0}^{k}(-1)^j\binom{n+1}{j}(k+1-j)^n$.

Bell numbers Total number of partitions of n distinct elements. $B(n) =$
$1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, \ldots$.
$B(3) = 5 = \{a|b|c, a|bc, b|ac, c|ab, abc\}$. For p prime,
$B(p^m + n) \equiv mB(n) + B(n+1) \pmod p$.

Catalan numbers
$C_n = \frac{1}{n+1}\binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{(n+1)!n!}$
$C_0 = 1,\ C_{n+1} = \frac{2(2n+1)}{n+2}C_n,\ C_{n+1} = \sum C_iC_{n-i}$
$C_n = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \ldots$
- UR path from $(0,0)$ to (n,n) below $y = x$.
- strings with n pairs of parenthesis, correctly nested.
- binary trees with with $n+1$ leaves (0 or 2 children).
- ordered trees with $n+1$ vertices.
- ways a convex polygon with $n+2$ sides can be cut into triangles by connecting vertices with straight lines.
- permutations of $[n]$ with no 3-term increasing subseq.

Labeled unrooted trees: # on n vertices: n^{n-2}
on k existing trees of size n_i: $n_1n_2\cdots n_kn^{k-2}$
with degrees d_i: $(n-2)!/((d_1-1)!\cdots(d_n-1)!)$
ways to connect k components with $k-1$ edges: $s_1\cdots s_k\cdot n^{k-2}$

Number of Spanning Trees Create an $N\times N$ matrix `mat`, and for each edge $a\to b\in G$, do `mat[a][b]--`, `mat[b][b]++` (and `mat[b][a]--`, `mat[a][a]++` if G is undirected). Remove the ith row and column and take the determinant; this yields the number of directed spanning trees rooted at i (if G is undirected, remove any row/column).

Erdős–Gallai theorem A simple graph with node degrees $d_1 \geq \cdots \geq d_n$ exists iff $d_1 + \cdots + d_n$ is even and for every $k = 1\ldots n$,

$$\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n}\min(d_i, k).$$

Sprague-Grundy Theorem: Viewing the game as a DAG, where a player moves from one node v to

Graph (7)

7.1 Shortest Paths

BellmanFord.h

Description: Calculates shortest paths from s in a graph that might have negative edge weights. Unreachable nodes get dist = inf; nodes reachable through negative-weight cycles get dist = -inf. Assumes $V^2\max|w_i| < \sim 2^{63}$.
Time: $\mathcal{O}(VE)$

830a8f, 23 lines

```
const ll inf = LLONG_MAX;
struct Ed { int a, b, w, s() { return a < b ?
     a : -a; }};
struct Node { ll dist = inf; int prev = -1; };

void bellmanFord(vector<Node>& nodes, vector<
     Ed>& eds, int s) {
  nodes[s].dist = 0;
  sort(all(eds), [](Ed a, Ed b) { return a.s()
       < b.s(); });

  int lim = sz(nodes) / 2 + 2; // /3+100 with
       shuffled vertices
  rep(i,0,lim) for (Ed ed : eds) {
    Node cur = nodes[ed.a], &dest = nodes[ed.b
         ];
    if (abs(cur.dist) == inf) continue;
    ll d = cur.dist + ed.w;
    if (d < dest.dist) {
      dest.prev = ed.a;
      dest.dist = (i < lim-1 ? d : -inf);
    }
  }
  rep(i,0,lim) for (Ed e : eds) {
    if (nodes[e.a].dist == -inf)
      nodes[e.b].dist = -inf;
  }
}
```

FloydWarshall.h

Description: Calculates all-pairs shortest path in a directed graph that might have negative edge weights. Input is an distance matrix m, where $m[i][j] =$ `inf` if i and j are not adjacent. As output, $m[i][j]$ is set to the shortest distance between i and j, `inf` if no path, or `-inf` if the path goes through a negative-weight cycle.
Time: $\mathcal{O}(N^3)$

531245, 12 lines

```
const ll inf = 1LL << 62;
void floydWarshall(vector<vector<ll>>& m) {
  int n = sz(m);
  rep(i,0,n) m[i][i] = min(m[i][i], 0LL);
  rep(k,0,n) rep(i,0,n) rep(j,0,n)
    if (m[i][k] != inf && m[k][j] != inf) {
      auto newDist = max(m[i][k] + m[k][j], -
           inf);
      m[i][j] = min(m[i][j], newDist);
    }
  rep(k,0,n) if (m[k][k] < 0) rep(i,0,n) rep(j
       ,0,n)
    if (m[i][k] != inf && m[k][j] != inf) m[i
         ][j] = -inf;
}
```

TopoSort.h

Description: Topological sorting. Given is an oriented graph. Output is an ordering of vertices, such that there are edges only from left to right. If there are cycles, the returned list will have size smaller than n – nodes reachable from cycles will not be returned.
Time: $\mathcal{O}(|V| + |E|)$

d678d8, 8 lines

```
vi topoSort(const vector<vi>& gr) {
  vi indeg(sz(gr)), q;
  for (auto& li : gr) for (int x : li) indeg[x
       ]++;
  rep(i,0,sz(gr)) if (indeg[i] == 0) q.
       push_back(i);
  rep(j,0,sz(q)) for (int x : gr[q[j]])
    if (--indeg[x] == 0) q.push_back(x);
  return q;
}
```

Johnson.h

Description: APSP on weighted directed graphs with no negative cycles. Add a dummy node q connected by 0-weighted edge to each other node. Then run Bellman from q to find minimum weight $h(v)$ of a path $q \rightsquigarrow v$ (terminate if negative cycle found). Next, reweight the original graph: $\forall u \to v$ with weight $w(u,v)$, assign new weight $w(u,v) + h(u) - h(v)$. Now $D(u,v) =$ Dijkstra$(u,v) + h(v) - h(u)$.
Time: $\mathcal{O}(Bellman) + \mathcal{O}(V) * \mathcal{O}(Dijkstra)$

7.2 Network flow

Dinic.h

Description: Flow algorithm. with complexity
Time: $\mathcal{O}(VE\log U)$ where $U = \max|\text{cap}|$. $\mathcal{O}\left(\min(E^{1/2}, V^{2/3})E\right)$ if $U = 1$; $\mathcal{O}\left(\sqrt{V}E\right)$ for bipartite matching.

d7f0f1, 40 lines

```
struct Dinic {
  struct Edge {
    int to, rev; ll c, oc;
    ll flow() { return max(oc - c, 0LL); }
  }; // .flow() gives actual flow
  vi lvl, ptr, q;
  vector<vector<Edge>> adj;
  Dinic(int n) : lvl(n),ptr(n),q(n),adj(n) {}
  void addEdge(int a,int b,ll c,ll rcap=0) {
    adj[a].push_back({b, sz(adj[b]), c, c});
    adj[b].push_back({a, sz(adj[a]) - 1, rcap,
         rcap});
  } // rcap = c on bidirectional
  ll dfs(int v, int t, ll f) {
    if (v == t || !f) return f;
    for (int& i = ptr[v]; i<sz(adj[v]); i++){
      Edge& e = adj[v][i];
      if (lvl[e.to] == lvl[v] + 1)
        if (ll p = dfs(e.to, t, min(f,e.c))){
          e.c -= p, adj[e.to][e.rev].c += p;
          return p;
        }
    } return 0;
  }
  ll calc(int s, int t) {
    ll flow = 0; q[0] = s;
    rep(L,0,31) do { // 'int L=30' maybe
         faster for random data
      lvl = ptr = vi(sz(q));
      int qi = 0, qe = lvl[s] = 1;
      while (qi < qe && !lvl[t]) {
        int v = q[qi++];
```

```
      for (Edge e : adj[v])
        if (!lvl[e.to] && e.c >> (30 - L))
          q[qe++]=e.to, lvl[e.to]=lvl[v]+1;
      }
      while (ll p=dfs(s,t,LLONG_MAX)) flow+=p;
    } while (lvl[t]);
    return flow;
  }
  bool leftOfMinCut(int a) {return lvl[a]!=0;}
};
```

PushRelabel.h

Description: Push-relabel using the highest label selection rule and the gap heuristic. Quite fast in practice. To obtain the actual flow, look at positive values only.
Time: $\mathcal{O}\left(V^2\sqrt{E}\right)$

2fd373, 40 lines

```
struct PushRelabel {
  struct Edge { int dest, back; ll f, c; };
  vector<vector<Edge>> g; vector<ll> ec;
  vector<Edge*> cur; vector<vi> hs; vi H;
  PushRelabel(int n) : g(n), ec(n), cur(n), hs
      (2*n), H(n) {}
  void addEdge(int s,int t,ll cap,ll rcap=0) {
    if (s == t) return;
    g[s].push_back({t, sz(g[t]), 0, cap});
    g[t].push_back({s, sz(g[s])-1, 0, rcap});
  } // rcap = cap on bidirectional
  void addFlow(Edge& e, ll f) {
    Edge &back = g[e.dest][e.back];
    if (!ec[e.dest] && f) hs[H[e.dest]].
        push_back(e.dest);
    e.f += f; e.c -= f; ec[e.dest] += f;
    back.f-=f; back.c += f; ec[back.dest]-=f;
  }
  ll calc(int s, int t) {
    int v = sz(g); H[s] = v; ec[t] = 1;
    vi co(2*v); co[0] = v-1;
    rep(i,0,v) cur[i] = g[i].data();
    for (Edge& e : g[s]) addFlow(e, e.c);

    for (int hi = 0;;) {
      while (hs[hi].empty()) if (!hi--) return
          -ec[s];
      int u=hs[hi].back(); hs[hi].pop_back();
      while (ec[u] > 0) { // discharge u
        if (cur[u] == g[u].data()+sz(g[u])) {
          H[u] = 1e9;
          for (Edge& e : g[u]) if (e.c && H[u]
              > H[e.dest]+1)
            H[u] = H[e.dest]+1, cur[u] = &e;
          if (++co[H[u]], !--co[hi] && hi < v)
            rep(i,0,v) if (hi<H[i] && H[i]<v)
              --co[H[i]], H[i] = v + 1;
          hi = H[u];
        } else if (cur[u]->c && H[u] == H[cur[
            u]->dest]+1)
          addFlow(*cur[u],min(ec[u], cur[u]->c
              ));
        else ++cur[u];
  } } }
 bool leftOfMinCut(int a){return H[a]>=sz(g);}
};
```

MinCostMaxFlow.h

Description: Min-cost max-flow. If costs can be negative, call setpi before maxflow, but note that negative cost cycles are not supported. To obtain the actual flow, look at positive values only.
Time: $\mathcal{O}(FE\log(V))$ where F is max flow. $\mathcal{O}(VE)$ for setpi.

15e3e9, 62 lines

```
#include <bits/extc++.h>
const ll INF = numeric_limits<ll>::max() / 4;

struct MCMF {
  struct edge {
    int from, to, rev;  ll cap, cost, flow;
  };
  int N; V<V<edge>> ed; vi seen;
  V<ll> dist, pi; V<edge*> par;
  MCMF(int N) : N(N), ed(N), seen(N),
    dist(N), pi(N), par(N) {}
  void addEdge(int from, int to, ll cap, ll
      cost) {
    if (from == to) return;
    ed[from].push_back(edge{ from,to,sz(ed[to
        ]),cap,cost,0 });
    ed[to].push_back(edge{ to,from,sz(ed[from
        ])-1,0,-cost,0 });
  }
  void path(int s) {
    fill(all(seen), 0); fill(all(dist), INF);
    dist[s] = 0; ll di;
    __gnu_pbds::priority_queue<pair<ll,int>>q;
    V<decltype(q)::point_iterator> its(N);
    q.push({ 0, s });
    while (!q.empty()) {
      s = q.top().second; q.pop();
      seen[s] = 1; di = dist[s] + pi[s];
      for (edge& e : ed[s]) if (!seen[e.to]) {
        ll val = di - pi[e.to] + e.cost;
        if (e.cap-e.flow>0 && val<dist[e.to]){
          dist[e.to] = val;  par[e.to] = &e;
          if (its[e.to] == q.end())
            its[e.to] = q.push({ -dist[e.to],
                e.to });
          else q.modify(its[e.to], { -dist[e.
              to], e.to });
    } } }
    rep(i,0,N) pi[i] = min(pi[i] + dist[i],
        INF);
  } // Hash without maxflow() setpi() = 061a45
  pair<ll, ll> maxflow(int s, int t) {
    ll totflow = 0, totcost = 0;
    while (path(s), seen[t]) {
      ll fl = INF;
      for (edge* x = par[t]; x; x = par[x->
          from])
        fl = min(fl, x->cap - x->flow);

      totflow += fl;
      for (edge* x = par[t]; x; x = par[x->
          from]) {
        x->flow += fl;
        ed[x->to][x->rev].flow -= fl;
      }
    }
    rep(i,0,N) for(edge& e : ed[i]) totcost +=
        e.cost * e.flow;
    return {totflow, totcost/2};
  } // Hash without setpi() = d04eb5
  void setpi(int s) {
    fill(all(pi), INF); pi[s] = 0;
    int it = N, ch = 1; ll v;
    while (ch-- && it--)
      rep(i,0,N) if (pi[i] != INF)
        for (edge& e : ed[i]) if (e.cap)
          if ((v = pi[i] + e.cost) < pi[e.to])
            pi[e.to] = v, ch = 1;
    assert(it >= 0); // negative cost cycle
  }
};
```

MinCut.h

Description: After running max-flow, the left side of a min-cut from s to t is given by all vertices reachable from s, only traversing edges with positive residual capacity.

GlobalMinCut.h

Description: Find a global minimum cut in an undirected graph, as represented by an adjacency matrix.
Time: $\mathcal{O}(V^3)$

8b0e19, 21 lines

```
pair<int, vi> globalMinCut(vector<vi> mat) {
  pair<int, vi> best = {INT_MAX, {}};
  int n = sz(mat);
  vector<vi> co(n);
  rep(i,0,n) co[i] = {i};
  rep(ph,1,n) {
    vi w = mat[0];
    size_t s = 0, t = 0;
    rep(it,0,n-ph) { // O(V^2) -> O(E log V)
        with prio. queue
      w[t] = INT_MIN;
      s=t, t=max_element(all(w)) - w.begin();
      rep(i,0,n) w[i] += mat[t][i];
    }
    best = min(best, {w[t]-mat[t][t], co[t]});
    co[s].insert(co[s].end(), all(co[t]));
    rep(i,0,n) mat[s][i] += mat[t][i];
    rep(i,0,n) mat[i][s] = mat[s][i];
    mat[0][t] = INT_MIN;
  }
  return best;
}
```

GomoryHu.h

Description: Given a list of edges representing an undirected flow graph, returns edges of the Gomory-Hu tree. The max flow between any pair of vertices is given by minimum edge weight along the Gomory-Hu tree path.
Time: $\mathcal{O}(V)$ Flow Computations

"PushRelabel.h" 0418b3, 12 lines

```
typedef array<ll, 3> Edge;
vector<Edge> gomoryHu(int N,vector<Edge> ed) {
  vector<Edge> tree; vi par(N);
  rep(i,1,N) {
    PushRelabel D(N); // Dinic also works
    for (Edge t : ed) D.addEdge(t[0], t[1], t
        [2], t[2]);
    tree.push_back({i, par[i], D.calc(i, par[i
        ])});
    rep(j,i+1,N)
      if (par[j]==par[i] && D.leftOfMinCut(j))
        par[j] = i;
  } return tree;
}
```

7.3 Matching

HopcroftKarp.h

Description: Fast bipartite matching algorithm. Graph g should be a list of neighbors of the left partition, and $btoa$ should be a vector full of -1's of the same size as the right partition. Returns the size of the matching. $btoa[i]$ will be the match for vertex i on the right side, or -1 if it's not matched.
Usage: vi btoa(m, -1); hopcroftKarp(g, btoa);
Time: $\mathcal{O}\left(\sqrt{V}E\right)$

728df7, 34 lines

```
bool dfs(int a, int L, vector<vi>& g, vi& btoa
    , vi& A, vi& B) {
  if (A[a] != L) return 0;
  A[a] = -1;
  for (int b : g[a]) if (B[b] == L + 1) {
    B[b] = 0;
    if (btoa[b] == -1 || dfs(btoa[b], L + 1, g
        , btoa, A, B))
      return btoa[b] = a, 1;
  } return 0;
}
int hopcroftKarp(vector<vi>& g, vi& btoa) {
  int res = 0;
  vi A(g.size()), B(btoa.size()), cur, next;
  for (;;) {
    fill(all(A), 0); fill(all(B), 0);
    cur.clear();
    for (int a : btoa) if(a != -1) A[a] = -1;
    rep(a,0,sz(g))if(A[a]==0)cur.push_back(a);
    for (int lay = 1;; lay++) {
      bool islast = 0; next.clear();
      for (int a : cur) for (int b : g[a]) {
        if (btoa[b] == -1)
          B[b]=lay, islast = 1;
        else if (btoa[b] != a && !B[b]) {
          B[b]=lay; next.push_back(btoa[b]);
        }
      }
      if (islast) break;
      if (next.empty()) return res;
      for (int a : next) A[a] = lay;
      cur.swap(next);
    }
    rep(a,0,sz(g))
      res += dfs(a, 0, g, btoa, A, B);
} }
```

DFSMatching.h

Description: Simple bipartite matching algorithm. Graph g should be a list of neighbors of the left partition, and $btoa$ should be a vector full of -1's of the same size as the right partition. Returns the size of the matching. $btoa[i]$ will be the match for vertex i on the right side, or -1 if it's not matched.
Usage: vi btoa(m, -1); dfsMatching(g, btoa);
Time: $\mathcal{O}(VE)$

522b98, 22 lines

```
bool find(int j, vector<vi>& g, vi& btoa, vi&
    vis) {
  if (btoa[j] == -1) return 1;
  vis[j] = 1; int di = btoa[j];
  for (int e : g[di])
    if (!vis[e] && find(e, g, btoa, vis)) {
      btoa[e] = di;
      return 1;
    }
  return 0;
}
int dfsMatching(vector<vi>& g, vi& btoa) {
  vi vis;
```

```
  rep(i,0,sz(g)) {
    vis.assign(sz(btoa), 0);
    for (int j : g[i])
      if (find(j, g, btoa, vis)) {
        btoa[j] = i;
        break;
      }
  }
  return sz(btoa) - (int)count(all(btoa), -1);
}
```

MinimumVertexCover.h

Description: Finds a minimum vertex cover in a bipartite graph. The size is the same as the size of a maximum matching, and the complement is a maximum independent set.

