Any possible infinite recursion?  
Invalidated pointers or iterators?  
Are you using too much memory?  
Debug with resubmits (e.g. remapped signals, see Various).

Time limit exceeded:  
Do you have any possible infinite loops?  
What is the complexity of your algorithm?  
Are you copying a lot of unnecessary data? (References)  
What is the complexity of your algorithm?  
Memory limit exceeded:  
Are you copying a lot of unnecessary data? (References)  
What is the complexity of your algorithm?  
Do you have any possible infinite loops?  
Invalidated iterators?

Any possible division by 0? (mod 0 for example)  
Any assertions that might fail?  
Are you reading or writing outside the range of any vector?  
Any uninitialized variables?  
Have you tested all corner cases locally?

Rewrite your solution from the start or let a team mate do it.  
Is your output format correct? (including whitespace)  
Go for a small walk, e.g. to the toilet.  
Ask the team mate to look at your code.  
Explain your algorithm to a team mate.  
Create some testcases to run your algorithm on.  
Add some assertions, maybe resubmit.  
Wrong answer:  
Print your solution! Print debug output, as well.  
Are you clearing all datastructures between test cases?  
Are you sure the STL functions you use work as you think?  
Are you sure your algorithm works?

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

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

For cyclic quadrilaterals the sum of opposite angles is $180^\circ$,  
e,f = ac + bd, and $A = \sqrt{(p-a)(p-b)(p-c)(p-d)}$.
2.4.3 Spherical coordinates

\[
x = r \sin \theta \cos \phi \\
y = r \sin \theta \sin \phi \\
z = r \cos \theta
\]

\[r = \sqrt{x^2 + y^2 + z^2}\]

\[\theta = \arccos(z/\sqrt{x^2 + y^2 + z^2})\]

\[\phi = \arctan(y, x)\]

2.5 Derivatives/Integrals

\[
\frac{dx}{d\arcsin x} = \frac{1}{\sqrt{1-x^2}}
\]

\[
\frac{dx}{d\arccos x} = -\frac{1}{\sqrt{1-x^2}}
\]

\[
\frac{dx}{d\tan x} = 1 + \tan^2 x
\]

\[
\frac{dx}{d\tan x} = \frac{1 + x^2}{x}
\]

\[
\int \tan ax = \frac{\ln|\cos ax|}{a}
\]

\[
\int \tan ax = \int ax \cos ax = \frac{\sin ax - ax \cos ax}{a^2}
\]

\[
\int e^{-x^2} = \frac{\sqrt{\pi}}{2} \text{erf}(x)
\]

\[
\int x^a dx = \frac{e^ax}{a^2} (a - 1)
\]

Integration by parts:

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

2.6 Sums

\[
c + c^{a+1} + \ldots + c^b = \frac{c^{b+1} - c^a}{c - 1}, c \neq 1
\]

\[
1 + 2 + 3 + \ldots + n = \frac{n(n + 1)}{2}
\]

\[
1^2 + 2^2 + 3^2 + \ldots + n^2 = \frac{n(2n + 1)(n + 1)}{6}
\]

\[
1^3 + 2^3 + 3^3 + \ldots + n^3 = \frac{n^2(n + 1)^2}{4}
\]

\[
1^4 + 2^4 + 3^4 + \ldots + n^4 = \frac{n(2n + 1)(3n^2 + 3n - 1)}{30}
\]

2.7 Series

\[
e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots, (-\infty < x < \infty)
\]

\[
\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots, (-1 < x \leq 1)
\]

\[
\sqrt{1 + x} = 1 + \frac{x}{2} + \frac{x^2}{3} + \frac{2x^3}{32} - \frac{5x^4}{128} + \ldots, (-1 \leq x \leq 1)
\]

\[
\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots, (-\infty < x < \infty)
\]

\[
\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots, (-\infty < x < \infty)
\]

2.8 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) \( \mu = \mathbb{E}(X) = \sum_x p_X(x) \) and variance

\[
\sigma^2 = \mathbb{V}(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = \sum_x (x - \mathbb{E}(X))^2 p_X(x)
\]

where \( \sigma \) 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 \( Y \),

\[
\mathbb{V}(aX + bY) = a^2\mathbb{V}(X) + b^2\mathbb{V}(Y).
\]