"HopcroftKarp.h" 23c286, 18 lines

```
vi cover(vector<vi>& g, int n, int m) {
  vi match(m, -1);
  int res = hopcroftKarp(g, match);
  vector<bool> lfound(n, true), seen(m);
  for (int it:match) if(it!=-1) lfound[it]=0;
  vi q, cover;
  rep(i,0,n) if (lfound[i]) q.push_back(i);
  while (!q.empty()) {
    int i = q.back(); q.pop_back();
    lfound[i] = 1;
    for(int e:g[i]) if(!seen[e]&&match[e]!=-1)
      { seen[e] = 1; q.push_back(match[e]); }
  }
  rep(i,0,n) if(!lfound[i])cover.push_back(i);
  rep(i,0,m) if(seen[i]) cover.push_back(n+i);
  assert(sz(cover) == res);
  return cover;
}
```

WeightedMatching.h

Description: Given a weighted bipartite graph, matches every node on the left with a node on the right such that no nodes are in two matchings and the sum of the edge weights is minimal. Takes cost[N][M], where cost[i][j] = cost for L[i] to be matched with R[j] and returns (min cost, match), where L[i] is matched with R[match[i]]. Negate costs for max cost. Requires $N \le M$.
Time: $\mathcal{O}(N^2M)$

1e0fe9, 34 lines

```
pair<int, vi> hungarian(const vector<vi> &a) {
  if (a.empty()) return {0, {}};
  int n = sz(a) + 1, m = sz(a[0]) + 1;
  vi u(n), v(m), p(m), ans(n - 1);
  rep(i,1,n) {
    p[0] = i;
    int j0 = 0; // add "dummy" worker 0
    vi dist(m, INT_MAX), pre(m, -1);
    vector<bool> done(m + 1);
    do { // dijkstra
      done[j0] = true;
      int i0 = p[j0], j1, delta = INT_MAX;
      rep(j,1,m) if (!done[j]) {
        auto cur = a[i0-1][j-1] - u[i0]-v[j];
        if (cur < dist[j])
          dist[j] = cur, pre[j] = j0;
        if (dist[j] < delta)
          delta = dist[j], j1 = j;
      }
      rep(j,0,m) {
        if (done[j])
          u[p[j]] += delta, v[j] -= delta;
        else dist[j] -= delta;
      }
      j0 = j1;
    } while (p[j0]);
    while (j0) { // update alternating path
      int j1 = pre[j0];
      p[j0] = p[j1], j0 = j1;
    }
  }
  rep(j,1,m) if (p[j]) ans[p[j] - 1] = j - 1;
  return {-v[0], ans}; // min cost
}
```

GeneralMatching.h

Description: Matching for general graphs. Fails with probability N/mod.
Time: $\mathcal{O}(N^3)$

"../numerical/MatrixInverse-mod.h" 9eead0, 37 lines

```
V<pii> generalMatching(int N, V<pii>& ed) {
  V<V<ll>> mat(N, V<ll>(N)), A;
  for (pii pa : ed) {
    int a=pa.first,b=pa.second,r=rand()%mod;
    mat[a][b] = r, mat[b][a] = (mod-r) % mod;
  }
  int r = matInv(A = mat), M = 2*N-r, fi, fj;
  assert(r % 2 == 0);
  if (M != N) do {
    mat.resize(M, vector<ll>(M));
    rep(i,0,N) {
      mat[i].resize(M);
      rep(j,N,M) {
        int r = rand() % mod;
        mat[i][j]=r, mat[j][i]=(mod-r)%mod;
      }
    }
  } while (matInv(A = mat) != M);
  vi has(M, 1); vector<pii> ret;
  rep(it,0,M/2) {
    rep(i,0,M) if (has[i])
      rep(j,i+1,M) if (A[i][j] && mat[i][j]) {
        fi = i; fj = j; goto done;
    } assert(0); done:
    if (fj < N) ret.emplace_back(fi, fj);
    has[fi] = has[fj] = 0;
    rep(sw,0,2) {
      ll a = modpow(A[fi][fj], mod-2);
      rep(i,0,M) if (has[i] && A[i][fj]) {
        ll b = A[i][fj] * a % mod;
        rep(j,0,M) A[i][j] = (A[i][j] - A[fi][
            j] * b) % mod;
      }
      swap(fi,fj);
    }
  }
  return ret;
}
```

7.4 DFS algorithms

SCC.h

Description: Finds strongly connected components in a directed graph. If vertices u, v belong to the same component, we can reach u from v and vice versa.
Usage: `scc(graph, [&](vi& v) { ... })` visits all components
in reverse topological order. `comp[i]` holds the component
index of a node (a component only has edges to components with
lower index). `ncomps` will contain the number of components.
Time: $\mathcal{O}(E + V)$

76b5c9, 21 lines

```
vi val, comp, z, cont;
int Time, ncomps; template<class G, class F>
int dfs(int j, G& g, F& f) {
  int low=val[j]=++Time, x; z.push_back(j);
  for (auto e : g[j]) if (comp[e] < 0)
    low = min(low, val[e] ? : dfs(e,g,f));
  if (low == val[j]) {
    do {
      x = z.back(); z.pop_back();
      comp[x] = ncomps; cont.push_back(x);
    } while (x != j);
    f(cont); cont.clear(); ncomps++;
  }
  return val[j] = low;
}
template<class G,class F> void scc(G& g, F f){
  int n = sz(g);
  val.assign(n, 0); comp.assign(n, -1);
  Time = ncomps = 0;
  rep(i,0,n) if (comp[i] < 0) dfs(i, g, f);
}
```

SCC-kosaraju.h

Description: Kosaraju's algorithm for finding strongly connected components. Step 1: Perform DFS on original graph, record finishing order. Step 2: Create transpose graph (reverse all edges). Step 3: Perform DFS on transpose graph in reverse finishing order. Each DFS tree in step 3 is one strongly connected component.
Time: $\mathcal{O}(V + E)$

BiconnectedComponents.h

Description: Finds all biconnected components in an undirected graph, and runs a callback for the edges in each. In a biconnected component there are at least two distinct paths between any two nodes. Note that a node can be in several components. An edge which is not in a component is a bridge. A node is a cut point if (1) Exists in multiple bccs, or (2) Endpoint of a bridge with degree > 1 (self loops don't count as degree).
Usage: `int eid = 0; g.resize(N);`
`for each edge (a,b) {`
`g[a].emplace_back(b, eid);`
`g[b].emplace_back(a, eid++); }`
`bicomps([&](const vi& edgelist) {...});`
Time: $\mathcal{O}(E + V)$

c5905f, 26 lines

```
vector<vector<pii>> g;
vi num, st; int Time;
template<class F>
int dfs(int at, int par, F& f) {
  int me = num[at] = ++Time, top = me;
  for (auto [y, e] : g[at]) if (e != par) {
    if (num[y]) {
      top = min(top, num[y]);
      if (num[y] < me) st.push_back(e);
    } else {
      int si = sz(st), up = dfs(y, e, f);
      top = min(top, up);
      if (up == me) {
        st.push_back(e);
        f(vi(st.begin() + si, st.end()));
        st.resize(si);
      }
      else if (up < me) st.push_back(e);
      else { /* e is a bridge */ }
    }
  }
  return top;
}
template<class F> void bicomps(F f) {
  Time = 0; num.assign(sz(g), 0);
  rep(i,0,sz(g)) if (!num[i]) dfs(i, -1, f);
}
```

BlockCutTree.h

Description: Finds the block-cut tree of a bidirectional graph. Tree nodes are either cut points or a block. All edges are between a block and a cut point. Combining all nodes in a block with its neighbor cut points give the whole BCC.
Usage: `art[i] = true if cut point. Cut-points are relabeled within [1,ncut]. Higher labels are for blocks. Resets: art, g[1,n], tree[1,ptr], st, comp[1,cur], ptr, cur, in;`

c5cc0e, 36 lines

```
bitset <N> art;
vector <int> g[N], tree[N], st, comp[N];
int n, m, ptr, cur, ncut, in[N],low[N],id[N];
void dfs (int u, int from = -1) {
  in[u] = low[u] = ++ptr; st.emplace_back(u);
  for (int v : g[u]) if (v ^ from) {
    if (!in[v]) {
      dfs(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] >= in[u]) {
        art[u] = in[u] > 1 or in[v] > 2;
        comp[++cur].emplace_back(u);
        while (comp[cur].back() ^ v) {
          comp[cur].emplace_back(st.back());
          st.pop_back();
        }
      }
    } else { low[u] = min(low[u], in[v]); }
} }
void buildTree() {
  ptr = 0;
  for (int i = 1; i <= n; ++i) {
    if (art[i]) id[i] = ++ptr;
  } ncut = ptr;
  for (int i = 1; i <= cur; ++i) {
    int x = ++ptr;
    for (int u : comp[i]) {
      if (art[u]) {
        tree[x].emplace_back(id[u]);
        tree[id[u]].emplace_back(x);
      } else { id[u] = x; }
} } }
int main() {
  for (int i = 1; i <= n; ++i)
    if (!in[i]) dfs(i);
  buildTree();
}
```

2sat.h

Description: Calculates a valid assignment to boolean variables a, b, c,... to a 2-SAT problem, so that an expression of the type $(a||b)\&\&(!a||c)\&\&(d||!b)\&\&...$ becomes true, or reports that it is unsatisfiable. Negated variables are represented by bit-inversions (~x).

Usage: TwoSat ts(number of boolean variables);
ts.either(0, ~3); // Var 0 is true or var 3 is false
ts.setValue(2); // Var 2 is true
ts.atMostOne({0,~1,2}); // <= 1 of vars 0, ~1 and 2 are true
ts.solve(); // Returns true iff it is solvable
ts.values[0..N-1] holds the assigned values to the vars

Time: $\mathcal{O}(N+E)$, where N is the number of boolean variables, and E is the number of clauses.

5f9706, 56 lines

```cpp
struct TwoSat {
  int N;
  vector<vi> gr;
  vi values; // 0 = false, 1 = true

  TwoSat(int n = 0) : N(n), gr(2*n) {}

  int addVar() { // (optional)
    gr.emplace_back();
    gr.emplace_back();
    return N++;
  }

  void either(int f, int j) {
    f = max(2*f, -1-2*f);
    j = max(2*j, -1-2*j);
    gr[f].push_back(j^1);
    gr[j].push_back(f^1);
  }
  void setValue(int x) { either(x, x); }

  void atMostOne(const vi& li) { // (optional)
    if (sz(li) <= 1) return;
    int cur = ~li[0];
    rep(i,2,sz(li)) {
      int next = addVar();
      either(cur, ~li[i]);
      either(cur, next);
      either(~li[i], next);
      cur = ~next;
    }
    either(cur, ~li[1]);
  }

  vi val, comp, z; int time = 0;
  int dfs(int i) {
    int low = val[i] = ++time, x; z.push_back(
        i);
    for(int e : gr[i]) if (!comp[e])
      low = min(low, val[e] ?: dfs(e));
    if (low == val[i]) do {
      x = z.back(); z.pop_back();
      comp[x] = low;
      if (values[x>>1] == -1)
        values[x>>1] = x&1;
    } while (x != i);
    return val[i] = low;
  }

  bool solve() {
    values.assign(N, -1);
    val.assign(2*N, 0); comp = val;
    rep(i,0,2*N) if (!comp[i]) dfs(i);
    rep(i,0,N) if (comp[2*i] == comp[2*i+1])
      return 0;
    return 1;
  }
};
```

EulerWalk.h

Description: Eulerian undirected/directed path/cycle algorithm. Input should be a vector of (dest, global edge index), where for undirected graphs, forward/backward edges have the same index. Returns a list of nodes in the Eulerian path/cycle with src at both start and end, or empty list if no cycle/path exists. To get edge indices back, add .second to s and ret.

Time: $\mathcal{O}(V+E)$

780b64, 15 lines

```cpp
vi eulerWalk(vector<vector<pii>>& gr, int
      nedges, int src=0) {
  int n = sz(gr);
  vi D(n), its(n), eu(nedges), ret, s = {src};
  D[src]++; // to allow Euler paths, not just
      cycles
  while (!s.empty()) {
    int x = s.back(), y, e, &it = its[x], end
        = sz(gr[x]);
    if (it == end){ ret.push_back(x); s.
        pop_back(); continue; }
    tie(y, e) = gr[x][it++];
    if (!eu[e]) {
      D[x]--, D[y]++;
      eu[e] = 1; s.push_back(y);
    }}
  for (int x : D) if (x < 0 || sz(ret) !=
      nedges+1) return {};
  return {ret.rbegin(), ret.rend()};
}
```

CycleDetection.h

Description: Cycle detection for directed and undirected graphs

Time: $\mathcal{O}(V+E)$

c3e74b, 32 lines

```cpp
//cycle detect (undirected)
bool dfs(int u, int p) {
  vis[u] = true;
  par[u] = p;
  for (auto &it: adj[u]) {
    if (it == p) continue;
    if (vis[it]) {
      s = it, e = u;
      return true;
    }
    if (dfs(it, u)) return true;
  }
  return false;
};

//cycle detect (directed)
bool dfs(int v) {
  color[v] = 1;
  for (int u : adj[v]) {
    if (color[u] == 0) {
      parent[u] = v;
      if (dfs(u))
        return true;
    } else if (color[u] == 1) {
      cycle_end = v;
      cycle_start = u;
      return true;
    }
  }
  color[v] = 2;
  return false;
}
```

De-Bruijn Sequence: of order n on a k-size alphabet A is a cyclic sequence in which every possible length n string on A occurs exactly once as a substring. $B(k,n)$ has length k^n and number of distinct sequences is $\{(k!)^{k^{n-1}}\}/k^n$. Find an Euler tour on graph where nodes are $n-1$ length strings and each node has k outgoing edges for each character.

GrayCode.h

Description: Sequence of binary strings where each successive values differ in only 1 bit. Can be used to find Hamiltonian cycle on n-dimensional hypercube by calling $g(0),...,g(2^n-1)$.

e87165, 5 lines

```cpp
int g (int n) { return n ^ (n >> 1); }
int rev_g (int g) { int n = 0;
  for (; g; g >>= 1) n ^= g;
  return n;
}
```

7.5 Coloring

EdgeColoring.h

Description: Given a simple, undirected graph with max degree D, computes a $(D+1)$-coloring of the edges such that no neighboring edges share a color. (D-coloring is NP-hard, but can be done for bipartite graphs by repeated matchings of max-degree nodes.)

Time: $\mathcal{O}(NM)$

e210e2, 31 lines

```cpp
vi edgeColoring(int N, vector<pii> eds) {
  vi cc(N + 1), ret(sz(eds)), fan(N), free(N),
      loc;
  for (pii e : eds) ++cc[e.first], ++cc[e.
      second];
  int u, v, ncols = *max_element(all(cc)) + 1;
  vector<vi> adj(N, vi(ncols, -1));
  for (pii e : eds) {
    tie(u, v) = e;
    fan[0] = v;
    loc.assign(ncols, 0);
    int at = u, end = u, d, c = free[u], ind =
        0, i = 0;
    while (d = free[v], !loc[d] && (v = adj[u
        ][d]) != -1)
      loc[d] = ++ind, cc[ind] = d, fan[ind] =
          v;
    cc[loc[d]] = c;
    for (int cd = d; at != -1; cd ^= c ^ d, at
        = adj[at][cd])
      swap(adj[at][cd], adj[end = at][cd ^ c ^
          d]);
    while (adj[fan[i]][d] != -1) {
      int left = fan[i], right = fan[++i], e =
          cc[i];
      adj[u][e] = left;
      adj[left][e] = u;
      adj[right][e] = -1;
      free[right] = e;
    }
    adj[u][d] = fan[i];
    adj[fan[i]][d] = u;
    for (int y : {fan[0], u, end})
      for (int& z = free[y] = 0; adj[y][z] !=
          -1; z++);
  }
  rep(i,0,sz(eds))
    for (tie(u, v) = eds[i]; adj[u][ret[i]] !=
        v;) ++ret[i];
  return ret;
}
```

7.6 Heuristics

MaximalCliques.h

Description: Runs a callback for all maximal cliques in a graph (given as a symmetric bitset matrix; self-edges not allowed). Callback is given a bitset representing the maximal clique.

Time: $\mathcal{O}\left(3^{n/3}\right)$, much faster for sparse graphs

b0d5b1, 12 lines

```cpp
typedef bitset<128> B;
template<class F>
void cliques(vector<B>& eds, F f, B P = ~B(),
      B X={}, B R={}) {
  if (!P.any()) { if (!X.any()) f(R); return;
      }
  auto q = (P | X)._Find_first();
  auto cands = P & ~eds[q];
  rep(i,0,sz(eds)) if (cands[i]) {
    R[i] = 1;
    cliques(eds, f, P & eds[i], X & eds[i], R)
        ;
    R[i] = P[i] = 0; X[i] = 1;
  }
}
```

MaximumClique.h

Description: Quickly finds a maximum clique of a graph (given as symmetric bitset matrix; self-edges not allowed). Can be used to find a maximum independent set by finding a clique of the complement graph.

Time: Runs in about 1s for n=155 and worst case random graphs (p=.90). Runs faster for sparse graphs.

f7c0bc, 49 lines

```cpp
typedef vector<bitset<200>> vb;
struct Maxclique {
  double limit=0.025, pk=0;
  struct Vertex { int i, d=0; };
  typedef vector<Vertex> vv;
  vb e;
  vv V;
  vector<vi> C;
  vi qmax, q, S, old;
  void init(vv& r) {
    for (auto& v : r) v.d = 0;
    for (auto& v : r) for (auto j : r) v.d +=
        e[v.i][j.i];
    sort(all(r), [](auto a, auto b) { return a
        .d > b.d; });
    int mxD = r[0].d;
    rep(i,0,sz(r)) r[i].d = min(i, mxD) + 1;
  }
  void expand(vv& R, int lev = 1) {
    S[lev] += S[lev - 1] - old[lev];
    old[lev] = S[lev - 1];
    while (sz(R)) {
      if (sz(q) + R.back().d <= sz(qmax))
        return;
      q.push_back(R.back().i);
      vv T;
```

```
      for(auto v:R) if (e[R.back().i][v.i]) T.
          push_back({v.i});
      if (sz(T)) {
        if (S[lev]++ / ++pk < limit) init(T);
        int j = 0, mxk = 1, mnk = max(sz(qmax)
             - sz(q) + 1, 1);
        C[1].clear(), C[2].clear();
        for (auto v : T) {
          int k = 1;
          auto f = [&](int i) { return e[v.i][
               i]; };
          while (any_of(all(C[k]), f)) k++;
          if (k > mxk) mxk = k, C[mxk + 1].
               clear();
          if (k < mnk) T[j++].i = v.i;
          C[k].push_back(v.i);
        }
        if (j > 0) T[j - 1].d = 0;
        rep(k,mnk,mxk + 1) for (int i : C[k])
          T[j].i = i, T[j++].d = k;
        expand(T, lev + 1);
      } else if (sz(q) > sz(qmax)) qmax = q;
      q.pop_back(), R.pop_back();
    }
  }
  vi maxClique() { init(V), expand(V); return
       qmax; }
  Maxclique(vb conn) : e(conn), C(sz(e)+1), S(
       sz(C)), old(S) {
    rep(i,0,sz(e)) V.push_back({i});
  }
};
```

MaximumIndependentSet.h

Description: To obtain a maximum independent set of a graph, find a max clique of the complement. If the graph is bipartite, see MinimumVertexCover.

7.7 Trees

BinaryLifting.h

Description: Calculate power of two jumps in a tree, to support fast upward jumps and LCAs. Assumes the root node points to itself.
Time: construction $\mathcal{O}(N \log N)$, queries $\mathcal{O}(\log N)$

bfce85, 25 lines

```
vector<vi> treeJump(vi& P){
  int on = 1, d = 1;
  while(on < sz(P)) on *= 2, d++;
  vector<vi> jmp(d, P);
  rep(i,1,d) rep(j,0,sz(P))
    jmp[i][j] = jmp[i-1][jmp[i-1][j]];
  return jmp;
}

int jmp(vector<vi>& tbl, int nod, int steps){
  rep(i,0,sz(tbl))
    if(steps&(1<<i)) nod = tbl[i][nod];
  return nod;
}

int lca(vector<vi>& tbl, vi& depth, int a, int
     b) {
  if (depth[a] < depth[b]) swap(a, b);
  a = jmp(tbl, a, depth[a] - depth[b]);
  if (a == b) return a;
  for (int i = sz(tbl); i--;) {
    int c = tbl[i][a], d = tbl[i][b];
    if (c != d) a = c, b = d;
  }
  return tbl[0][a];
}
```

LCA.h

Description: Data structure for computing lowest common ancestors in a tree (with 0 as root). C should be an adjacency list of the tree, either directed or undirected.
Time: $\mathcal{O}(N \log N + Q)$

"../data-structures/RMQ.h" 0f62fb, 21 lines

```
struct LCA {
  int T = 0;
  vi time, path, ret;
  RMQ<int> rmq;

  LCA(vector<vi>& C) : time(sz(C)), rmq((dfs(C
       ,0,-1), ret)) {}
  void dfs(vector<vi>& C, int v, int par) {
    time[v] = T++;
    for (int y : C[v]) if (y != par) {
      path.push_back(v), ret.push_back(time[v
           ]);
      dfs(C, y, v);
    }
  }

  int lca(int a, int b) {
    if (a == b) return a;
    tie(a, b) = minmax(time[a], time[b]);
    return path[rmq.query(a, b)];
  }
  //dist(a,b){return depth[a] + depth[b] - 2*
       depth[lca(a,b)];}
};
```

CompressTree.h

Description: Given a rooted tree and a subset S of nodes, compute the minimal subtree that contains all the nodes by adding all (at most $|S|-1$) pairwise LCA's and compressing edges. Returns a list of (par, orig_index) representing a tree rooted at 0. The root points to itself.
Time: $\mathcal{O}(|S| \log |S|)$

"LCA.h" 9775a0, 21 lines

```
typedef vector<pair<int, int>> vpi;
vpi compressTree(LCA& lca, const vi& subset) {
  static vi rev; rev.resize(sz(lca.time));
  vi li = subset, &T = lca.time;
  auto cmp = [&](int a, int b) { return T[a] <
       T[b]; };
  sort(all(li), cmp);
  int m = sz(li)-1;
  rep(i,0,m) {
    int a = li[i], b = li[i+1];
    li.push_back(lca.lca(a, b));
  }
  sort(all(li), cmp);
  li.erase(unique(all(li)), li.end());
  rep(i,0,sz(li)) rev[li[i]] = i;
  vpi ret = {pii(0, li[0])};
  rep(i,0,sz(li)-1) {
    int a = li[i], b = li[i+1];
    ret.emplace_back(rev[lca.lca(a, b)], b);
  }
  return ret;
}
```

HLD.h

Description: Heavy-Light Decomposition (HLD) for efficient path queries on trees. Decomposes a tree into paths where a path from any node to the root traverses at most log(n) paths.
Usage: Process queries like finding max/min/sum on paths between nodes or updating node values.
Time: $\mathcal{O}(logn)$ per query/update, $\mathcal{O}(n)$ for preprocessing

08c750, 98 lines

```
const int N = 2e5 + 9, LG = 18;

struct ST {
  //include SegmentTreeIterative.h
} t;

vector<int> g[N];
int par[N][LG + 1], dep[N], sz[N];

void dfs(int u, int p = 0) {
  par[u][0] = p;
  dep[u] = dep[p] + 1;
  sz[u] = 1;
  for (int i = 1; i <= LG; i++) par[u][i] =
       par[par[u][i - 1]][i - 1];
  if (p) g[u].erase(find(g[u].begin(), g[u].
       end(), p));
  for (auto &v: g[u])
    if (v != p) {
      dfs(v, u);
      sz[u] += sz[v];
      if (sz[v] > sz[g[u][0]]) swap(v, g[u
           ][0]);
    }
}

int lca(int u, int v) {
  if (dep[u] < dep[v]) swap(u, v);
  for (int k = LG; k >= 0; k--) if (dep[par[u
       ][k]] >= dep[v]) u = par[u][k];
  if (u == v) return u;
  for (int k = LG; k >= 0; k--) if (par[u][k]
       != par[v][k]) u = par[u][k], v = par[v][
       k];
  return par[u][0];
}

int kth(int u, int k) {
  for (int i = 0; i <= LG; i++) if (k & (1 <<
       i)) u = par[u][i];
  return u;
}

int T, arr[N], head[N], st[N], en[N];

void dfs_hld(int u) {
  st[u] = ++T;
  for (auto v: g[u]) {
    head[v] = (v == g[u][0] ? head[u] : v);
    dfs_hld(v);
  }
  en[u] = T;
}

int n, q;

int query_up(int u, int v) {
  int ans = 0;
  while (head[u] != head[v]) {
    ans = max(ans, t.query(st[head[u]], st[u])
         );
    u = par[head[u]][0];
  }
  ans = max(ans, t.query(st[v], st[u]));
  return ans;
}

int query(int u, int v) {
  int l = lca(u, v);
  int ans = query_up(u, l);
  if (v != l) ans = max(ans, query_up(v, kth(v
       , dep[v] - dep[l] - 1)));
  return ans;
}

signed main() {
  cin.tie(0)->sync_with_stdio(0);
  cin >> n >> q;
  for (int i = 1; i <= n; ++i) cin >> arr[i];

  for (int i = 1; i < n; ++i) {
    int u, v;
    cin >> u >> v;
    g[u].push_back(v);
    g[v].push_back(u);
  }
  dfs(1);
  head[1] = 1;
  dfs_hld(1);
  t.n = n;
  t.t.assign(2 * n + 1, 0);
  for (int i = 1; i <= n; ++i) t.upd(st[i],
       arr[i]);
  while (q--) {
    int type;
    cin >> type;
    if (type == 1) {
      int s, x;
      cin >> s >> x;
      t.upd(st[s], x);
    }
    else {
      int a, b;
      cin >> a >> b;
      cout << query(a, b) << ' ';
    }
  }
}
```

LinkCutTree.h

Description: Represents a forest of unrooted trees. You can add and remove edges (as long as the result is still a forest), and check whether two nodes are in the same tree.
Time: All operations take amortized $\mathcal{O}(\log N)$.

0fb462, 90 lines

```
struct Node { // Splay tree. Root's pp
     contains tree's parent.
  Node *p = 0, *pp = 0, *c[2];
  bool flip = 0;
  Node() { c[0] = c[1] = 0; fix(); }
  void fix() {
    if (c[0]) c[0]->p = this;
    if (c[1]) c[1]->p = this;
    // (+ update sum of subtree elements etc.
         if wanted)
  }
```

```
  void pushFlip() {
    if (!flip) return;
    flip = 0; swap(c[0], c[1]);
    if (c[0]) c[0]->flip ^= 1;
    if (c[1]) c[1]->flip ^= 1;
  }
  int up() { return p ? p->c[1] == this : -1;
      }
  void rot(int i, int b) {
    int h = i ^ b;
    Node *x = c[i], *y = b == 2 ? x : x->c[h],
        *z = b ? y : x;
    if ((y->p = p)) p->c[up()] = y;
    c[i] = z->c[i ^ 1];
    if (b < 2) {
      x->c[h] = y->c[h ^ 1];
      y->c[h ^ 1] = x;
    }
    z->c[i ^ 1] = this;
    fix(); x->fix(); y->fix();
    if (p) p->fix();
    swap(pp, y->pp);
  }
  void splay() {
    for (pushFlip(); p; ) {
      if (p->p) p->p->pushFlip();
      p->pushFlip(); pushFlip();
      int c1 = up(), c2 = p->up();
      if (c2 == -1) p->rot(c1, 2);
      else p->p->rot(c2, c1 != c2);
    }
  }
  Node* first() {
    pushFlip();
    return c[0] ? c[0]->first() : (splay(),
        this);
  }
};

struct LinkCut {
  vector<Node> node;
  LinkCut(int N) : node(N) {}

  void link(int u, int v) { // add an edge (u,
      v)
    assert(!connected(u, v));
    makeRoot(&node[u]);
    node[u].pp = &node[v];
  }
  void cut(int u, int v) { // remove an edge (
      u, v)
    Node *x = &node[u], *top = &node[v];
    makeRoot(top); x->splay();
    assert(top == (x->pp ?: x->c[0]));
    if (x->pp) x->pp = 0;
    else {
      x->c[0] = top->p = 0;
      x->fix();
    }
  }
  bool connected(int u, int v) { // are u, v
      in the same tree?
    Node* nu = access(&node[u])->first();
    return nu == access(&node[v])->first();
  }
  void makeRoot(Node* u) {
    access(u);
    u->splay();
    if(u->c[0]) {
      u->c[0]->p = 0;
      u->c[0]->flip ^= 1;
      u->c[0]->pp = u;
      u->c[0] = 0;
      u->fix();
    }
  }
  Node* access(Node* u) {
    u->splay();
    while (Node* pp = u->pp) {
      pp->splay(); u->pp = 0;
      if (pp->c[1]) {
        pp->c[1]->p = 0; pp->c[1]->pp = pp; }
      pp->c[1] = u; pp->fix(); u = pp;
    }
    return u;
  }
};
```

TreeBinarize.h

Description: Given a weighted tree in edge-listing representation, transforms it into a binary tree by adding at most $2n$ extra nodes.

Usage: call add_edge() for both directions to create the tree. Then call binarize(1). Will change n.

84b697, 31 lines

```
// N = 3 * max nodes, M = 2 * N
int n, o = 2;
int to[M],wgt[M],prv[M],nxt[M],lst[N],deg[N];

void add_edge (int u, int v, int w) {
  to[o] = v, wgt[o] = w, deg[v]++;
  prv[o] = lst[u], lst[u] = nxt[lst[u]] = o++;
}
void binarize (int u, int f = 0) {
  int d = deg[u] - 2 - (f != 0);
  if (d > 0) {
    int tmp_lst = (to[lst[u]] == f ? prv[lst[u
        ]] : lst[u]), x;
    for (int e = lst[u], at = n+d; at > n; ){
        x = prv[e];
        if (to[e] == f) { e = x; continue; }
        nxt[x] = nxt[e];
        nxt[e] ? prv[nxt[e]] = x : lst[u] = x;
        prv[e] = lst[at], nxt[e] = 0;
        lst[at] = nxt[lst[at]] = e, deg[at]++;
        to[e ^ 1] = at;
        if (e != tmp_lst) --at;
        e = x;
    }
    for (int i=1, p=u; i <= d; p = n + i++)
       add_edge(p, n + i, 0),
       add_edge(n + i, p, 0);
    n += d, deg[u] -= d + 1;
  }
  for (int e = lst[u]; e; e = prv[e])
    if (to[e] != f) binarize(to[e], u);
}
```

CentroidDecomp.h

Description: Divide and conquer on trees. Useful for solving problems regarding all pairs of paths. Simple modifications are needed to integrate TreeBinarize into this.

Usage: Just call decompose(1). ctp[u] = parent of u in ctree. cth[u] = height of u in ctree,
root has height = 1. dist[u][h] = original tree distance (u -> ctree ancestor of u at height h).

Time: $\mathcal{O}(N \lg N)$

096de1, 24 lines

```
// H = __lg(N), reset: cth, ctp, dist
int sub[N], cth[N], ctp[N], dist[N][H + 1];
void dfs_siz (int u, int f) {
  sub[u] = 1;
  for (int v : g[u]) if (!cth[v] && v ^ f)
    dfs_siz(v, u), sub[u] += sub[v];
}
int fc (int u, int f, int lim) {
  for (int v : g[u]) if (!cth[v] && v ^ f &&
      sub[v] > lim) return fc(v, u, lim);
  return u;
}
void dfs_dist (int u, int f, int d, int h) {
  dist[u][h] = d;
  for (int v : g[u]) if (!cth[v] && v ^ f)
    dfs_dist(v, u, d + 1, h);
}
void decompose (int u, int f = 0, int h = 1) {
  dfs_siz(u, 0);
  u = fc(u, 0, sub[u] >> 1);
  dfs_dist(u, 0, 0, h);
  cth[u] = h, ctp[u] = f; // u now deleted
  for (int v : g[u]) if (!cth[v])
    decompose(v, u, h + 1);
}
```

EulerTour.h

Description: Euler tour technique for tree queries with segment tree.

Usage: euler_tour(root, -1);
update(1, 1, n, tin[i], tin[i], arr[i]); // Update node i's value
query(1, 1, n, tin[s], tout[s]); // Query subtree sum of node s

Time: $\mathcal{O}(n)$ for tour, $\mathcal{O}(logn)$ for queries/updates

3e476b, 8 lines

```
vector<int> t(4 * N), lazy(4 * N, -1), g[N];
int n, q, arr[N], tin[N], tout[N], timer = 0;

void euler_tour(int u, int par) {
  tin[u] = ++timer;
  for (auto &v: g[u]) if (v != par) euler_tour
      (v, u);
  tout[u] = timer;
}
```

Geometry (8)

8.1 Geometric primitives

Point.h

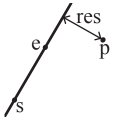
Description: Class to handle points in the plane. T can be e.g. double or long long. (Avoid int.)

ef0c0e, 29 lines

```
template <class T> int sgn(T x) { return (x >
    0) - (x < 0); }
template<class T>
struct Point {
typedef Point P;
T x, y;
explicit Point(T _x=0, T _y=0) : x(_x),y(_y){}
bool operator<(P p) const { return tie(x,y) <
    tie(p.x,p.y); }
bool operator==(P p) const { return tie(x,y)==
    tie(p.x,p.y); }
P operator+(P p) const{return P(x+p.x,y+p.y);}
P operator-(P p) const{return P(x-p.x,y-p.y);}
P operator*(T d) const { return P(x*d, y*d); }
P operator/(T d) const { return P(x/d, y/d); }
T dot(P p) const { return x*p.x + y*p.y; }
T cross(P p) const { return x*p.y - y*p.x; }
T cross(P a, P b) const { return (a-*this).
    cross(b-*this); }
T dist2() const { return x*x + y*y; }
double dist() const { return sqrt((double)
    dist2()); }
// angle to x-axis in interval [-pi, pi]
double angle() const { return atan2(y, x); }
// makes dist() = 1
P unit() const { return *this/dist(); }
// rotate by +90 degree
P perp() const { return P(-y, x); }
P normal() const { return perp().unit(); }
//rotate 'a' radians ccw around (0,0)
P rotate(double a) const { return P(x*cos(a)-y
    *sin(a),x*sin(a)+y*cos(a)); }
friend ostream& operator<<(ostream& os, P p) {
  return os<<"("<< p.x << "," << p.y << ")";}
};
```

lineDistance.h

Description:
Returns the signed distance between point p and the line containing points a and b. Positive value on left side and negative on right as seen from a towards b. a==b gives nan. P is supposed to be Point<T> or Point3D<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long. Using Point3D will always give a non-negative distance. For Point3D, call .dist on the result of the cross product.

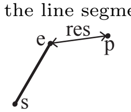


"Point.h" f6bf6b, 4 lines

```
template<class P>
double lineDist(const P& a, const P& b, const
    P& p) {
  return (double)(b-a).cross(p-a)/(b-a).dist()
      ;
}
```

SegmentDistance.h

Description:
Returns the shortest distance between point p and the line segment from point s to e.



Usage: Point<double> a, b(2,2), p(1,1);
bool onSegment = segDist(a,b,p) < 1e-10;

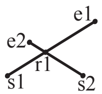
"Point.h" 5c88f4, 6 lines

```
typedef Point<double> P;
double segDist(P& s, P& e, P& p) {
  if (s==e) return (p-s).dist();
```

```
  auto d = (e-s).dist2(), t = min(d,max(.0,(p-
      s).dot(e-s)));
  return ((p-s)*d-(e-s)*t).dist()/d;
}
```

SegmentIntersection.h

Description:
If a unique intersection point between the line segments going from s1 to e1 and from s2 to e2 exists then it is returned. If no intersection point exists an empty vector is returned. If infinitely many exist a vector with 2 elements is returned, containing the endpoints of the common line segment. The wrong position will be returned if P is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or long long.



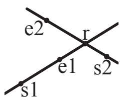
Usage: vector<P> inter = segInter(s1,e1,s2,e2);
if (sz(inter)==1)
cout << "segments intersect at " << inter[0] << endl;

"Point.h", "OnSegment.h" 9d57f2, 13 lines

```
template<class P> vector<P> segInter(P a, P b,
      P c, P d) {
  auto oa = c.cross(d, a), ob = c.cross(d, b),
       oc = a.cross(b, c), od = a.cross(b, d);
  // Checks if intersection is single non-
       endpoint point.
  if (sgn(oa) * sgn(ob) < 0 && sgn(oc) * sgn(
       od) < 0)
    return {(a * ob - b * oa) / (ob - oa)};
  set<P> s;
  if (onSegment(c, d, a)) s.insert(a);
  if (onSegment(c, d, b)) s.insert(b);
  if (onSegment(a, b, c)) s.insert(c);
  if (onSegment(a, b, d)) s.insert(d);
  return {all(s)};
}
```

lineIntersection.h

Description:
If a unique intersection point of the lines going through s1,e1 and s2,e2 exists {1, point} is returned. If no intersection point exists {0, (0,0)} is returned and if infinitely many exists {-1, (0,0)} is returned. The wrong position will be returned if P is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or ll.



Usage: auto res = lineInter(s1,e1,s2,e2);
if (res.first == 1)
cout << "intersection point at " << res.second << endl;

"Point.h" a01f81, 8 lines

```
template<class P>
pair<int, P> lineInter(P s1, P e1, P s2, P e2)
      {
  auto d = (e1 - s1).cross(e2 - s2);
  if (d == 0) // if parallel
    return {-(s1.cross(e1, s2) == 0), P(0, 0)}
        ;
  auto p = s2.cross(e1, e2), q = s2.cross(e2,
      s1);
  return {1, (s1 * p + e1 * q) / d};
}
```

sideOf.h

Description: Returns where p is as seen from s towards e. 1/0/-1 $\Leftrightarrow$ left/on line/right. If the optional argument *eps* is given 0 is returned if p is within distance *eps* from the line. P is supposed to be Point<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long.
Usage: bool left = sideOf(p1,p2,q)==1;

"Point.h" 3af81c, 9 lines

```
template<class P>
int sideOf(P s, P e, P p) { return sgn(s.cross
     (e, p)); }

template<class P>
int sideOf(const P& s, const P& e, const P& p,
     double eps) {
  auto a = (e-s).cross(p-s);
  double l = (e-s).dist()*eps;
  return (a > l) - (a < -l);
}
```

OnSegment.h

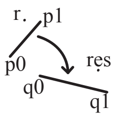
Description: Returns true iff p lies on the line segment from s to e. Use (segDist(s,e,p)<=epsilon) instead when using Point<double>.

"Point.h" c597e8, 3 lines

```
template<class P> bool onSegment(P s, P e, P p
     ) {
  return p.cross(s, e) == 0 && (s - p).dot(e -
       p) <= 0;
}
```

linearTransformation.h

Description:
Apply the linear transformation (translation, rotation and scaling) which takes line p0-p1 to line q0-q1 to point r.



"Point.h" 03a306, 6 lines

```
typedef Point<double> P;
P linearTransformation(const P& p0, const P&
     p1,
    const P& q0, const P& q1, const P& r) {
  P dp = p1-p0, dq = q1-q0, num(dp.cross(dq),
       dp.dot(dq));
  return q0 + P((r-p0).cross(num), (r-p0).dot(
       num))/dp.dist2();
}
```

Angle.h

Description: A class for ordering angles (as represented by int points and a number of rotations around the origin). Useful for rotational sweeping. Sometimes also represents points or vectors.
Usage: vector<Angle> v = {w[0], w[0].t360() ...}; // sorted
int j = 0; rep(i,0,n) { while (v[j] < v[i].t180()) ++j; }
// sweeps j such that (j-i) represents the number of positively oriented triangles with vertices at 0 and i

0f0602, 35 lines

```
struct Angle {
  int x, y;
  int t;
  Angle(int x, int y, int t=0) : x(x), y(y), t
      (t) {}
  Angle operator-(Angle b) const { return {x-b
      .x, y-b.y, t}; }
  int half() const {
    assert(x || y);
    return y < 0 || (y == 0 && x < 0);
  }
  Angle t90() const { return {-y, x, t + (half
      () && x >= 0)}; }
  Angle t180() const { return {-x, -y, t +
      half()}; }
  Angle t360() const { return {x, y, t + 1}; }
};
bool operator<(Angle a, Angle b) {
  // add a.dist2() and b.dist2() to also
       compare distances
  return make_tuple(a.t, a.half(), a.y * (ll)b
      .x) <
         make_tuple(b.t, b.half(), a.x * (ll)b
             .y);
}

// Given two points, this calculates the
     smallest angle between
// them, i.e., the angle that covers the
     defined line segment.
pair<Angle, Angle> segmentAngles(Angle a,
     Angle b) {
  if (b < a) swap(a, b);
  return (b < a.t180() ?
          make_pair(a, b) : make_pair(b, a.
              t360()));
}
Angle operator+(Angle a, Angle b) { // point a
     + vector b
  Angle r(a.x + b.x, a.y + b.y, a.t);
  if (a.t180() < r) r.t--;
  return r.t180() < a ? r.t360() : r;
}
Angle angleDiff(Angle a, Angle b) { // angle b
     - angle a
  int tu = b.t - a.t; a.t = b.t;
  return {a.x*b.x + a.y*b.y, a.x*b.y - a.y*b.x
      , tu - (b < a)};
}
```

8.2 Circles

CircleIntersection.h

Description: Computes the pair of points at which two circles intersect. Returns false in case of no intersection.

"Point.h" 84d6d3, 11 lines

```
typedef Point<double> P;
bool circleInter(P a,P b,double r1,double r2,
     pair<P, P>* out) {
  if (a == b) { assert(r1 != r2); return false
      ; }
  P vec = b - a;
  double d2 = vec.dist2(), sum = r1+r2, dif =
       r1-r2,
         p = (d2 + r1*r1 - r2*r2)/(d2*2), h2 =
             r1*r1 - p*p*d2;
  if (sum*sum < d2 || dif*dif > d2) return
       false;
  P mid = a + vec*p, per = vec.perp() * sqrt(
       fmax(0, h2) / d2);
  *out = {mid + per, mid - per};
  return true;
}
```

CircleTangents.h

Description: Finds the external tangents of two circles, or internal if r2 is negated. Can return 0, 1, or 2 tangents – 0 if one circle contains the other (or overlaps it, in the internal case, or if the circles are the same); 1 if the circles are tangent to each other (in which case .first = .second and the tangent line is perpendicular to the line between the centers). .first and .second give the tangency points at circle 1 and 2 respectively. To find the tangents of a circle with a point set r2 to 0.

"Point.h" b0153d, 13 lines

```
template<class P>
vector<pair<P, P>> tangents(P c1, double r1, P
      c2, double r2) {
  P d = c2 - c1;
  double dr = r1 - r2, d2 = d.dist2(), h2 = d2
       - dr * dr;
  if (d2 == 0 || h2 < 0)  return {};
  vector<pair<P, P>> out;
  for (double sign : {-1, 1}) {
    P v = (d * dr + d.perp() * sqrt(h2) * sign
        ) / d2;
    out.push_back({c1 + v * r1, c2 + v * r2});
  }
  if (h2 == 0) out.pop_back();
  return out;
}
```

CircleLine.h

Description: Finds the intersection between a circle and a line. Returns a vector of either 0, 1, or 2 intersection points. P is intended to be Point<double>.