2.8.1 Discrete distributions

Binomial distribution

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

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

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

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

First success distribution

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

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

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

Poisson distribution

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

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

\[
\mu = \lambda, \sigma^2 = \lambda
\]

2.8.2 Continuous distributions

Uniform distribution

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

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

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

Exponential distribution

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

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

\[
\mu = \frac{1}{\lambda}, \sigma^2 = \frac{1}{\lambda^2}
\]
Normal distribution

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

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

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

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

2.9 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 such a sequence of random variables generated by the Markov process. Then there is a transition matrix $P = (p_{ij})$, with $p_{ij} = Pr(X_n = i | X_{n-1} = j)$, and $p^n = P^np(0)$ is the probability distribution for $X_n$ (i.e., $p^n(0) = Pr(X_n = i)$), where $p(0)$ is the initial distribution.

$\pi$ is a stationary distribution if $\pi = \pi P$. 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 $i$. $\pi_j / \pi_i$ 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, $\lim_{k \to \infty} P^k = \pi$.

A Markov chain is ergodic if it is irreducible andaperiodic (i.e., the god of cycle lengths is 1). $\lim_{k \to \infty} P^k = \pi$.

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

Data structures (3)
Matrix LineContainer Treap FenwickTree FenwickTree2d

SubMatrix.h

Description: Computes submatrix sums quickly, given upper-left and lower-right corners (half-open).

Usage: SubMatrix<ToInt> m(matrix);

m.sum(0, 0, 2, 2); // top left 4 elements

Usage:

Calculate submatrix sums quickly, given upper-left and lower-right corners (half-open).

Matrix.h

Description: Basic operations on square matrices.

Usage: Matrix<int, 3> A;

A.d = {{{1,2,3}}, {{4,5,6}}, {{7,8,9}}};

vector<int> vec = (A)* vec;

vec = (A)^ 2, 3 vec;

typedef array<array<T, N>, N> d{};

typedef LineContainer<T, int>

template <class T> struct SubMatrix {
  vector<T> vec;
}

struct SubMatrix(vector<vector<T>>& v) {
  vector<vector<T>> p;
  p.assign(R+1, vector<T>(C+1));
  int res = 0;
  for (int r = 0; r < R; ++r)
    for (int c = 0; c < C; ++c)
      p[r+1][c+1] = v[r][c] + p[r][c+1] + p[r+1][c] - p[r][c];
  return res;
}

Matrix::Matrix() {
  d = array<array<T, N>, N>{};
  x = {}; y = {}; c = 1;
}

struct Node {
  int x, y, c = 1;
  Node *l = 0, *r = 0;
}

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: $O(log N)$. (Use persistent segment trees for $log (log N)$). (Call fakeUpdate() before initializing.)

FenwickTree.h

Description: Computes partial sums a[0] + a[1] + ... + a[pos], and updates single elements a[j], taking the difference between the old and new value.

Time: Both operations are $O(log N)$.

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 initializing).