"Point.h" e0cfba, 9 lines

```
template<class P>
vector<P> circleLine(P c, double r, P a, P b)
     {
  P ab = b - a, p = a + ab * (c-a).dot(ab) /
       ab.dist2();
  double s = a.cross(b, c), h2 = r*r - s*s /
       ab.dist2();
  if (h2 < 0) return {};
  if (h2 == 0) return {p};
  P h = ab.unit() * sqrt(h2);
  return {p - h, p + h};
}
```

CirclePolygonIntersection.h

Description: Returns the area of the intersection of a circle with a ccw polygon.

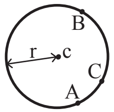
Time: $\mathcal{O}(n)$

"../../content/geometry/Point.h" a1ee63, 19 lines

```
typedef Point<double> P;
#define arg(p, q) atan2(p.cross(q), p.dot(q))
double circlePoly(P c, double r, vector<P> ps) {
  auto tri = [&](P p, P q) {
    auto r2 = r * r / 2;
    P d = q - p;
    auto a = d.dot(p)/d.dist2(), b = (p.dist2()-r*r)/d.dist2();
    auto det = a * a - b;
    if (det <= 0) return arg(p, q) * r2;
    auto s = max(0., -a-sqrt(det)), t = min(1., -a+sqrt(det));
    if (t < 0 || 1 <= s) return arg(p, q) * r2;
    P u = p + d * s, v = p + d * t;
    return arg(p,u) * r2 + u.cross(v)/2 + arg(v,q) * r2;
  };
  auto sum = 0.0;
  rep(i,0,sz(ps))
    sum += tri(ps[i] - c, ps[(i + 1) % sz(ps)] - c);
  return sum;
}
```

circumcircle.h

Description:
The circumcirle of a triangle is the circle intersecting all three vertices. ccRadius returns the radius of the circle going through points A, B and C and ccCenter returns the center of the same circle.



"Point.h" 1caa3a, 9 lines

```
typedef Point<double> P;
double ccRadius(const P& A, const P& B, const P& C) {
  return (B-A).dist()*(C-B).dist()*(A-C).dist()/
      abs((B-A).cross(C-A))/2;
}
P ccCenter(const P& A, const P& B, const P& C) {
  P b = C-A, c = B-A;
  return A + (b*c.dist2()-c*b.dist2()).perp()/b.cross(c)/2;
}
```

MinimumEnclosingCircle.h

Description: Computes the minimum circle that encloses a set of points.
Time: expected $\mathcal{O}(n)$

"circumcircle.h" 09dd0a, 17 lines

```
pair<P, double> mec(vector<P> ps) {
  shuffle(all(ps), mt19937(time(0)));
  P o = ps[0];
  double r = 0, EPS = 1 + 1e-8;
  rep(i,0,sz(ps)) if ((o - ps[i]).dist() > r * EPS) {
    o = ps[i], r = 0;
    rep(j,0,i) if ((o - ps[j]).dist() > r * EPS) {
      o = (ps[i] + ps[j]) / 2;
      r = (o - ps[i]).dist();
      rep(k,0,j) if ((o - ps[k]).dist() > r * EPS) {
        o = ccCenter(ps[i], ps[j], ps[k]);
        r = (o - ps[i]).dist();
      }
    }
  }
  return {o, r};
}
```

8.3 Polygons

InsidePolygon.h

Description: Returns true if p lies within the polygon. If strict is true, it returns false for points on the boundary. The algorithm uses products in intermediate steps so watch out for overflow.
Usage: `vector<P> v = {P{4,4}, P{1,2}, P{2,1}};`
`bool in = inPolygon(v, P{3, 3}, false);`
Time: $\mathcal{O}(n)$

"Point.h", "OnSegment.h", "SegmentDistance.h" 2bf504, 11 lines

```
template<class P>
bool inPolygon(vector<P> &p, P a, bool strict = true) {
  int cnt = 0, n = sz(p);
  rep(i,0,n) {
    P q = p[(i + 1) % n];
    if (onSegment(p[i], q, a)) return !strict;
    //or: if (segDist(p[i], q, a) <= eps) return !strict;
    cnt ^= ((a.y<p[i].y) - (a.y<q.y)) * a.cross(p[i], q) > 0;
  }
  return cnt;
}
```

PolygonArea.h

Description: Returns twice the signed area of a polygon. Clockwise enumeration gives negative area. Watch out for overflow if using int as T!

"Point.h" f12300, 6 lines

```
template<class T>
T polygonArea2(vector<Point<T>>& v) {
  T a = v.back().cross(v[0]);
  rep(i,0,sz(v)-1) a += v[i].cross(v[i+1]);
  return a;
}
```

PolygonCenter.h

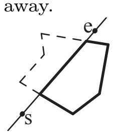
Description: Returns the center of mass for a polygon.
Time: $\mathcal{O}(n)$

"Point.h" 9706dc, 9 lines

```
typedef Point<double> P;
P polygonCenter(const vector<P>& v) {
  P res(0, 0); double A = 0;
  for (int i = 0, j = sz(v) - 1; i < sz(v); j = i++) {
    res = res + (v[i] + v[j]) * v[j].cross(v[i]);
    A += v[j].cross(v[i]);
  }
  return res / A / 3;
}
```

PolygonCut.h

Description:
Returns a vector with the vertices of a polygon with everything to the left of the line going from s to e cut away.



Usage: `vector<P> p = ...;`
`p = polygonCut(p, P(0,0), P(1,0));`

"Point.h", "lineIntersection.h" f2b7d4, 13 lines

```
typedef Point<double> P;
vector<P> polygonCut(const vector<P>& poly, P s, P e) {
  vector<P> res;
  rep(i,0,sz(poly)) {
    P cur = poly[i], prev = i ? poly[i-1] : poly.back();
    bool side = s.cross(e, cur) < 0;
    if (side != (s.cross(e, prev) < 0))
      res.push_back(lineInter(s, e, cur, prev).second);
    if (side)
      res.push_back(cur);
  }
  return res;
}
```

PolygonUnion.h

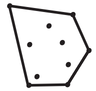
Description: Calculates the area of the union of n polygons (not necessarily convex). The points within each polygon must be given in CCW order. (Epsilon checks may optionally be added to sideOf/sgn, but shouldn't be needed.)
Time: $\mathcal{O}(N^2)$, where N is the total number of points

"Point.h", "sideOf.h" 3931c6, 33 lines

```
typedef Point<double> P;
double rat(P a, P b) { return sgn(b.x) ? a.x/b.x : a.y/b.y; }
double polyUnion(vector<vector<P>>& poly) {
  double ret = 0;
  rep(i,0,sz(poly)) rep(v,0,sz(poly[i])) {
    P A = poly[i][v], B = poly[i][(v + 1) % sz(poly[i])];
    vector<pair<double, int>> segs = {{0, 0}, {1, 0}};
    rep(j,0,sz(poly)) if (i != j) {
      rep(u,0,sz(poly[j])) {
        P C = poly[j][u], D = poly[j][(u + 1) % sz(poly[j])];
        int sc = sideOf(A, B, C), sd = sideOf(A, B, D);
        if (sc != sd) {
          double sa = C.cross(D, A), sb = C.cross(D, B);
          if (min(sc, sd) < 0)
            segs.emplace_back(sa / (sa - sb), sgn(sc - sd));
        } else if (!sc && !sd && j<i && sgn((B-A).dot(D-C))>0){
          segs.emplace_back(rat(C - A, B - A), 1);
          segs.emplace_back(rat(D - A, B - A), -1);
        }
      }
    }
    sort(all(segs));
    for (auto& s : segs) s.first = min(max(s.first, 0.0), 1.0);
    double sum = 0;
    int cnt = segs[0].second;
    rep(j,1,sz(segs)) {
      if (!cnt) sum += segs[j].first - segs[j - 1].first;
      cnt += segs[j].second;
    }
    ret += A.cross(B) * sum;
  }
  return ret / 2;
}
```

ConvexHull.h

Description:
Returns a vector of the points of the convex hull in counter-clockwise order. Points on the edge of the hull between two other points are not considered part of the hull.



Time: $\mathcal{O}(n \log n)$

"Point.h" 310954, 13 lines

```
typedef Point<ll> P;
vector<P> convexHull(vector<P> pts) {
  if (sz(pts) <= 1) return pts;
  sort(all(pts));
  vector<P> h(sz(pts)+1);
  int s = 0, t = 0;
  for (int it = 2; it--; s = --t, reverse(all(pts)))
    for (P p : pts) {
      while (t >= s + 2 && h[t-2].cross(h[t-1], p) <= 0) t--;
      h[t++] = p;
    }
  return {h.begin(), h.begin() + t - (t == 2 && h[0] == h[1])};
}
```

HullDiameter.h

Description: Returns the two points with max distance on a convex hull (ccw, no duplicate/collinear points).
Time: $\mathcal{O}(n)$

"Point.h" c571b8, 12 lines

```
typedef Point<ll> P;
array<P, 2> hullDiameter(vector<P> S) {
  int n = sz(S), j = n < 2 ? 0 : 1;
  pair<ll, array<P, 2>> res({0, {S[0], S[0]}});
  rep(i,0,j)
    for (;; j = (j + 1) % n) {
      res = max(res, {(S[i] - S[j]).dist2(), {S[i], S[j]}});
      if ((S[(j + 1) % n] - S[j]).cross(S[i + 1] - S[i]) >= 0)
        break;
    }
  return res.second;
}
```

PointInsideHull.h

Description: Determine whether a point t lies inside a convex hull (CCW order, with no collinear points). Returns true if point lies within the hull. If strict is true, points on the boundary aren't included.

Time: $\mathcal{O}(\log N)$

"Point.h", "sideOf.h", "OnSegment.h" 71446b, 14 lines

```
typedef Point<ll> P;

bool inHull(const vector<P>& l, P p, bool
    strict = true) {
  int a = 1, b = sz(l) - 1, r = !strict;
  if (sz(l) < 3) return r && onSegment(l[0], l
      .back(), p);
  if (sideOf(l[0], l[a], l[b]) > 0) swap(a, b)
      ;
  if (sideOf(l[0], l[a], p) >= r || sideOf(l
      [0], l[b], p)<= -r)
    return false;
  while (abs(a - b) > 1) {
    int c = (a + b) / 2;
    (sideOf(l[0], l[c], p) > 0 ? b : a) = c;
  }
  return sgn(l[a].cross(l[b], p)) < r;
}
```

LineHullIntersection.h

Description: Line-convex polygon intersection. The polygon must be ccw and have no collinear points. lineHull(line, poly) returns a pair describing the intersection of a line with the polygon: • $(-1,-1)$ if no collision, • $(i,-1)$ if touching the corner i, • (i,i) if along side $(i,i+1)$, • (i,j) if crossing sides $(i,i+1)$ and $(j,j+1)$. In the last case, if a corner i is crossed, this is treated as happening on side $(i,i+1)$. The points are returned in the same order as the line hits the polygon. `extrVertex` returns the point of a hull with the max projection onto a line.

Time: $\mathcal{O}(\log n)$

"Point.h" 7cf45b, 39 lines

```
#define cmp(i,j) sgn(dir.perp().cross(poly[(i)
    %n]-poly[(j)%n]))
#define extr(i) cmp(i + 1, i) >= 0 && cmp(i, i
    - 1 + n) < 0
template <class P> int extrVertex(vector<P>&
    poly, P dir) {
  int n = sz(poly), lo = 0, hi = n;
  if (extr(0)) return 0;
  while (lo + 1 < hi) {
    int m = (lo + hi) / 2;
    if (extr(m)) return m;
    int ls = cmp(lo + 1, lo), ms = cmp(m + 1,
        m);
    (ls < ms || (ls == ms && ls == cmp(lo, m))
        ? hi : lo) = m;
  }
  return lo;
}

#define cmpL(i) sgn(a.cross(poly[i], b))
template <class P>
array<int, 2> lineHull(P a, P b, vector<P>&
    poly) {
  int endA = extrVertex(poly, (a - b).perp());
  int endB = extrVertex(poly, (b - a).perp());
  if (cmpL(endA) < 0 || cmpL(endB) > 0)
    return {-1, -1};
  array<int, 2> res;
  rep(i,0,2) {
    int lo = endB, hi = endA, n = sz(poly);
    while ((lo + 1) % n != hi) {
      int m = ((lo + hi + (lo < hi ? 0 : n)) /
          2) % n;
      (cmpL(m) == cmpL(endB) ? lo : hi) = m;
    }
    res[i] = (lo + !cmpL(hi)) % n;
    swap(endA, endB);
  }
  if (res[0] == res[1]) return {res[0], -1};
  if (!cmpL(res[0]) && !cmpL(res[1]))
    switch ((res[0] - res[1] + sz(poly) + 1) %
        sz(poly)) {
      case 0: return {res[0], res[0]};
      case 2: return {res[1], res[1]};
    }
  return res;
}
```

8.4 Misc. Point Set Problems

ClosestPair.h

Description: Finds the closest pair of points.

Time: $\mathcal{O}(n \log n)$

"Point.h" ac41a6, 17 lines

```
typedef Point<ll> P;
pair<P, P> closest(vector<P> v) {
  assert(sz(v) > 1);
  set<P> S;
  sort(all(v), [](P a, P b) { return a.y < b.y
      ; });
  pair<ll, pair<P, P>> ret{LLONG_MAX, {P(), P
      ()}};
  int j = 0;
  for (P p : v) {
    P d{1 + (ll)sqrt(ret.first), 0};
    while (v[j].y <= p.y - d.x) S.erase(v[j
        ++]);
    auto lo = S.lower_bound(p - d), hi = S.
        upper_bound(p + d);
    for (; lo != hi; ++lo)
      ret = min(ret, {(*lo - p).dist2(), {*lo,
          p}});
    S.insert(p);
  }
  return ret.second;
}
```

kdTree.h

Description: KD-tree (2d, can be extended to 3d)

"Point.h" bac5b0, 63 lines

```
typedef long long T;
typedef Point<T> P;
const T INF = numeric_limits<T>::max();

bool on_x(const P& a, const P& b) { return a.x
    < b.x; }
bool on_y(const P& a, const P& b) { return a.y
    < b.y; }

struct Node {
  P pt; // if this is a leaf, the single point
      in it
  T x0 = INF, x1 = -INF, y0 = INF, y1 = -INF;
      // bounds
  Node *first = 0, *second = 0;

  T distance(const P& p) { // min squared
      distance to a point
    T x = (p.x < x0 ? x0 : p.x > x1 ? x1 : p.x
        );
    T y = (p.y < y0 ? y0 : p.y > y1 ? y1 : p.y
        );
    return (P(x,y) - p).dist2();
  }

  Node(vector<P>&& vp) : pt(vp[0]) {
    for (P p : vp) {
      x0 = min(x0, p.x); x1 = max(x1, p.x);
      y0 = min(y0, p.y); y1 = max(y1, p.y);
    }
    if (vp.size() > 1) {
      // split on x if width >= height (not
          ideal...)
      sort(all(vp), x1 - x0 >= y1 - y0 ? on_x
          : on_y);
      // divide by taking half the array for
          each child (not
      // best performance with many duplicates
          in the middle)
      int half = sz(vp)/2;
      first = new Node({vp.begin(), vp.begin()
          + half});
      second = new Node({vp.begin() + half, vp
          .end()});
    }
  }
};

struct KDTree {
  Node* root;
  KDTree(const vector<P>& vp) : root(new Node(
      {all(vp)})) {}

  pair<T, P> search(Node *node, const P& p) {
    if (!node->first) {
      // uncomment if we should not find the
          point itself:
      // if (p == node->pt) return {INF, P()};
      return make_pair((p - node->pt).dist2(),
          node->pt);
    }

    Node *f = node->first, *s = node->second;
    T bfirst = f->distance(p), bsec = s->
        distance(p);
    if (bfirst > bsec) swap(bsec, bfirst),
        swap(f, s);

    // search closest side first, other side
        if needed
    auto best = search(f, p);
    if (bsec < best.first)
      best = min(best, search(s, p));
    return best;
  }

  // find nearest point to a point, and its
      squared distance
  // (requires an arbitrary operator< for
      Point)
  pair<T, P> nearest(const P& p) {
    return search(root, p);
  }
};
```

hplane-cpalg.h

Description: Half plane intersection in O(n log n). The direction of the plane is ccw of pq vector in Halfplane struct. Usage: Status:

2e310c, 75 lines

```
const long double eps = 1e-9, inf = 1e9;

struct Point {
    long double x, y;
    explicit Point(long double x = 0, long
        double y = 0) : x(x), y(y) {}
    friend Point operator+(const Point &p,
        const Point &q) { return Point(p.x + q
        .x, p.y + q.y); }
    friend Point operator-(const Point &p,
        const Point &q) { return Point(p.x - q
        .x, p.y - q.y); }
    friend Point operator*(const Point &p,
        const long double &k) { return Point(p
        .x * k, p.y * k); }
    friend long double dot(const Point &p,
        const Point &q) { return p.x * q.x + p
        .y * q.y; }
    friend long double cross(const Point &p,
        const Point &q) { return p.x * q.y - p
        .y * q.x; }
};

struct Halfplane {
    Point p, pq;
    long double angle;
    Halfplane() {}
    Halfplane(const Point &a, const Point &b)
        : p(a), pq(b - a) {
        angle = atan2l(pq.y, pq.x);
    }
    bool out(const Point &r) { return cross(pq
        , r - p) < -eps; }
    bool operator<(const Halfplane &e) const {
         return angle < e.angle; }
    friend Point inter(const Halfplane &s,
        const Halfplane &t) {
        long double alpha = cross((t.p - s.p),
            t.pq) / cross(s.pq, t.pq);
        return s.p + (s.pq * alpha);
    }
};

vector<Point> hp_intersect(vector<Halfplane> &
    H) {

    Point box[4] = {Point(inf, inf), Point(-
        inf, inf), Point(-inf, -inf),
                    Point(inf, -inf)};

    for (int i = 0; i < 4; i++) {
        Halfplane aux(box[i], box[(i + 1) %
            4]);
        H.push_back(aux);
    }

    sort(H.begin(), H.end());
    deque<Halfplane> dq;
    int len = 0;
    for (int i = 0; i < int(H.size()); i++) {
        while (len > 1 && H[i].out(inter(dq[
            len - 1], dq[len - 2]))) {
            dq.pop_back(); --len;
```

```
        }
        while (len > 1 && H[i].out(inter(dq
             [0], dq[1]))) {
            dq.pop_front(); --len;
        }
        if (len > 0 && fabsl(cross(H[i].pq, dq
             [len - 1].pq)) < eps) {
            if (dot(H[i].pq, dq[len - 1].pq) <
                 0.0)
                return vector<Point>();
            if (H[i].out(dq[len - 1].p)) {
                dq.pop_back();
                --len;
            } else
                continue;
        }
        dq.push_back(H[i]);
        ++len;
    }
    while (len > 2 && dq[0].out(inter(dq[len -
         1], dq[len - 2]))) {
        dq.pop_back(); --len;
    }

    while (len > 2 && dq[len - 1].out(inter(dq
         [0], dq[1]))) {
        dq.pop_front(); --len;
    }
    if (len < 3)
        return vector<Point>();
    vector<Point> ret(len);
    for (int i = 0; i + 1 < len; i++) {
        ret[i] = inter(dq[i], dq[i + 1]);
    }
    ret.back() = inter(dq[len - 1], dq[0]);
    return ret;
}
```

8.5 3D

PolyhedronVolume.h

Description: Magic formula for the volume of a polyhedron. Faces should point outwards.

3058c3, 6 lines

```
template<class V, class L>
double signedPolyVolume(const V& p, const L&
     trilist) {
  double v = 0;
  for (auto i : trilist) v += p[i.a].cross(p[i
       .b]).dot(p[i.c]);
  return v / 6;
}
```

Point3D.h

Description: Class to handle points in 3D space. T can be e.g. double or long long.

8058ae, 32 lines

```
template<class T> struct Point3D {
  typedef Point3D P;
  typedef const P& R;
  T x, y, z;
  explicit Point3D(T x=0, T y=0, T z=0) : x(x)
     , y(y), z(z) {}
  bool operator<(R p) const {
    return tie(x, y, z) < tie(p.x, p.y, p.z);
         }
  bool operator==(R p) const {
    return tie(x, y, z) == tie(p.x, p.y, p.z);
         }
  P operator+(R p) const { return P(x+p.x, y+p
       .y, z+p.z); }
  P operator-(R p) const { return P(x-p.x, y-p
       .y, z-p.z); }
  P operator*(T d) const { return P(x*d, y*d,
       z*d); }
  P operator/(T d) const { return P(x/d, y/d,
       z/d); }
  T dot(R p) const { return x*p.x + y*p.y + z*
       p.z; }
  P cross(R p) const {
    return P(y*p.z - z*p.y, z*p.x - x*p.z, x*p
         .y - y*p.x);
  }
  T dist2() const { return x*x + y*y + z*z; }
  double dist() const { return sqrt((double)
       dist2()); }
  //Azimuthal angle (longitude) to x-axis in
       interval [-pi, pi]
  double phi() const { return atan2(y, x); }
  //Zenith angle (latitude) to the z-axis in
       interval [0, pi]
  double theta() const { return atan2(sqrt(x*x
       +y*y),z); }
  P unit() const { return *this/(T)dist(); }
       //makes dist()=1
  //returns unit vector normal to *this and p
  P normal(P p) const { return cross(p).unit()
       ; }
  //returns point rotated 'angle' radians ccw
       around axis
  P rotate(double angle, P axis) const {
    double s = sin(angle), c = cos(angle); P u
         = axis.unit();
    return u*dot(u)*(1-c) + (*this)*c - cross(
         u)*s;
  }
};
```

3dHull.h

Description: Computes all faces of the 3-dimension hull of a point set. *No four points must be coplanar*, or else random results will be returned. All faces will point outwards.

Time: $\mathcal{O}\left(n^2\right)$

"Point3D.h" 5b45fc, 49 lines

```
typedef Point3D<double> P3;

struct PR {
  void ins(int x) { (a == -1 ? a : b) = x; }
  void rem(int x) { (a == x ? a : b) = -1; }
  int cnt() { return (a != -1) + (b != -1); }
  int a, b;
};

struct F { P3 q; int a, b, c; };

vector<F> hull3d(const vector<P3>& A) {
  assert(sz(A) >= 4);
  vector<vector<PR>> E(sz(A), vector<PR>(sz(A)
       , {-1, -1}));
#define E(x,y) E[f.x][f.y]
  vector<F> FS;
  auto mf = [&](int i, int j, int k, int l) {
    P3 q = (A[j] - A[i]).cross((A[k] - A[i]));
    if (q.dot(A[l]) > q.dot(A[i]))
      q = q * -1;
    F f{q, i, j, k};
    E(a,b).ins(k); E(a,c).ins(j); E(b,c).ins(i
         );
    FS.push_back(f);
  };
  rep(i,0,4) rep(j,i+1,4) rep(k,j+1,4)
    mf(i, j, k, 6 - i - j - k);

  rep(i,4,sz(A)) {
    rep(j,0,sz(FS)) {
      F f = FS[j];
      if(f.q.dot(A[i]) > f.q.dot(A[f.a])) {
        E(a,b).rem(f.c);
        E(a,c).rem(f.b);
        E(b,c).rem(f.a);
        swap(FS[j--], FS.back());
        FS.pop_back();
      }
    }
    int nw = sz(FS);
    rep(j,0,nw) {
      F f = FS[j];
#define C(a, b, c) if (E(a,b).cnt() != 2) mf(f
     .a, f.b, i, f.c);
      C(a, b, c); C(a, c, b); C(b, c, a);
    }
  }
  for (F& it : FS) if ((A[it.b] - A[it.a]).
       cross(
    A[it.c] - A[it.a]).dot(it.q) <= 0) swap(it
         .c, it.b);
  return FS;
};
```

sphericalDistance.h

Description: Returns the shortest distance on the sphere with radius radius between the points with azimuthal angles (longitude) f1 (ϕ_1) and f2 (ϕ_2) from x axis and zenith angles (latitude) t1 (θ_1) and t2 (θ_2) from z axis (0 = north pole). All angles measured in radians. The algorithm starts by converting the spherical coordinates to cartesian coordinates so if that is what you have you can use only the two last rows. dx*radius is then the difference between the two points in the x direction and d*radius is the total distance between the points.

611f07, 8 lines

```
double sphericalDistance(double f1, double t1,
    double f2, double t2, double radius) {
  double dx = sin(t2)*cos(f2) - sin(t1)*cos(f1
       );
  double dy = sin(t2)*sin(f2) - sin(t1)*sin(f1
       );
  double dz = cos(t2) - cos(t1);
  double d = sqrt(dx*dx + dy*dy + dz*dz);
  return radius*2*asin(d/2);
}
```

Strings (9)

KMP.h

Description: pi[x] computes the length of the longest prefix of s that ends at x, other than s[0...x] itself (abacaba -> 0010123). Can be used to find all occurrences of a string.

Time: $\mathcal{O}(n)$

d4375c, 16 lines

```
vi pi(const string& s) {
  vi p(sz(s));
  rep(i,1,sz(s)) {
    int g = p[i-1];
    while (g && s[i] != s[g]) g = p[g-1];
    p[i] = g + (s[i] == s[g]);
  }
  return p;
}

vi match(const string& s, const string& pat) {
  vi p = pi(pat + '\0' + s), res;
  rep(i,sz(p)-sz(s),sz(p))
    if (p[i] == sz(pat)) res.push_back(i - 2 *
         sz(pat));
  return res;
}
```

Zfunc.h

Description: z[i] computes the length of the longest common prefix of s[i:] and s, except z[0] = 0. (abacaba -> 0010301)

Time: $\mathcal{O}(n)$

ee09e2, 12 lines

```
vi Z(const string& S) {
  vi z(sz(S));
  int l = -1, r = -1;
  rep(i,1,sz(S)) {
    z[i] = i >= r ? 0 : min(r - i, z[i - l]);
    while (i + z[i] < sz(S) && S[i + z[i]] ==
         S[z[i]])
      z[i]++;
    if (i + z[i] > r)
      l = i, r = i + z[i];
  }
  return z;
}
```

Manacher.h

Description: For each position in a string, computes p[0][i] = half length of longest even palindrome around pos i, p[1][i] = longest odd (half rounded down).

Time: $\mathcal{O}(N)$

e7ad79, 13 lines

```
array<vi, 2> manacher(const string& s) {
  int n = sz(s);
  array<vi,2> p = {vi(n+1), vi(n)};
  rep(z,0,2) for (int i=0,l=0,r=0; i < n; i++)
       {
    int t = r-i+!z;
    if (i<r) p[z][i] = min(t, p[z][l+t]);
    int L = i-p[z][i], R = i+p[z][i]-!z;
    while (L>=1 && R+1<n && s[L-1] == s[R+1])
      p[z][i]++, L--, R++;
    if (R>r) l=L, r=R;
  }
  return p;
}
```

Trie.h

Description: Basic Trie (Prefix Tree) implementation for string storage and retrieval. Supports insertion, search, and prefix matching operations.

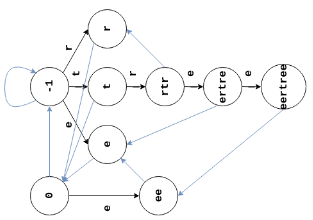
Usage: Insert strings, check prefixes, search for complete words

Time: $\mathcal{O}(L)$ per operation where L is the length of the string

94818b, 7 lines

```
struct trie {
  int sz = 0;
  trie *nxt[26];
};
```

```
trie root = trie();
trie *now = &root;
```



PalindromicTree.h

Description: Makes a trie of $\mathcal{O}(|S|)$ vertices containing all distinct palindromes of a string. Suffix links give the longest proper suffix/prefix of that palindrome which is also a palindrome.
Usage: S := 1-indexed string. append characters 1-by-1.
After adding the ith character, ptr points to the node containing
the longest palindrome ending at i. Change ALPHA, ID() as problem requires.
Time: $\mathcal{O}(|S|)$ 13f2cf, 36 lines

```
const int ALPHA = 26;
struct PalindromicTree {
  struct node {
    int to[ALPHA];
    int link, len;
    node(int a=0, int b=0) : link(a), len(b) {
      memset(to, 0, sizeof to);
  } };
  V<node> T; int ptr;
  int ID(char x) { return x - 'a'; }
  void init() {
    T.clear(); ptr = 1;
    T.emplace_back(0, -1); // 0=Odd root
    T.emplace_back(0, 0);  // 1=Evn root
  }
  void append(int i, string &s) {
    while (s[i - T[ptr].len - 1] != s[i])
      ptr = T[ptr].link;

    int id = ID(s[i]);
    // if node already exists, return
    if (T[ptr].to[id]) return void(ptr = T[ptr
        ].to[id]);
    int tmp = T[ptr].link;
    while (s[i - T[tmp].len - 1] != s[i])
      tmp = T[tmp].link;

    int newlink = T[ptr].len == -1 ? 1 : T[tmp
        ].to[id];

    // ptr is the parent of this new node
    T.emplace_back(newlink, T[ptr].len + 2);

    // Now shift ptr to the newly created node
    T[ptr].to[id] = sz(T) - 1;
    ptr = sz(T) - 1;
  }
};
```

MinRotation.h

Description: Finds the lexicographically smallest rotation of a string.
Usage: rotate(v.begin(), v.begin()+minRotation(v), v.end());
Time: $\mathcal{O}(N)$ d07a42, 8 lines

```
int minRotation(string s) {
  int a=0, N=sz(s); s += s;
  rep(b,0,N) rep(k,0,N) {
    if (a+k == b || s[a+k] < s[b+k]) {b += max
        (0, k-1); break;}
    if (s[a+k] > s[b+k]) { a = b; break; }
  }
  return a;
}
```

SuffixArray.h

Description: Builds suffix array for a string. sa[i] is the starting index of the suffix which is i'th in the sorted suffix array. The returned vector is of size $n+1$, and sa[0] = n. lcp[i] = lcp(sa[i], sa[i-1]), lcp[0] = 0. The input string must not contain any nul chars.
Time: $\mathcal{O}(n \log n)$ 635552, 22 lines

```
struct SuffixArray {
  vi sa, lcp; // passing V<int> also works
  SuffixArray(string s, int lim=256) {
    s.push_back(0); int n = sz(s), k = 0, a,b;
    vi x(all(s)), y(n), ws(max(n, lim));
    sa = lcp = y, iota(all(sa), 0);
    for (int j = 0, p = 0; p < n; j = max(1, j
        * 2), lim = p) {
      p = j, iota(all(y), n - j);
      rep(i,0,n) if(sa[i]>=j) y[p++]=sa[i]-j;
      fill(all(ws), 0);
      rep(i,0,n) ws[x[i]]++;
      rep(i,1,lim) ws[i] += ws[i - 1];
for(int i = n; i--;) sa[--ws[x[y[i]]]] = y[i];
      swap(x, y), p = 1, x[sa[0]] = 0;
      rep(i,1,n) a=sa[i-1], b=sa[i], x[b] =
        (y[a]==y[b] && y[a+j] == y[b+j]) ? p-1
            : p++;
    }
    for (int i=0,j; i<n-1; lcp[x[i++]]=k)
      for (k && k--, j = sa[x[i] - 1];
          s[i + k] == s[j + k]; k++);
  } // loop with no body, wrong indentation
};
```

SuffixTree.h

Description: Ukkonen's algorithm for online suffix tree construction. Each node contains indices [l, r) into the string, and a list of child nodes. Suffixes are given by traversals of this tree, joining [l, r) substrings. The root is 0 (has l = -1, r = 0), non-existent children are -1. To get a complete tree, append a dummy symbol – otherwise it may contain an incomplete path (still useful for substring matching, though).
Time: $\mathcal{O}(26N)$ ca8d4d, 50 lines

```
struct SuffixTree { // N ~ 2*maxlen+10
  enum { N = 200010, ALP = 26 };
  int toi(char c) { return c - 'a'; }
  string a; // v = cur node, q = cur position
int t[N][ALP],l[N],r[N],p[N],s[N],v=0,q=0,m=2;

  void ukkadd(int i, int c) { suff:
    if (r[v]<=q) {
      if (t[v][c]==-1) { t[v][c]=m;  l[m]=i;
        p[m++]=v; v=s[v]; q=r[v]; goto suff; }
      v=t[v][c]; q=l[v];
    }
    if (q==-1 || c==toi(a[q])) q++; else {
      l[m+1]=i;  p[m+1]=m;  l[m]=l[v]; r[m]=q;
      p[m]=p[v];  t[m][c]=m+1;
      t[m][toi(a[q])] = v;
      l[v]=q;  p[v]=m;
      t[p[m]][toi(a[l[m]])] = m;
      v=s[p[m]];  q=l[m];
      while (q<r[m]) { v=t[v][toi(a[q])];  q+=
          r[v]-l[v]; }
      if (q==r[m])  s[m]=v;  else s[m]=m+2;
      q=r[v]-(q-r[m]);  m+=2;  goto suff;
    }
  }
  SuffixTree(string a) : a(a) {
    fill(r,r+N,sz(a));
    memset(s, 0, sizeof s);
    memset(t, -1, sizeof t);
    fill(t[1],t[1]+ALP,0);
    s[0] = 1; l[0] = l[1] = -1; r[0] = r[1] =
        p[0] = p[1] = 0;
    rep(i,0,sz(a)) ukkadd(i, toi(a[i]));
  }
  // example: find longest common substring (
      uses ALP = 28)
  pii best;
  int lcs(int node, int i1, int i2, int olen)
      {
    if (l[node]<=i1 && i1<r[node]) return 1;
    if (l[node]<=i2 && i2<r[node]) return 2;
    int mask = 0, len = node ? olen + (r[node]
        - l[node]) : 0;
    rep(c,0,ALP) if (t[node][c] != -1)
      mask |= lcs(t[node][c], i1, i2, len);
    if (mask == 3)
      best = max(best, {len, r[node] - len});
    return mask;
  }
  static pii LCS(string s, string t) {
    SuffixTree st(s + (char)('z' + 1) + t + (
        char)('z' + 2));
    st.lcs(0, sz(s), sz(s) + 1 + sz(t), 0);
    return st.best;
  }
};
```

SuffixAutomaton.h

Description: Suffix Automaton implementation for string processing. A suffix automaton is a minimal DFA that recognizes all suffixes of a string. Useful for substring queries, pattern matching, and counting unique substrings.
Time: $\mathcal{O}(n)$ for construction, where n is the string length 41c2c3, 159 lines

```
// Suffix Automaton Structure
// len -> largest string length of the
     corresponding endpos-equivalent class
// link -> longest suffix that is another
     endpos-equivalent class.
// firstpos -> 1 indexed end position of the
     first occurrence of the largest string of
     that node
// minlen(v) -> smallest string of node v =
     len(link(v)) + 1
// terminal nodes -> store the suffixes
struct SuffixAutomaton {
  struct node {
    int len, link, firstpos;
    map<char, int> nxt;
  };
  int sz, last;
  vector<node> t;
  vector<int> terminal;
  vector<long long> dp;
  vector<vector<int>> g;

  SuffixAutomaton() {}

  SuffixAutomaton(int n) {
    t.resize(2 * n);
    terminal.resize(2 * n, 0);
    dp.resize(2 * n, -1);
    sz = 1;
    last = 0;
    g.resize(2 * n);
    t[0].len = 0;
    t[0].link = -1;
    t[0].firstpos = 0;
  }

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

```
// Build the suffix link tree
void build_tree() {
  for (int i = 1; i < sz; i++) g[t[i].link].
      push_back(i);
}

// Build the automaton from a string
void build(string &s) {
  for (auto x: s) {
    extend(x);
    terminal[last] = 1;
  }
  build_tree();
}

// Count the number of times i-th node
    occurs in the string
long long cnt(int i) {
  if (dp[i] != -1) return dp[i];
  long long ret = terminal[i];
  for (auto &x: g[i]) ret += cnt(x);
  return dp[i] = ret;
}

// Count the total number of unique
    substrings
long long countUniqueSubstrings() {
  long long ans = 0;
  for (int i = 1; i < sz; i++) {
    ans += t[i].len - t[t[i].link].len;
  }
  return ans;
}

// Check if a pattern exists in the original
    string
bool exists(string &pattern) {
  int cur = 0;
  for (char c : pattern) {
    if (!t[cur].nxt.count(c)) return false;
    cur = t[cur].nxt[c];
  }
  return true;
}

// Find the first occurrence of a pattern in
    the original string
// Returns the end position (1-indexed) or
    -1 if not found
int findFirstOccurrence(string &pattern) {
  int cur = 0;
  for (char c : pattern) {
    if (!t[cur].nxt.count(c)) return -1;
    cur = t[cur].nxt[c];
  }
  // Return the end position of the first
      occurrence
  return t[cur].firstpos;
}

// Count the number of occurrences of a
    pattern in the original string
long long countOccurrences(string &pattern)
    {
  int cur = 0;
  for (char c : pattern) {
    if (!t[cur].nxt.count(c)) return 0;
    cur = t[cur].nxt[c];
  }
  return cnt(cur);
}

// Find the longest common substring with
    another string
string longestCommonSubstring(string &s2) {
  int v = 0, len = 0, best = 0, bestpos = 0;
  for (int i = 0; i < s2.size(); i++) {
    while (v && !t[v].nxt.count(s2[i])) {
      v = t[v].link;
      len = t[v].len;
    }
    if (t[v].nxt.count(s2[i])) {
      v = t[v].nxt[s2[i]];
      len++;
    }
    if (len > best) {
      best = len;
      bestpos = i;
    }
  }
  return s2.substr(bestpos - best + 1, best)
      ;
}
};
```

Hashing.h

Description: Static hashing for 0-indexed string. Intervals are $[l, r]$.

82983c, 20 lines

```
template<const ll M, const ll B>
struct Hashing {
  int n; V<ll> h, pw;
  Hashing(const string &s) : n(sz(s)),h(n+1),
      pw(n+1) {
    pw[0] = 1; // ^^ s is 0 indexed
    for (int i = 1; i <= n; ++i)
      pw[i] = (pw[i-1] * B) % M,
      h[i] = (h[i-1] * B + s[i-1]) % M;
  }
  ll eval(int l, int r) { // assert(l <= r);
    return (h[r+1] - ((h[l] * pw[r-l+1])%M) +
        M)%M;
} };
struct Double_Hash {
  using H1 = Hashing<916969619, 101>;
  using H2 = Hashing<285646799, 103>;
  H1 h1; H2 h2;
  Double_Hash(const string &s):h1(s),h2(s){}
  pii eval(int l, int r)
    { return {h1.eval(l,r), h2.eval(l,r)}; }
};
```

HashingDynamic.h

Description: Hashing with point updates on string (0-indexed). `upd(i, x)`: `s[i] += x`. Intervals are $[l, r]$.

Time: $\mathcal{O}(n \log n)$

c51931, 33 lines

```
template<const ll M, const ll B>
struct Dynamic_Hashing {
  int n; V<ll> h, pw;
  void upd(int pos, int c_add) {
    if (c_add < 0) c_add = (c_add + M) % M;
    for (int i = ++pos; i <= n; i += i&-i)
      h[i] = (h[i]+c_add *1LL* pw[i - pos])%M;
  }
  ll get(int pos, int r = 0) {
    for (int i = ++pos, j = 0; i; i -= i&-i) {
      r = (r + h[i] * 1LL * pw[j]) % M;
      j += i&-i;
    } return r;
  }
  Dynamic_Hashing(const string &s) : n(sz(s)),
      h(n+1), pw(n+1) {
    pw[0] = 1; // ^^ s is 0 indexed
    for (int i = 1; i <= n; ++i) pw[i] = (pw[i
        -1] * 1LL * B) % M;
    for (int i = 0; i < n; ++i) upd(i, s[i]);
  }
  ll eval(int l, int r) { // assert(l <= r);
    return (get(r) - ((get(l-1) * 1LL * pw[r-l
        +1]) % M) + M) % M;
} };
struct Double_Dynamic {
  using DH1 = Dynamic_Hashing<916969619, 571>;
  using DH2 = Dynamic_Hashing<285646799, 953>;
  DH1 h1; DH2 h2;
  Double_Dynamic(const string &s) : h1(s), h2(
      s) {}
  void upd(int pos, int c_add) {
    h1.upd(pos, c_add);
    h2.upd(pos, c_add);
  } pll eval(int l, int r)
    { return {h1.eval(l,r), h2.eval(l,r)}; }
};
```

HashingIstiaque.h

Description: Double hashing with both forward and reverse hash support. Uses random bases and two different moduli for collision resistance. Call pre() before using HashedString.

Time: $\mathcal{O}(n)$ preprocessing, $\mathcal{O}(1)$ query

8b6a96, 44 lines