Time: $O(log N^2)$. (Use persistent segment trees for $log (log N)$.)
```
for (j; j < x - k - 1) { 
    sum += ft[x-1].query(index-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. Set inf to something reasonable before use. 
Usage: RMQ rmq(values) 
rmq.query(inclusive, exclusive); 
Time: O(|V|*log|V| + Q) 
const int inf = numeric_limits<int>::max(); 

template <class T> 
struct RMQ { 
vector<vector<T>> jmp; 
RMQ(const vector<T>& V) { 
    int N = sz(V); on = 1, depth = 1; 
    while (on < sz(V)) on *= 2, depth++; 
    jmp.assign(depth, V); 
    rep(i,0,depth-1) jmp[i][1] = min(jmp[i][i]), jmp[i][0] = min(N - 1, 3 + (1 << i)); } 

T query(int a, int b) { 
    if (b <= a) return inf; 
    int dep = 31 - __builtin_clz(b - a); 
    return min(jmp[dep][a], jmp[dep][b - (1 << dep)]); }

}

Numerical (4)

Polynomial.h
struct Polynomial 
int n; vector<double> a; 
Polynomial(int n; a[n+1]) { 
    double val = 0; 
    for(int i = 0; i <= n; i++) { 
        val *= x; 
    } 
    return val; 
} 

void diff() { 
    rep(i,1,n+1) a[i-1] = i*a[i]; 
    a.pop_back(); 
} 

void divroot(double x0) { 
    double b = a.back(); 
    c = a.back(); 
    for (int i = n-1; i >= 0; --i) { 
        a[i+1] = x0*c; 
    } 
    a.pop_back(); 
} 

BinarySearch.h
Description: Finds a zero point of f on the interval [a,b]. f(a) must be less than 0 and f(b) greater than 0. Useful for solving equations like kx=sin(x) as the example below. 
Usage: double function(double x) { return 0.23*x-sin(x); } 
double x0 = bs(0,1000,d); 
Time: O(log(k-a)) 
}

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

PolyRoots.h
Description: Finds the real roots to a polynomial. 
Usage: 
PolyRoots() 
PolyRoots(const int n; a[n+1]) { 
    double val = 0; 
    for(int i = 0; i <= n; i++) { 
        val *= x; 
    } 
    return val; 
} 

void roots(vector<double>& roots) { 
    double val = 0; 
    for(int i = 0; i <= n; i++) { 
        val *= x; 
    } 
    return val; 
} 

PolyInterpolate.h
Description: Given n points (x[i], y[i]), computes an n-1-degree polynomial that passes through them. p(x) = a[0] + a[1]*x + [a[2] + a[3]*x]*x + ... + [a[n-1] + a[n]*x]*x^(n-1). For numerical precision, pick x[k] = e*cos(K*(n-1) + k), k = 0...n-1. 
Time: O(n^2) 
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-1] - y[k] / (x[i] - x[k]); 
        double last = 0; 
        temp[i] = rep(0,n,0) rep(i,0,n) { 
            res[i] += y[k] * temp[i]; 
            temp[i] = last * x[k]; 
        } 
        return res; 
    } 
}

HillClimbing.h
Description: Poor man's optimization for unimodal functions. 
typedef array<double, 2*P; 
func(P p); 
pair<double, P> hillClimb(P start) { 
    pair<double, P> cur(func(start), start); 
    for (double jmp = 100; jmp >= 10; jmp /= 2) { 
        rep(j,0,100) rep(dx,-1,2) rep(dy,-1,2) { 
            P p = cur.second; 
            cur = min(cur, make_pair(func(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^5, although in practice you will want to verify that the result is stable to desired precision when epsilon changes. 

type double quad(double +(f)(double), double a, double b) { 
    double int n = 10000; 
    const double h = (b - a) / n; 
    double v = f(a) + f(b); 
    rep(i,1,n+1) v += f(a + h*i); 
    return v * h / 3; 
}

IntegrateAdaptive.h
Description: Fast integration using an adaptive Simpson's rule. 
Usage: 
quad(double x, double y) { 
    double x0 = x + y*x + z*x <= 1; 
    double g(double y) { 
        if (y <= x) return x; 
        return quad(y, y1); 
    } 
    double f(double z) { 
        if (z <= y) return quad(z, z1); 
        double sphereVol = quad(f, -1, 1); 
        pi = sphereVol/3/4; 
    } 
```
\textbf{Determinate.h}

\textbf{Description:} Calculates determinant of a matrix. The destroys.

\textbf{Time:} $O(N^3)$

\textbf{double det(vector<vector<double>> A)} \{\n  int n = sz(A); double res = 1; \n  rep(i,0,n) \{ \n    int b = 1; \n    rep(j,i+1,n) if (fabs(A[j][i])) > fabs(A[b][i]) \{ b = j; \n  \} \n  if (b != i) swap(A[b], A[i]); res *= -1; \n  \} \n  return res; \n\}

\textbf{IntDeterminate.h}

\textbf{Description:} Calculates determinant using modular arithmetics. Modulos can also be removed to get a pure-integer version.