```
random_device rd;
mt19937 gen(rd());
int range1 = 31, range2 = 1029;
uniform_int_distribution<> distr(range1,
    range2);
const int N = 2e6, M1 = 1e9 + 7, B1 = distr(
    gen), M2 = 998244353, B2 = distr(gen);
vi p1{1}, p2{1};

void pre() {
  for (int i = 1; i <= N; ++i) {
    p1.push_back((p1.back() * B1) % M1);
    p2.push_back((p2.back() * B2) % M2);
  }
}

struct HashedString {
  vi p_hash1, p_hash2, s_hash1, s_hash2;

  HashedString(const string &s) {
    p_hash1.resize(s.size() + 2);
    p_hash2.resize(s.size() + 2);
    s_hash1.resize(s.size() + 2);
    s_hash2.resize(s.size() + 2);
    for (int i = 0; i < s.size(); ++i) {
      p_hash1[i + 1] = ((p_hash1[i] * B1) % M1
          + s[i]) % M1;
      p_hash2[i + 1] = ((p_hash2[i] * B2) % M2
          + s[i]) % M2;
    }
    for (int i = s.size() - 1; i >= 0; --i) {
      s_hash1[i + 1] = ((s_hash1[i + 2] * B1)
          % M1 + s[i]) % M1;
      s_hash2[i + 1] = ((s_hash2[i + 2] * B2)
          % M2 + s[i]) % M2;
    }
  }

  pii get_hash(int start, int end) {
    int raw_val1 = (p_hash1[end + 1] - (
        p_hash1[start] * p1[end - start + 1]))
        ;
    int raw_val2 = (p_hash2[end + 1] - (
        p_hash2[start] * p2[end - start + 1]))
        ;
    return {(raw_val1 % M1 + M1) % M1, (
        raw_val2 % M2 + M2) % M2};
  }

  pii rev_hash(int start, int end) {
    int raw_val1 = (s_hash1[start + 1] - (
        s_hash1[end + 2] * p1[end - start +
        1]));
    int raw_val2 = (s_hash2[start + 1] - (
        s_hash2[end + 2] * p2[end - start +
        1]));
    return {(raw_val1 % M1 + M1) % M1, (
        raw_val2 % M2 + M2) % M2};
  }
};
```

AhoCorasick-arman.h

Usage: `insert strings first (0-indexed). Then call prepare to use everything. link = suffix link. to[ch] = trie transition. jump[ch] = aho transition to ch using links.`

Time: $\mathcal{O}(AL)$

36fabb, 35 lines

```
const int L = 5000; // Total no of characters
const int A = 10; // Alphabet size

struct Aho_Corasick {
  struct Node {
    bool end_flag;
    int par, pch, to[A], link, jump[A];
    Node() {
      par = link = end_flag = 0;
      memset(to, 0, sizeof to);
      memset(jump, 0, sizeof jump);
    }
  }; Node t[L]; int at;
  Aho_Corasick() { at = 0; }

  void insert(string &s) {
    int u = 0;
    for (auto ch : s) {
      int &v = t[u].to[ch - '0'];
      if (!v) v = ++at;
      t[v].par = u; t[v].pch = ch - '0'; u=v;
    } t[u].end_flag = true;
  }
  void prepare() {
    for(queue<int>q({0});!q.empty();q.pop()){
      int u = q.front(), w = t[u].link;
      for (int ch = 0; ch < A; ++ch) {
        int v = t[u].to[ch];
        if (v) {
          t[v].link = t[w].jump[ch];
          q.push(v);
        }
        t[u].jump[ch] = v ? v : t[w].jump[ch];
  } } }
```

```
}aho;
```

Various (10)

10.1 Have you tried?

- **Reading the problem once more?**
- step 1 =¿ think greedy
- step 2 =¿ think dp
- think greedy rather than overthinking!!
- look into constraints to assume complexity
- prefix, suffix, difference array
- key-indexing
- sorting using custom comparator
- binary search
- implementation
- sieve
- reverse way thinking
- divide and conqueror tasks
- power of map and other stl
- combinations(bitmask)
- comparing(min, max)
- gcd/lcm
- upper_bound, lower_bound
- sliding window/two pointers
- multiset/priority queue/stack/ordered set
- rearranging to form equation
- Sample I/O chcek to understand
- handle base case and corner case
- string related- create array of length 26
- parity check
- heavy light
- finding independent subproblem
- think end to start
- pattern finding by dry run
- take relative minimum such as 0
- express in prime factorial
- merge small tasks
- principle of inclusion-exclusion
- inversion (a[i]¿a[j]; i¡j)
- USE BRACKET to avoid PRECEDENCE ISSUE
- small to large marging
- use of XOR hashing/probability tricks
- clear global array for multiple testcases
- check typo, overflow, undefined behavior, additional information from the problem, function without return value, unsigned integer
- Stress testing
- Algorithm:
  - String- Hashing, Trie
  - Range Query: Seg, Mo, Sqrt dec, ordered set
  - Graph: dfs, bfs, dijkstra, Floyd
- combo:
  - seg tree + binary search + path compression
- dp tricks:
  - if bruteforce knapsack isn't optimal change state and redefine statement
  - space optimize by removing unnecessary states
  - digit dp, interval dp, bitmask dp, sos dp, probability dp

Random.h

Description: Nice uniform real/int distribution wrapper

24d233, 9 lines

```
mt19937 rng(chrono::steady_clock::now().
    time_since_epoch().count());
// use mt19937_64 for long long
uniform_int_distribution<int> dist1(lo, hi);
uniform_real_distribution<> dist2(lo, hi);
// Usage
#define rand(l,r) uniform_int_distribution<ll
    >(l, r)(rng_64)
int val = rng(), val3 = dist1(rng);
ll val2 = rng_64(); double val4 = dist2(rng);
shuffle(vec.begin(), vec.end(), rng);
```

10.2 Intervals

IntervalContainer.h

Description: Add and remove intervals from a set of disjoint intervals. Will merge the added interval with any overlapping intervals in the set when adding. Intervals are [inclusive, exclusive).

Time: $\mathcal{O}(\log N)$

edce47, 23 lines

```
set<pii>::iterator addInterval(set<pii>& is,
    int L, int R) {
  if (L == R) return is.end();
  auto it=is.lower_bound({L, R}), before=it;
  while (it != is.end() && it->first <= R) {
    R = max(R, it->second);
    before = it = is.erase(it);
  }
  if (it != is.begin() && (--it)->second >= L)
      {
    L = min(L, it->first);
    R = max(R, it->second);
    is.erase(it);
  }
  return is.insert(before, {L,R});
}

void removeInterval(set<pii>& is,int L,int R){
  if (L == R) return;
  auto it = addInterval(is, L, R);
  auto r2 = it->second;
  if (it->first == L) is.erase(it);
  else (int&)it->second = L;
  if (R != r2) is.emplace(R, r2);
}
```

IntervalCover.h

Description: Compute indices of smallest set of intervals covering another interval. Intervals should be [inclusive, exclusive). To support [inclusive, inclusive], change (A) to add `|| R.empty()`. Returns empty set on failure (or if G is empty).

Time: $\mathcal{O}(N \log N)$

9e9d8d, 18 lines

```
template<class T>
vi cover(pair<T, T> G, vector<pair<T, T>> I) {
  vi S(sz(I)), R;
  iota(all(S), 0); sort(all(S), [&](int a, int
       b) { return I[a] < I[b]; });
  T cur = G.first;
  int at = 0;
  while (cur < G.second) { // (A)
    pair<T, int> mx = make_pair(cur, -1);
    while (at<sz(I) && I[S[at]].first <= cur){
      mx = max(mx, make_pair(I[S[at]].second,
          S[at]));
      at++;
    }
    if (mx.second == -1) return {};
    cur = mx.first;
    R.push_back(mx.second);
  }
  return R;
}
```

ConstantIntervals.h

Description: Split a monotone function on [from, to) into a minimal set of half-open intervals on which it has the same value. Runs a callback g for each such interval.

Usage: `constantIntervals(0, sz(v), [&](int x){return v[x];}, [&](int lo, int hi, T val){...});`

Time: $\mathcal{O}\left(k \log \frac{n}{k}\right)$

753a4c, 19 lines

```
template<class F, class G, class T>
void rec(int from, int to, F& f, G& g, int& i,
     T& p, T q) {
  if (p == q) return;
  if (from == to) {
    g(i, to, p);
    i = to; p = q;
  } else {
    int mid = (from + to) >> 1;
    rec(from, mid, f, g, i, p, f(mid));
    rec(mid+1, to, f, g, i, p, q);
  }
}
template<class F, class G>
void constantIntervals(int from, int to, F f,
    G g) {
  if (to <= from) return;
  int i = from; auto p = f(i), q = f(to-1);
  rec(from, to-1, f, g, i, p, q);
  g(i, to, q);
}
```

10.3 Miscellaneous

LIS.h

Description: Compute indices for the longest increasing subsequence.

Time: $\mathcal{O}(N \log N)$

2932a0, 17 lines

```
template<class I> vi lis(const vector<I>& S) {
  if (S.empty()) return {};
  vi prev(sz(S));
  typedef pair<I, int> p;
  vector<p> res;
  rep(i,0,sz(S)) {
    // change 0 -> i for longest non-
        decreasing subsequence
    auto it = lower_bound(all(res), p{S[i], 0}
        );
    if (it == res.end()) res.emplace_back(),
        it = res.end()-1;
    *it = {S[i], i};
    prev[i] = it == res.begin() ? 0 : (it-1)->
        second;
  }
  int L = sz(res), cur = res.back().second;
  vi ans(L);
  while (L--) ans[L] = cur, cur = prev[cur];
  return ans;
}
```

FastKnapsack.h

Description: Given N non-negative integer weights w and a non-negative target t, computes the maximum S <= t such that S is the sum of some subset of the weights.

Time: $\mathcal{O}(N \max(w_i))$

b20ccc, 16 lines

```
int knapsack(vi w, int t) {
  int a = 0, b = 0, x;
  while (b < sz(w) && a + w[b] <= t) a += w[b
      ++];
  if (b == sz(w)) return a;
  int m = *max_element(all(w));
  vi u, v(2*m, -1);
  v[a+m-t] = b;
  rep(i,b,sz(w)) {
    u = v;
    rep(x,0,m) v[x+w[i]] = max(v[x+w[i]], u[x
        ]);
    for (x = 2*m; --x > m;) rep(j, max(0,u[x])
        , v[x])
      v[x-w[j]] = max(v[x-w[j]], j);
  }
  for (a = t; v[a+m-t] < 0; a--) ;
  return a;
}
```

But actually we can do better. For the knapsack problem, it has at most n items, and the sum of weights of all items is also at most n. We can do a sqrt decomposition trick here.

- For items with weight $\geq \sqrt{n}$, there are at most $\sqrt{n}$ such items.
- For items with weight $< \sqrt{n}$, we count the number of items for each different weight. If there are c_w items for weight w, we decompose c_w into

$$c_w = 2^0 + 2^1 + \cdots + 2^k + y$$

where k is the largest integer satisfying $2^0 + 2^1 + \cdots + 2^k \le c_w$. Then we create new items with weights

$$2^0 \cdot w,\ 2^1 \cdot w,\ \ldots,\ 2^k \cdot w,\ y \cdot w.$$

The set of new items is the same as c_w items with weight w if we only consider the different sum of weights the set of items can achieve. Now we only have

$$\sum_{w=1}^{\sqrt{n}} \log(c_w) = \sqrt{n} \text{ items.}$$

The total time complexity is

$$O(\sqrt{n} \cdot n + \sqrt{n} \cdot n) = O(n\sqrt{n}).$$

Triplet.h

Description: A triplet struct with operator overloading for +, =, and ==.
Usage: triplet t(1, 2, 3); triplet t2 = t + triplet(4, 5, 6);
Time: $\mathcal{O}(1)$ 0a3770, 14 lines

```
struct triplet {
  int a, b, c;
  triplet(int a = 0, int b = 0, int c = 0) : a
      (a), b(b), c(c) {}
  triplet &operator=(const triplet &obj) {
    a = obj.a, b = obj.b;
    return *this;
  }
  triplet operator+(const triplet &obj) const
      {
    return triplet(a + obj.a, b + obj.b, c +
        obj.c);
  }
  bool operator==(const triplet &obj) const {
    return (a == obj.a && b == obj.b);
  }
};
```

Comparators.h

Description: Custom comparators for priority queues, sets, and sorting.
Usage: See code examples below
Time: $\mathcal{O}(nlogn)$ a08c7a, 6 lines

```
struct cmp {
    bool operator()(const pair<int, int> &i,
        const pair<int, int> &j) const {
        return i.second < j.second;
    }
};
// sort(v.begin(), v.end(), [&](auto i, auto j
    ) -> bool {return i[1] < j[1];});
```

CoordinateCompression.h

Description: Coordinate compression to map values to a continuous range [0...N-1].
Usage: vector<int> a = {10, 100, 5, 1000};
// After compression: a = {1, 2, 0, 3}
Time: $\mathcal{O}(nlogn)$ c38da4, 6 lines

```
vector<int> a = v;
map<T, int> mp;
int cnt = 0;
for (auto &it : a) mp[it];
for (auto &it : mp) it.second = cnt++;
for (auto &it : a) it = mp[it];
```

VectorOps.h

Description: Common vector operations like removing duplicates.
Usage: vector<int> v = {3, 1, 4, 1, 5, 9};
sortAndUnique(v); // v = {1, 3, 4, 5, 9}
Time: $\mathcal{O}(nlogn)$ for sorting, $\mathcal{O}(n)$ for unique 2ea4d5, 2 lines

```
sort(v.begin(), v.end());
v.erase(unique(v.begin(), v.end()), v.end());
```

BitOperations.h

Description: Common bit manipulation operations and bitset usage.
Usage: See examples below
Time: $\mathcal{O}(1)$ for most operations 100cd5, 28 lines

```
//Bitwise operations
num |= (1 << pos); //Set bit at position pos
num &= (~(1 << pos)); //Unset bit at position
    pos
num ^= (1 << pos); //Toggle bit at position
    pos
int ret = num & (-num); //Extract lowest set
    bit
bool bit = num & (1 << pos); //Check if bit at
    position pos is set
int ones = __builtin_popcountll(num); //Count
    set bits (long long)
int tz = __builtin_ctzll(num); //Count
    trailing zeros (long long)
int lz = __builtin_clzll(num); //Count leading
    zeros (long long)

//Bitset operations
bitset<32> bs; //Create a bitset of size 32 (
    all 0s)
bitset<32> bs(42); //From decimal number
    (101010)
bitset<32> bs("101010"); //From binary string
bs.set(i);  //Set bit at position i to 1
bs.reset(i); //Set bit at position i to 0
bs.flip(i); //Flip bit at position i
int count = bs.count(); //Count number of set
    bits
bool test = bs.test(i); //Check if bit at
    position i is set
bool any = bs.any(); //Check if any bit is set
bool none = bs.none(); //Check if no bit is
    set
bool all = bs.all(); //Check if all bits are
    set
size_t size = bs.size(); //Get size of bitset
string s = bs.to_string(); //Convert to string
unsigned long ul = bs.to_ulong(); //Convert to
    unsigned long
unsigned long long ull = bs.to_ullong(); //
    Convert to unsigned long long
size_t first = bs._Find_first(); //Position of
    first set bit
size_t next = bs._Find_next(i); //Position of
    first set bit after i
```

ArrayOps.h

Description: Common operations for arrays and multi-dimensional arrays.
Usage: See examples below
Time: $\mathcal{O}(n)$ where n is the number of elements d41d8c, 2 lines

```
// fill_n(&memo[0][0][0][0], sizeof memo /
    sizeof(int), INF);
// memset(memo, 0x3f, sizeof memo); //Set all
    ints to 0x3f3f3f3f (a large value ~1B)
```

MatrixOperations.h

Description: Rotate a matrix by 90 degrees clockwise.
Usage: See examples below
Time: $\mathcal{O}(n*n)$ for an n x n matrix ee9e27, 4 lines

```
ans[j][n + 1 - i] = v[i][j];
ans[n + 1 - i][n + 1 - j] = v[j][n + 1 - i];
ans[n + 1 - j][i] = v[n + 1 - i][n + 1 - j];
ans[i][j] = v[n + 1 - j][i];
```

SubsetFormulas.h

Description: Useful formulas related to subsets and combinations.
Usage: See examples below d41d8c, 12 lines

```
// Summation of product of all subsets formula
    :
// For a set of numbers {a, b, c, ...}, the
    sum of products of all possible subsets is
    :
// (a+1)*(b+1)*(c+1)*... - 1
//
// Example:
// Set = {2, 3, 4}
// Subsets: {}, {2}, {3}, {4}, {2,3}, {2,4}, {
    3,4}, {2,3,4}
// Products: 1, 2, 3, 4, 2*3=6, 2*4=8, 3*4=12,
    2*3*4=24
// Sum: 1 + 2 + 3 + 4 + 6 + 8 + 12 + 24 = 60
// Using formula: (2+1)*(3+1)*(4+1) - 1 =
    3*4*5 - 1 = 60 - 1 = 59 + 1 = 60 (correct)
//
// Note: The empty subset {} has a product of
    1 (the multiplicative identity)
```

10.4 Formulas

Arithmetic Sequence:

$$a_n = a + (n-1)d$$

Sum of the First n Terms of an Arithmetic Series:

$$S_n = \frac{n}{2}\left[2a + (n-1)d\right]$$

Geometric Sequence:

$$a_n = ar^{(n-1)}$$

Sum of the First n Terms of a Geometric Series:

$$S_n = \frac{a(1-r^n)}{1-r}$$

Sum of Infinite Terms of a Geometric Series (when $|r| < 1$):

$$S_\infty = \frac{a}{1-r}$$

10.5 Dynamic programming

KnuthDP.h

Description: When doing DP on intervals: $a[i][j] = \min_{i<k<j}(a[i][k] + a[k][j]) + f(i,j)$, where the (minimal) optimal k increases with both i and j, one can solve intervals in increasing order of length, and search $k = p[i][j]$ for $a[i][j]$ only between $p[i][j-1]$ and $p[i+1][j]$. This is known as Knuth DP. Sufficient criteria for this are if $f(b,c) \le f(a,d)$ and $f(a,c) + f(b,d) \le f(a,d) + f(b,c)$ for all $a \le b \le c \le d$. Consider also: LineContainer (ch. Data structures), monotone queues, ternary search.
Time: $\mathcal{O}(N^2)$

DivideAndConquerDP.h

Description: Given $a[i] = \min_{lo(i) \le k < hi(i)}(f(i,k))$ where the (minimal) optimal k increases with i, computes $a[i]$ for $i = L..R-1$.
Time: $\mathcal{O}((N + (hi - lo)) \log N)$ d38d2b, 18 lines

```
struct DP { // Modify at will:
  int lo(int ind) { return 0; }
  int hi(int ind) { return ind; }
  ll f(int ind, int k) { return dp[ind][k]; }
  void store(int ind, int k, ll v) { res[ind]
      = pii(k, v); }

  void rec(int L, int R, int LO, int HI) {
    if (L >= R) return;
    int mid = (L + R) >> 1;
    pair<ll, int> best(LLONG_MAX, LO);
    rep(k, max(LO,lo(mid)), min(HI,hi(mid)))
      best = min(best, make_pair(f(mid, k), k)
          );
    store(mid, best.second, best.first);
    rec(L, mid, LO, best.second+1);
    rec(mid+1, R, best.second, HI);
  }
  void solve(int L, int R) { rec(L, R, INT_MIN
      , INT_MAX); }
};
```

SOSDP.h

Description: Sum Over Subsets (SOS) DP implementation. Technique for efficiently calculating functions over subsets or supersets of bitmasks.
Time: $\mathcal{O}\left(N*2^N\right)$ for preprocessing, $\mathcal{O}(1)$ for queries a54741, 49 lines

```
const int B = 20; // Maximum number of bits (
    adjust as needed)

// Precomputes sum over all subsets of masks
// After this, f[mask] = sum of f[submask] for
    all submask <= mask
void precomputeSubsetSums(vector<int>& f, int
    bits = B) {
  for (int i = 0; i < bits; i++) {
    for (int mask = 0; mask < (1 << bits);
        mask++) {
      if (mask & (1 << i)) {
        f[mask] += f[mask ^ (1 << i)];
      }
    }
  }
}

// Precomputes sum over all supersets of masks
```

```cpp
// After this, g[mask] = sum of g[supermask] for all mask <= supermask
void precomputeSupersetSums(vector<int>& g, int bits = B) {
  for (int i = 0; i < bits; i++) {
    for (int mask = (1 << bits) - 1; mask >= 0; mask--) {
      if ((mask & (1 << i)) == 0) {
        g[mask] += g[mask ^ (1 << i)];
      }
    }
  }
}

// Alternative implementation with better constant factor
// Precomputes sum over all subsets of masks in O(N * 2^N)
void fastSubsetSums(vector<int>& f, int bits = B) {
  for (int i = 0; i < bits; i++) {
    for (int mask = 0; mask < (1 << bits); mask++) {
      if ((mask & (1 << i)) == 0) {
        f[mask | (1 << i)] += f[mask];
      }
    }
  }
}

// Alternative implementation with better constant factor
// Precomputes sum over all supersets of masks in O(N * 2^N)
void fastSupersetSums(vector<int>& g, int bits = B) {
  for (int i = 0; i < bits; i++) {
    for (int mask = 0; mask < (1 << bits); mask++) {
      if (mask & (1 << i)) {
        g[mask ^ (1 << i)] += g[mask];
      }
    }
  }
}
```

DPPatterns.h

Description: Comprehensive Dynamic Programming Problem Types and Patterns.

442530, 1090 lines

```cpp
/*
COMPREHENSIVE DYNAMIC PROGRAMMING PATTERNS WITH CODE

1. KNAPSACK VARIATIONS:
*/

// a) 0/1 Knapsack: dp[i][w] = max value using first i items with weight <= w
// Time: O(n*W), Space: O(n*W)
int knapsack01(vector<int>& weights, vector<int>& values, int W) {
  int n = weights.size();
  vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));
  for (int i = 1; i <= n; i++) {
    for (int w = 1; w <= W; w++) {
      dp[i][w] = dp[i-1][w]; // don't take item i-1
      if (weights[i-1] <= w) {
        dp[i][w] = max(dp[i][w], dp[i-1][w-weights[i-1]] + values[i-1]);
      }
    }
  }
  return dp[n][W];
}

// b) Unbounded Knapsack: unlimited items, dp[w] = max value with weight <= w
// Time: O(n*W), Space: O(W)
int unboundedKnapsack(vector<int>& weights, vector<int>& values, int W) {
  vector<int> dp(W + 1, 0);
  for (int w = 1; w <= W; w++) {
    for (int i = 0; i < weights.size(); i++) {
      if (weights[i] <= w) {
        dp[w] = max(dp[w], dp[w - weights[i]] + values[i]);
      }
    }
  }
  return dp[W];
}

// c) Bounded Knapsack: each item has limited copies, uses binary splitting
// Time: O(W * sum(log counts)), Space: O(W)
int boundedKnapsack(vector<int>& weights, vector<int>& values, vector<int>& counts, int W) {
  vector<int> dp(W + 1, 0);

  for (int i = 0; i < weights.size(); i++) {
    for (int k = 1; k <= counts[i]; k *= 2) {
      int take = min(k, counts[i]);
      int w = weights[i] * take;
      int v = values[i] * take;

      for (int j = W; j >= w; j--) {
        dp[j] = max(dp[j], dp[j - w] + v);
      }
      counts[i] -= take;
    }
  }
  return dp[W];
}

// d) Subset Sum (Boolean): dp[s] = true if sum s is achievable
// Time: O(n*target), Space: O(target)
bool subsetSum(vector<int>& nums, int target) {
  vector<bool> dp(target + 1, false);
  dp[0] = true;

  for (int num : nums) {
    for (int j = target; j >= num; j--) {
      dp[j] = dp[j] || dp[j - num];
    }
  }
  return dp[target];
}

// e) Subset Sum with Bitset: ultra-fast using bitshift operations
// Time: O(n*target/64), Space: O(target)
bool subsetSumBitset(vector<int>& nums, int target) {
  bitset<100001> bs;
  bs[0] = 1;
  for (int num : nums) {
    bs |= (bs << num);
  }
  return bs[target];
}

// f) Partition Equal Subset Sum: can split array into two equal-sum parts
// Time: O(n*sum), Space: O(sum)
bool canPartition(vector<int>& nums) {
  int sum = 0;
  for (int num : nums) sum += num;
  if (sum % 2 == 1) return false;
  return subsetSum(nums, sum / 2);
}

// g) Value-Optimized Knapsack: when W is large but sum of values is small
// Time: O(n*sum_values), Space: O(sum_values)
int valueOptimizedKnapsack(vector<int>& weights, vector<int>& values, int W) {
  int maxValue = 0;
  for (int v : values) maxValue += v;

  vector<int> dp(maxValue + 1, INT_MAX);
  dp[0] = 0;

  for (int i = 0; i < weights.size(); i++) {
    for (int v = maxValue; v >= values[i]; v--) {
      if (dp[v - values[i]] != INT_MAX) {
        dp[v] = min(dp[v], dp[v - values[i]] + weights[i]);
      }
    }
  }

  for (int v = maxValue; v >= 0; v--) {
    if (dp[v] <= W) return v;
  }
  return 0;
}

// h) Meet-in-the-Middle Knapsack: exponential split for n~40
// Time: O(2^(n/2) * log(2^(n/2))), Space: O(2^(n/2))
int meetInMiddleKnapsack(vector<int>& weights, vector<int>& values, int W) {
  int n = weights.size();
  int n1 = n / 2, n2 = n - n1;

  // Generate all possible sums for first half
  vector<pair<int, int>> first; // {weight, value}
  for (int mask = 0; mask < (1 << n1); mask++) {
    int w = 0, v = 0;
    for (int i = 0; i < n1; i++) {
      if (mask & (1 << i)) {
        w += weights[i];
        v += values[i];
      }
    }
    if (w <= W) first.push_back({w, v});
  }

  // Sort by weight and keep only pareto optimal (max value for each weight)
  sort(first.begin(), first.end());
  vector<pair<int, int>> optimal;
  int maxVal = 0;
  for (auto& p : first) {
    if (p.second > maxVal) {
      maxVal = p.second;
      optimal.push_back(p);
    }
  }

  int result = 0;
  // Generate all possible sums for second half
  for (int mask = 0; mask < (1 << n2); mask++) {
    int w2 = 0, v2 = 0;
    for (int i = 0; i < n2; i++) {
      if (mask & (1 << i)) {
        w2 += weights[n1 + i];
        v2 += values[n1 + i];
      }
    }
    if (w2 <= W) {
      // Binary search for best match from first half
      int remaining = W - w2;
      auto it = upper_bound(optimal.begin(), optimal.end(),
                            make_pair(remaining, INT_MAX));
      if (it != optimal.begin()) {
        --it;
        result = max(result, v2 + it->second);
      }
    }
  }
  return result;
}

/*
2. COIN CHANGE VARIATIONS:
*/

// a) Min Coins: dp[amount] = min coins to make amount
// Time: O(n*amount), Space: O(amount)
int coinChange(vector<int>& coins, int amount) {
  vector<int> dp(amount + 1, INT_MAX);
  dp[0] = 0;
  for (int coin : coins) {
    for (int i = coin; i <= amount; i++) {
      if (dp[i - coin] != INT_MAX) {
        dp[i] = min(dp[i], dp[i - coin] + 1);
      }
    }
  }
  return dp[amount] == INT_MAX ? -1 : dp[amount];
}

// b) Coin Combinations: dp[amount] = ways to make amount (order doesn't matter)
// Time: O(n*amount), Space: O(amount)
```

```cpp
int coinCombinations(vector<int>& coins, int
    amount) {
  vector<int> dp(amount + 1, 0);
  dp[0] = 1;
  for (int coin : coins) {          // COIN
      OUTER -> combinations
    for (int i = coin; i <= amount; i++) {
      dp[i] += dp[i - coin];
    }
  }
  return dp[amount];
}

// c) Coin Permutations: dp[amount] = ways to
    make amount (order matters)
// Time: O(n*amount), Space: O(amount)
int coinPermutations(vector<int>& coins, int
    amount) {
  vector<int> dp(amount + 1, 0);
  dp[0] = 1;
  for (int i = 1; i <= amount; i++) {    //
      AMOUNT OUTER -> permutations
    for (int coin : coins) {
      if (coin <= i) {
        dp[i] += dp[i - coin];
      }
    }
  }
  return dp[amount];
}

// d) Bounded Coin Change: each coin type has
    limited quantity
// Time: O(amount * sum(log counts)), Space: O
    (amount)
int boundedCoinChange(vector<int>& coins,
    vector<int>& counts, int amount) {
  vector<int> dp(amount + 1, 0);
  dp[0] = 1;

  for (int i = 0; i < coins.size(); i++) {
    int coin = coins[i];
    int count = counts[i];

    // Binary splitting for efficiency
    for (int k = 1; k <= count; k *= 2) {
      int use = min(k, count);
      int value = coin * use;

      for (int j = amount; j >= value; j--) {
        dp[j] += dp[j - value];
      }
      count -= use;
    }

    if (count > 0) {
      int value = coin * count;
      for (int j = amount; j >= value; j--) {
        dp[j] += dp[j - value];
      }
    }
  }
  return dp[amount];
}

// e) Count subsets with exact sum: dp[s] =
    count of subsets with sum s
// Time: O(n*target), Space: O(target)
int countSubsetsWithSum(vector<int>& nums, int
    target, int mod) {
  vector<int> dp(target + 1, 0);
  dp[0] = 1;

  for (int num : nums) {
    for (int j = target; j >= num; j--) {
      dp[j] = (dp[j] + dp[j - num]) % mod;
    }
  }
  return dp[target];
}

/*
SPECIALIZED KNAPSACK VARIANTS:
*/

// Tree Knapsack: items have parent-child
    dependency
class TreeKnapsack {
  struct Node {
    int weight, value;
    vector<int> children;
  };

  vector<Node> tree;

  vector<int> dfs(int node, int W) {
    vector<int> dp(W + 1, 0);

    // If we take this node
    if (tree[node].weight <= W) {
      vector<int> combined(W - tree[node].
          weight + 1, tree[node].value);

      for (int child : tree[node].children) {
        vector<int> childDP = dfs(child, W -
            tree[node].weight);
        vector<int> newCombined(W - tree[node
            ].weight + 1, 0);

        // Merge knapsack
        for (int i = 0; i <= W - tree[node].
            weight; i++) {
          for (int j = 0; j <= W - tree[node].
              weight - i; j++) {
            newCombined[i + j] = max(
                newCombined[i + j], combined[i
                ] + childDP[j]);
          }
        }
        combined = newCombined;
      }

      for (int i = tree[node].weight; i <= W;
          i++) {
        dp[i] = max(dp[i], combined[i - tree[
            node].weight]);
      }
    }

    // Don't take this node (but can take
        subtrees)
    for (int child : tree[node].children) {
      vector<int> childDP = dfs(child, W);
      for (int i = 0; i <= W; i++) {
        dp[i] = max(dp[i], childDP[i]);
      }
    }
    return dp;
  }

public:
  int solve(int root, int W) {
    vector<int> result = dfs(root, W);
    return result[W];
  }
};

// 2D Knapsack: two constraints (weight and
    volume)
int knapsack2D(vector<int>& weights, vector<
    int>& volumes, vector<int>& values, int W,
    int V) {
  vector<vector<int>> dp(W + 1, vector<int>(V
      + 1, 0));

  for (int i = 0; i < weights.size(); i++) {
    for (int w = W; w >= weights[i]; w--) {
      for (int v = V; v >= volumes[i]; v--) {
        dp[w][v] = max(dp[w][v], dp[w -
            weights[i]][v - volumes[i]] +
            values[i]);
      }
    }
  }
  return dp[W][V];
}

// Knapsack with item dependencies (general
    DAG)
int knapsackDAG(vector<int>& weights, vector<
    int>& values, vector<vector<int>>& prereq,
    int W) {
  int n = weights.size();
  vector<int> indegree(n, 0);
  vector<vector<int>> adj(n);

  for (int i = 0; i < n; i++) {
    for (int dep : prereq[i]) {
      adj[dep].push_back(i);
      indegree[i]++;
    }
  }

  // Topological sort
  queue<int> q;
  for (int i = 0; i < n; i++) {
    if (indegree[i] == 0) q.push(i);
  }

  vector<int> topo;
  while (!q.empty()) {
    int u = q.front(); q.pop();
    topo.push_back(u);
    for (int v : adj[u]) {
      if (--indegree[v] == 0) {
        q.push(v);
      }
    }
  }

  // DP in topological order
  vector<vector<int>> dp(n, vector<int>(W + 1,
      0));

  for (int item : topo) {
    for (int w = 0; w <= W; w++) {
      // Don't take item
      dp[item][w] = 0;
      for (int dep : prereq[item]) {
        dp[item][w] = max(dp[item][w], dp[dep
            ][w]);
      }

      // Take item (if weight allows and all
          prerequisites satisfied)
      if (w >= weights[item]) {
        int minFromDeps = 0;
        for (int dep : prereq[item]) {
          minFromDeps = max(minFromDeps, dp[
              dep][w - weights[item]]);
        }
        dp[item][w] = max(dp[item][w],
            minFromDeps + values[item]);
      }
    }
  }

  int result = 0;
  for (int i = 0; i < n; i++) {
    result = max(result, dp[i][W]);
  }
  return result;
}

/*
KNAPSACK TEMPLATES & MISTAKE PREVENTION:
*/

// Clean 0/1 Knapsack (competitive style): dp[
    w] = max value with weight <= w
// Time: O(n*W), Space: O(W)
int knapsack01Clean(int n, int W, vector<int>&
     wt, vector<int>& val) {
  vector<int> dp(W + 1, 0);
  for (int i = 0; i < n; ++i) {
    for (int w = W; w >= wt[i]; --w) {    //
        REVERSE for 0/1
      dp[w] = max(dp[w], dp[w - wt[i]] + val[i
          ]);
    }
  }
  return dp[W];
}

// Unbounded/Coin Change clean template: dp[
    amount] = ways to make amount
// Time: O(n*amount), Space: O(amount)
int coinChangeCount(vector<int>& coins, int
    amount) {
  vector<int> dp(amount + 1, 0);
  dp[0] = 1;
  for (int coin : coins) {              //
      coin outer -> combinations
    for (int x = coin; x <= amount; ++x) {
        // FORWARD for unbounded
      dp[x] += dp[x - coin];
    }
  }
  return dp[amount];
}

// Min coins template: dp[amount] = minimum
    coins needed for amount
// Time: O(n*amount), Space: O(amount)
```

```cpp
int minCoins(vector<int>& coins, int amount) {
  const int INF = 1e9;
  vector<int> dp(amount + 1, INF);
  dp[0] = 0;
  for (int coin : coins) {
    for (int x = coin; x <= amount; ++x) {
      dp[x] = min(dp[x], dp[x - coin] + 1);
    }
  }
  return dp[amount] >= INF ? -1 : dp[amount];
}

/*
CRITICAL REMINDERS:
0/1 knapsack: REVERSE loop (prevents reuse)
Unbounded: FORWARD loop (allows reuse)
Combinations: coin outer, Permutations: amount outer
Always dp[0] = 0 for max/min, dp[0] = 1 for counting
*/

/*
3. LONGEST COMMON SUBSEQUENCE (LCS):
*/

// a) Standard LCS: dp[i][j] = LCS length of first i chars of s1, first j chars of s2
// Time: O(m*n), Space: O(m*n)
int LCS(string s1, string s2) {
  int m = s1.length(), n = s2.length();
  vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
      if (s1[i-1] == s2[j-1]) {
        dp[i][j] = dp[i-1][j-1] + 1;
      } else {
        dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
      }
    }
  }
  return dp[m][n];
}

// d) Edit Distance (Levenshtein): min operations to transform s1 to s2
// Time: O(m*n), Space: O(m*n)
int editDistance(string s1, string s2) {
  int m = s1.length(), n = s2.length();
  vector<vector<int>> dp(m + 1, vector<int>(n + 1));
  for (int i = 0; i <= m; i++) dp[i][0] = i;
  for (int j = 0; j <= n; j++) dp[0][j] = j;

  for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
      if (s1[i-1] == s2[j-1]) {
        dp[i][j] = dp[i-1][j-1];
      } else {
        dp[i][j] = 1 + min({dp[i-1][j], dp[i][j-1], dp[i-1][j-1]});
      }
    }
  }
  return dp[m][n];
}
```

```cpp
/*
4. LONGEST INCREASING SUBSEQUENCE (LIS):
*/

// a) Standard LIS: dp[i] = length of LIS ending at position i
// Time: O(n^2), Space: O(n)
int LIS(vector<int>& nums) {
  int n = nums.size();
  vector<int> dp(n, 1);
  int maxLen = 1;
  for (int i = 1; i < n; i++) {
    for (int j = 0; j < i; j++) {
      if (nums[j] < nums[i]) {
        dp[i] = max(dp[i], dp[j] + 1);
      }
    }
    maxLen = max(maxLen, dp[i]);
  }
  return maxLen;
}

// b) LIS using binary search: maintains array of smallest tail for each length
// Time: O(n log n), Space: O(n)
int LIS_fast(vector<int>& nums) {
  vector<int> lis;
  for (int num : nums) {
    auto it = lower_bound(lis.begin(), lis.end(), num);
    if (it == lis.end()) {
      lis.push_back(num);
    } else {
      *it = num;
    }
  }
  return lis.size();
}

/*
5. PATH PROBLEMS:
*/

// a) Unique Paths: dp[i][j] = number of ways to reach position (i,j)
// Time: O(m*n), Space: O(m*n)
int uniquePaths(int m, int n) {
  vector<vector<int>> dp(m, vector<int>(n, 1));
  for (int i = 1; i < m; i++) {
    for (int j = 1; j < n; j++) {
      dp[i][j] = dp[i-1][j] + dp[i][j-1];
    }
  }
  return dp[m-1][n-1];
}

// b) Min Path Sum: dp[i][j] = minimum sum to reach position (i,j)
// Time: O(m*n), Space: O(m*n)
int minPathSum(vector<vector<int>>& grid) {
  int m = grid.size(), n = grid[0].size();
  vector<vector<int>> dp(m, vector<int>(n));
  dp[0][0] = grid[0][0];

  for (int i = 1; i < m; i++) dp[i][0] = dp[i-1][0] + grid[i][0];
  for (int j = 1; j < n; j++) dp[0][j] = dp[0][j-1] + grid[0][j];
  for (int i = 1; i < m; i++) {
    for (int j = 1; j < n; j++) {
      dp[i][j] = grid[i][j] + min(dp[i-1][j], dp[i][j-1]);
    }
  }
  return dp[m-1][n-1];
}

/*
6. SUBSTRING/SUBARRAY PROBLEMS:
*/

// a) Max Subarray Sum (Kadane's): tracks current and maximum sum ending at each position
// Time: O(n), Space: O(1)
int maxSubArray(vector<int>& nums) {
  int maxSum = nums[0], currentSum = nums[0];
  for (int i = 1; i < nums.size(); i++) {
    currentSum = max(nums[i], currentSum + nums[i]);
    maxSum = max(maxSum, currentSum);
  }
  return maxSum;
}

// b) Max Product Subarray: tracks both max and min product ending at each position
// Time: O(n), Space: O(1)
int maxProduct(vector<int>& nums) {
  int maxProd = nums[0], minProd = nums[0], result = nums[0];
  for (int i = 1; i < nums.size(); i++) {
    if (nums[i] < 0) swap(maxProd, minProd);
    maxProd = max(nums[i], maxProd * nums[i]);
    minProd = min(nums[i], minProd * nums[i]);
    result = max(result, maxProd);
  }
  return result;
}

// c) Longest Palindromic Substring: dp[i][j] = true if substring s[i..j] is palindrome
// Time: O(n^2), Space: O(n^2)
string longestPalindrome(string s) {
  int n = s.length();
  vector<vector<bool>> dp(n, vector<bool>(n, false));
  int start = 0, maxLen = 1;

  // All single characters are palindromes
  for (int i = 0; i < n; i++) dp[i][i] = true;

  // Check for palindromes of length 2
  for (int i = 0; i < n - 1; i++) {
    if (s[i] == s[i + 1]) {
      dp[i][i + 1] = true;
      start = i;
      maxLen = 2;
    }
  }

  // Check for palindromes of length 3 and more
  for (int len = 3; len <= n; len++) {
    for (int i = 0; i < n - len + 1; i++) {
      int j = i + len - 1;
      if (s[i] == s[j] && dp[i + 1][j - 1]) {
        dp[i][j] = true;
        start = i;
        maxLen = len;
      }
    }
  }
  return s.substr(start, maxLen);
}

/*
7. TREE DP EXAMPLES:
*/

// Tree Diameter using DFS
class TreeDP {
  vector<vector<int>> adj;
  int maxDist = 0;

  int dfs(int node, int parent) {
    int max1 = 0, max2 = 0;
    for (int child : adj[node]) {
      if (child != parent) {
        int depth = dfs(child, node);
        if (depth > max1) {
          max2 = max1;
          max1 = depth;
        } else if (depth > max2) {
          max2 = depth;
        }
      }
    }
    maxDist = max(maxDist, max1 + max2);
    return max1 + 1;
  }

public:
  int treeDiameter(int n, vector<vector<int>>& edges) {
    adj.resize(n);
    for (auto& edge : edges) {
      adj[edge[0]].push_back(edge[1]);
      adj[edge[1]].push_back(edge[0]);
    }
    dfs(0, -1);
    return maxDist;
  }
};

/*
8. GAME THEORY DP:
*/

// Stone Game - optimal play: dp[i][j] = score difference for player 1 in range [i,j]
// Time: O(n^2), Space: O(n^2)
bool stoneGame(vector<int>& piles) {
  int n = piles.size();
  vector<vector<int>> dp(n, vector<int>(n, 0));

  for (int i = 0; i < n; i++) dp[i][i] = piles[i];

  for (int len = 2; len <= n; len++) {
    for (int i = 0; i <= n - len; i++) {
      int j = i + len - 1;
      dp[i][j] = max(piles[i] - dp[i+1][j], piles[j] - dp[i][j-1]);
```

```
    }
  }
  return dp[0][n-1] > 0;
}

/*
9. INTERVAL DP:
*/

// Matrix Chain Multiplication: dp[i][j] = min
    cost to multiply matrices i to j
// Time: O(n^3), Space: O(n^2)
int matrixChainMultiplication(vector<int>&
    dims) {
  int n = dims.size() - 1;
  vector<vector<int>> dp(n, vector<int>(n, 0))
      ;

  for (int len = 2; len <= n; len++) {
    for (int i = 0; i <= n - len; i++) {
      int j = i + len - 1;
      dp[i][j] = INT_MAX;
      for (int k = i; k < j; k++) {
        int cost = dp[i][k] + dp[k+1][j] +
            dims[i] * dims[k+1] * dims[j+1];
        dp[i][j] = min(dp[i][j], cost);
      }
    }
  }
  return dp[0][n-1];
}

/*
10. DIGIT DP:
*/

// Count numbers with digit sum equal to
    target: digit DP with tight bound
// Time: O(n * target * 2), Space: O(n *
    target * 2)
int digitDP(string num, int target) {
  int n = num.length();
  vector<vector<vector<int>>> memo(n, vector<
      vector<int>>(target + 1, vector<int>(2,
      -1)));

  function<int(int, int, bool)> solve = [&](
      int pos, int sum, bool tight) -> int {
    if (pos == n) return sum == target ? 1 :
        0;
    if (memo[pos][sum][tight] != -1) return
        memo[pos][sum][tight];

    int limit = tight ? (num[pos] - '0') : 9;
    int result = 0;

    for (int digit = 0; digit <= limit; digit
        ++) {
      if (sum + digit <= target) {
        result += solve(pos + 1, sum + digit,
            tight && (digit == limit));
      }
    }
    return memo[pos][sum][tight] = result;
  };

  return solve(0, 0, true);
}
```

```
/*
11. BITMASK DP:
*/

// Traveling Salesman Problem: dp[mask][i] =
    min cost visiting cities in mask, ending
    at i
// Time: O(n^2 * 2^n), Space: O(n * 2^n)
int TSP(vector<vector<int>>& dist) {
  int n = dist.size();
  vector<vector<int>> dp(1 << n, vector<int>(n
      , INT_MAX));
  dp[1][0] = 0; // Start from city 0

  for (int mask = 1; mask < (1 << n); mask++)
      {
    for (int u = 0; u < n; u++) {
      if (!(mask & (1 << u)) || dp[mask][u] ==
          INT_MAX) continue;
      for (int v = 0; v < n; v++) {
        if (mask & (1 << v)) continue;
        int newMask = mask | (1 << v);
        dp[newMask][v] = min(dp[newMask][v],
            dp[mask][u] + dist[u][v]);
      }
    }
  }

  int result = INT_MAX;
  for (int i = 1; i < n; i++) {
    result = min(result, dp[(1 << n) - 1][i] +
        dist[i][0]);
  }
  return result;
}

/*
12. STATE MACHINE DP:
*/

// Stock Trading with Cooldown: state machine
    DP with buy/sell/rest states
// Time: O(n), Space: O(n)
int stockWithCooldown(vector<int>& prices) {
  int n = prices.size();
  if (n <= 1) return 0;

  vector<int> buy(n), sell(n), rest(n);
  buy[0] = -prices[0];
  sell[0] = 0;
  rest[0] = 0;

  for (int i = 1; i < n; i++) {
    buy[i] = max(buy[i-1], rest[i-1] - prices[
        i]);
    sell[i] = max(sell[i-1], buy[i-1] + prices
        [i]);
    rest[i] = max(rest[i-1], sell[i-1]);
  }
  return sell[n-1];
}

// House Robber: dp[i] = max money robbed from
    houses 0 to i
// Time: O(n), Space: O(n)
int rob(vector<int>& nums) {
  int n = nums.size();
  if (n == 0) return 0;
  if (n == 1) return nums[0];
```

```
  vector<int> dp(n);
  dp[0] = nums[0];
  dp[1] = max(nums[0], nums[1]);

  for (int i = 2; i < n; i++) {
    dp[i] = max(dp[i-1], dp[i-2] + nums[i]);
  }
  return dp[n-1];
}

/*
SPACE OPTIMIZATIONS:
*/

// 0/1 Knapsack with O(W) space: space-
    optimized version using 1D array
// Time: O(n*W), Space: O(W)
int knapsack01Optimized(vector<int>& weights,
    vector<int>& values, int W) {
  vector<int> dp(W + 1, 0);
  for (int i = 0; i < weights.size(); i++) {
    for (int w = W; w >= weights[i]; w--) {
      dp[w] = max(dp[w], dp[w - weights[i]] +
          values[i]);
    }
  }
  return dp[W];
}

/*
COMPLEXITY REFERENCE:
- 1D DP: O(n) states, usually O(n) or O(n^2)
    time
- 2D DP: O(n^2) states, usually O(n^2) or O(n^
    3) time
- Tree DP: O(n) states, O(n) time for each
    node
- Bitmask DP: O(2^n * n) states for TSP-like
    problems
- Digit DP: O(log n * sum * tight) states
*/

/*
ADVANCED DP OPTIMIZATIONS:
*/

/*
13. CONVEX HULL TRICK (CHT):
For DP transitions of form: dp[i] = min(dp[j]
    + cost(j, i)) where cost has convex
    property
Optimizes from O(n^2) to O(n log n) or O(n)
    with monotonic queries
*/

struct Line {
  long long m, b; // y = mx + b
  long long eval(long long x) { return m * x +
      b; }
  long double intersectX(Line l) { return (
      long double)(l.b - b) / (m - l.m); }
};

class ConvexHullTrick {
  vector<Line> lines;
  int ptr = 0;

  bool bad(Line l1, Line l2, Line l3) {
```

```
    return l1.intersectX(l3) <= l1.intersectX(
        l2);
  }

public:
  void addLine(long long m, long long b) {
    Line newLine = {m, b};
    while (lines.size() >= 2 && bad(lines[
        lines.size()-2], lines[lines.size()
        -1], newLine)) {
      lines.pop_back();
    }
    lines.push_back(newLine);
  }

  long long query(long long x) {
    if (lines.empty()) return LLONG_MAX;
    ptr = min(ptr, (int)lines.size() - 1);
    while (ptr < lines.size() - 1 && lines[ptr
        ].eval(x) >= lines[ptr + 1].eval(x)) {
      ptr++;
    }
    return lines[ptr].eval(x);
  }
};

// Example: Building optimization DP using CHT
    for quadratic cost transitions
// Time: O(n), Space: O(n)
long long buildingOptimization(vector<long
    long>& cost) {
  int n = cost.size();
  vector<long long> dp(n + 1, LLONG_MAX);
  vector<long long> prefix(n + 1, 0);

  for (int i = 1; i <= n; i++) {
    prefix[i] = prefix[i-1] + cost[i-1];
  }

  ConvexHullTrick cht;
  dp[0] = 0;
  cht.addLine(0, 0);

  for (int i = 1; i <= n; i++) {
    dp[i] = cht.query(prefix[i]);
    cht.addLine(-prefix[i], dp[i] + prefix[i]
        * prefix[i]);
  }
  return dp[n];
}

/*
14. KNUTH-YAO OPTIMIZATION:
For DP of form: dp[i][j] = min(dp[i][k] + dp[k
    +1][j] + cost[i][j]) where cost satisfies
    quadrangle inequality
Reduces complexity from O(n^3) to O(n^2)
*/

// Matrix Chain Multiplication with Knuth
    optimization: uses quadrangle inequality
// Time: O(n^2), Space: O(n^2)
int matrixChainKnuth(vector<int>& dims) {
  int n = dims.size() - 1;
  vector<vector<int>> dp(n, vector<int>(n, 0))
      ;
  vector<vector<int>> opt(n, vector<int>(n, 0)
      );
```

```
  // Initialize opt for length 1
  for (int i = 0; i < n; i++) opt[i][i] = i;

  for (int len = 2; len <= n; len++) {
    for (int i = 0; i <= n - len; i++) {
      int j = i + len - 1;
      dp[i][j] = INT_MAX;

      int start = (i == 0) ? 0 : opt[i][j-1];
      int end = (j == n-1) ? j-1 : opt[i+1][j
          ];

      for (int k = start; k <= end; k++) {
        int cost = dp[i][k] + dp[k+1][j] +
             dims[i] * dims[k+1] * dims[j+1];
        if (cost < dp[i][j]) {
          dp[i][j] = cost;
          opt[i][j] = k;
        }
      }
    }
  }
  return dp[0][n-1];
}

/*
15. DIVIDE AND CONQUER OPTIMIZATION:
For DP where optimal k is monotonic: if opt[i
    ][j] <= opt[i][j+1]
Reduces from O(kn^2) to O(kn log n)
*/

void divideConquerDP(int l, int r, int optL,
     int optR, vector<vector<long long>>& dp,
                    vector<vector<long long
                         >>& cost, int layer)
                         {
  if (l > r) return;

  int mid = (l + r) / 2;
  pair<long long, int> best = {LLONG_MAX, -1};

  for (int k = optL; k <= min(mid, optR); k++)
       {
    long long val = dp[layer-1][k] + cost[k][
        mid];
    best = min(best, {val, k});
  }

  dp[layer][mid] = best.first;
  int opt = best.second;

  divideConquerDP(l, mid - 1, optL, opt, dp,
       cost, layer);
  divideConquerDP(mid + 1, r, opt, optR, dp,
       cost, layer);
}

/*
16. SUM OVER SUBSETS (SOS) DP:
Calculate sum of f(S) for all subsets S of
    given set
Time: O(n * 2^n) instead of O(3^n) brute force
*/

// SUM OVER SUBSETS (SOS) DP: calculate f(S)
     for all subsets S efficiently
// Time: O(n * 2^n), Space: O(2^n)
vector<long long> sumOverSubsets(vector<long
     long>& arr) {
  int n = __builtin_ctz(arr.size()); // log2
       of size
  vector<long long> dp = arr;

  for (int i = 0; i < n; i++) {
    for (int mask = 0; mask < (1 << n); mask
         ++) {
      if (mask & (1 << i)) {
        dp[mask] += dp[mask ^ (1 << i)];
      }
    }
  }
  return dp;
}

// SOS DP variant: Count subsets with XOR = 0
     using frequency array
// Time: O(maxVal * maxVal), Space: O(maxVal)
int countXORZeroSubsets(vector<int>& nums) {
  int maxXOR = 0;
  for (int x : nums) maxXOR = max(maxXOR, x);

  vector<int> cnt(maxXOR + 1, 0);
  for (int x : nums) cnt[x]++;

  vector<long long> dp(maxXOR + 1, 0);
  dp[0] = 1;

  for (int i = 0; i <= maxXOR; i++) {
    if (cnt[i] > 0) {
      vector<long long> newDp = dp;
      for (int mask = 0; mask <= maxXOR; mask
           ++) {
        newDp[mask ^ i] += dp[mask] * cnt[i];
      }
      dp = newDp;
    }
  }
  return dp[0] - 1; // Subtract empty set
}

/*
17. SLOPE TRICK:
Maintain piecewise linear function for
     optimization problems
Useful for problems with "buy low, sell high"
     patterns
*/

class SlopeTrick {
  priority_queue<long long> L; // max heap for
       left slopes
  priority_queue<long long, vector<long long>,
       greater<long long>> R; // min heap for
      right
  long long minVal = 0;
  long long lazyL = 0, lazyR = 0;

public:
  void addRamp(long long a) { // add max(0, x
       - a)
    if (L.empty() || a <= L.top() + lazyL) {
      L.push(a - lazyL);
    } else if (R.empty() || a >= R.top() +
         lazyR) {
      R.push(a - lazyR);
    } else {
      minVal += a - (R.top() + lazyR);
      L.push(R.top() + lazyR - lazyL);
      R.pop();
      R.push(a - lazyR);
    }
  }

  void shiftRight(long long d) { lazyR += d; }
  void shiftLeft(long long d) { lazyL += d; }

  long long getMin() { return minVal; }
};

/*
18. ALIENS TRICK (Lagrange Multipliers):
Convert constrained optimization to
     unconstrained using binary search on
     penalty
*/

pair<long long, int> aliensTrick(vector<long
     long>& cost, int k, long long penalty) {
  int n = cost.size();
  vector<long long> dp(n + 1, LLONG_MIN);
  vector<int> cnt(n + 1, 0);

  dp[0] = 0;
  for (int i = 1; i <= n; i++) {
    for (int j = 0; j < i; j++) {
      long long val = dp[j] + cost[i-1] -
           penalty;
      if (val > dp[i]) {
        dp[i] = val;
        cnt[i] = cnt[j] + 1;
      }
    }
  }
  return {dp[n], cnt[n]};
}

/*
19. BROKEN PROFILE DP (Tiling DP):
DP on grid with bitmask representing filled
     cells
*/

// DP on grid with bitmask: dp[col][mask] =
     ways to tile up to column col
// Time: O(m * 2^n * n), Space: O(2^n)
int tilingDP(int n, int m) {
  vector<vector<int>> dp(m, vector<int>(1 << n
       , 0));
  dp[0][0] = 1;

  for (int col = 0; col < m; col++) {
    for (int mask = 0; mask < (1 << n); mask
         ++) {
      if (dp[col][mask] == 0) continue;

      function<void(int, int, int)> generate =
           [&](int pos, int cur, int next) {
        if (pos == n) {
          if (col + 1 < m) dp[col + 1][next]
               += dp[col][cur];
          return;
        }

        if (cur & (1 << pos)) {
          generate(pos + 1, cur, next);
        } else {
          generate(pos + 1, cur | (1 << pos),
               next | (1 << pos)); // vertical
               tile
          if (pos + 1 < n && !(cur & (1 << (
               pos + 1)))) {
            generate(pos + 2, cur | (1 << pos)
                 | (1 << (pos + 1)), next); //
                 horizontal tile
          }
        }
      };

      generate(0, mask, 0);
    }
  }
  return dp[m-1][0];
}

/*
COMPLEXITY REFERENCE FOR ADVANCED TECHNIQUES:
- Convex Hull Trick: O(n log n) or O(n) with
     monotonic queries
- Knuth-Yao: O(n^2) instead of O(n^3)
- Divide & Conquer: O(kn log n) instead of O(
     kn^2)
- SOS DP: O(n * 2^n) instead of O(3^n)
- Slope Trick: O(n log n) with priority queues
- Aliens Trick: O(n^2 log V) where V is value
     range
- Broken Profile: O(m * 2^n * n) for nxm grid
*/
```

10.6 Optimization tricks

`__builtin_ia32_ldmxcsr(40896);` disables denormals (which make floats 20x slower near their minimum value).

10.6.1 Bit hacks

- `x & -x` is the least bit in `x`.
- `for (int x = m; x; ) { --x &= m; ... }` loops over all subset masks of `m` (except `m` itself).
- `c = x&-x, r = x+c; (((r^x) >> 2)/c) | r` is the next number after `x` with the same number of bits set.
- `rep(b,0,K) rep(i,0,(1 << K)) if (i & 1 << b) D[i] += D[i^(1 << b)]` computes all sums of subsets.

10.6.2 Pragmas

- `#pragma GCC optimize ("Ofast")` will make GCC auto-vectorize loops and optimizes floating points better.
- `#pragma GCC target ("avx2")` can double performance of vectorized code, but causes crashes on old machines.
- `#pragma GCC optimize ("trapv")` kills the program on integer overflows (but is really slow).

1. `#pragma GCC target ("avx2")`
   `#pragma GCC optimize("O3")`
   `#pragma GCC optimize("unroll-loops")`

2. `#pragma GCC optimize("Ofast")`
   `#pragma GCC target("avx,avx2,fma")`

FastMod.h

Description: Compute $a\%b$ about 5 times faster than usual, where b is constant but not known at compile time. Returns a value congruent to $a \pmod b$ in the range $[0, 2b)$.

751a02, 8 lines

```
typedef unsigned long long ull;
struct FastMod {
  ull b, m;
  FastMod(ull b) : b(b), m(-1ULL / b) {}
  ull reduce(ull a) { // a % b + (0 or b)
    return a - (ull)((__uint128_t(m) * a) >>
        64) * b;
  }
};
```

FastInput.h

Description: Read an integer from stdin. Usage requires your program to pipe in input from file.
Usage: `./a.out < input.txt`
Time: About 5x as fast as cin/scanf.

7b3c70, 15 lines

```
inline char gc() { // like getchar()
  static char buf[1 << 16];
  static size_t bc, be;
  if (bc >= be) {
    buf[0] = 0, bc = 0;
    be = fread(buf, 1, sizeof(buf), stdin);
  } return buf[bc++]; // returns 0 on EOF
}
int readInt() {
  int a, c;
  while ((a = gc()) < 40);
  if (a == '-') return -readInt();
  while ((c = gc()) >= 48) a = a*10+c-480;
  return a - 48;
}
```

NimSprague.h

Description: Nim and Sprague-Grundy Theory for impartial combinatorial games. Essential theory, definitions, theorems, and algorithmic template for computing Grundy numbers.
Usage: `Compute Grundy numbers for game positions, XOR for disjoint sums`
Time: $\mathcal{O}(N+E)$ for DAG, $\mathcal{O}(N+E+mexcosts)$ in general

e82675, 57 lines

```
/*
% Nim and Sprague-Grundy Theory

Basic Definitions:
- Impartial game: Two-player turn-based, no
    chance, legal moves depend only on
    position
- Normal play: Player who cannot move loses
- Terminal position: No legal moves available
- mex(S) = minimum excluded value = min{n in N
    : n not in S}

Nim Game (Bouton's Theorem):
A Nim position (h1, h2, ..., hk) is losing for
    current player iff
h1 XOR h2 XOR ... XOR hk = 0 (where XOR is
    bitwise exclusive or)

Grundy Number Definition:
g(x) = mex{g(y) : y in Options(x)}
Terminal positions have g(x) = 0

Sprague-Grundy Theorem:
For disjoint sum of games G1 + G2 + ... + Gm:
g(G) = g(G1) XOR g(G2) XOR ... XOR g(Gm)
Sum is losing position iff XOR of component
    nimbers = 0

Algorithmic Template:
*/

int mex(const vector<int>& vals) {
  int n = vals.size();
  unordered_set<int> s(vals.begin(), vals.end
      ());
  int m = 0;
  while (s.count(m)) ++m;
  return m;
}

vector<int> grundy;
vector<vector<int>> moves;

int compute(int u) {
  if (grundy[u] != -1) return grundy[u];
  vector<int> nxt;
  for (int v : moves[u]) nxt.push_back(compute
      (v));
  return grundy[u] = mex(nxt);
}

/*
Usage:
1. Build moves graph
2. Initialize: grundy.assign(N, -1)
3. Compute: for each u, call compute(u)
4. For disjoint sum: XOR the grundy values of
    components

Common Patterns:
- Subtraction game: g(n) = mex{g(n-s) : s in S
    , s <= n}
- Many games have ultimately periodic Grundy
    sequences
- For small nimbers (<64), use bitmask for
    fast mex computation

Complexity: O(N + E) plus mex costs
*/
```

SublimeConfig.h

b6cb89, 6 lines

```
{
  "shell_cmd": "g++ -std=c++20 -DLOCAL -Wall -
      Wshadow \"$file\" -o \"$file_base_name\"
       && timeout 5s ./$file_base_name < input
      .txt > output.txt 2> error.txt",
  "file_regex": "^(..[^:]*):([0-9]+):?([0-9]+)
      ?:? (.*)$",
  "working_dir": "${file_path}",
  "selector": "source.c++"
}
```