\textbf{Time:} $O(N^3)$

\textbf{const ll mod = 12345; \nll det(vector<vector<ll>>& a)} \{ \n  int n = sz(a); \n  rep(i,0,n) \{ \n    rep(j,0,n) a[i][j] = 0; \n  \} \n  rep(i,0,n) \{ \n    if (i == b) \{ swap(A[i], A[b]); res *= -1; \n    \} \n    int f = i; \n    rep(j,0,n) \{ \n      if (a[f][j] % mod == 0) continue; \n      if (abs(a[f][j]) > abs(a[i][j])) { i = f; \n    } \n  \} \n  return res; \n\}

\textbf{Simplex.h}

\textbf{Description:} Solves a general linear maximization problem: maximize $c^T x$ subject to $A x \leq b, x \geq 0$. 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.

\textbf{Usage:} \texttt{vd v = \{}{1,1,-1}, {-1,1,1}, {1,-1,-2}\}; \nT val = LSolver(A, b, c, b.solve()); \n\}

\textbf{Time:} $O(2^m)$ in the general case.
**MatrixInverse.h**

Description: Invert matrix $A$. Returns result, stored in $A$ unless singular (rank $< n$). Can easily be extended to prime modulus; for prime powers, repeatedly set $A^{-1} = A^{-1}(2I-A^{-1})$ modulo $p$ where $A^{-1}$ starts as the inverse of $A$ mod $p$.

Time: $O(N^3)$

```c
int matinv(vector<vector<double>>& A) {
    int n = sz(A);
    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;
        if (fabs(A[r][c]) < 1e-12) return -1;
        swap(A[r],A[c]);
        for(j=0;j<n;j++) swap(col[j], col[c]);

        rep(j,i+1,n) {
            double f = A[j][i] / v;
            rep(k,0,n) A[j][k] -= f*tmp[i][k];
        }
        if (f == 1) rep(j,i+1,n) A[j][i] = diag[j];
    }
    return 0;
}
```

**MatrixInverse Tridiagonal FFT ModularArithmetic ModInverse ModPow**

**Tridiagonal.h**

Description: Solves a linear equation system with a tridiagonal matrix $A$ with diagonal $d$, subdiagonal $s$, and superdiagonal $s$ super, i.e., $x = \text{tridiagonal}(d,s,s)$ solves the equation system:

\[
\begin{bmatrix}
0 & 1 & 0 & \cdots & 0 \\
1 & 0 & 1 & \cdots & 0 \\
0 & 1 & 0 & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & 0 \\
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
\vdots \\
x_n-1 \\
\end{bmatrix}
= \begin{bmatrix}
d_0 \\
d_1 & d_2 & 0 & \cdots & 0 \\
0 & d_2 & d_3 & \cdots & 0 \\
0 & 0 & d_3 & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & d_{n-1} \\
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
\vdots \\
x_{n-1} \\
\end{bmatrix}
\]

The size of $d$, $s$ and $s$ super should be the same and super and sub should be one element shorter. $T$ is intended to be double.

```c
template <class T> T tridiagonal(vector<T> d, vector<T> s, vector<T> s2, vector<T> b) {
    int n = sz(d);
    for (int i = 0; i < n; i++) b[i] /= d[i];

    for (int i = 0; i < n; i++) {
        double v = 1;
        rep(j,i+1,n) v *= d[j];
        rep(k,0,i) b[j] -= s[k]*v*tmp[k][i];
    }
    b[i] /= d[i];
    return b;
}
```

**FFT.h**

Description: Fast Fourier transform. Also includes a function for convolution:

\[
\text{conv}(a,b) = (a \star b, \mod m)
\]

```c
void fft(vector<
```
5.3 Primality

eratosthenes.h

Description: Prime sieve for generating all primes up to a certain limit. isprime(i) is true if i is a prime.
Time: \(\text{lim} = 100'000'000 \approx 0.8 \text{s.} \) Runs 30% faster if only odd indices are stored.

```cpp
const int MAX_PR = 5000000;
bitset<MAX_PR> isprime;
vi eratosthenes_sieve(int lim) { isprime.set(); isprime[0] = isprime[1] = 0;
for (int i = 4; i < lim; i += 2) isprime[i] = 0;
for (int j = 3; j < lim; j += 2) if (isprime[i])
    for (int k = j + j < lim; k = j + 2) isprime[j] = 0;
vi pr;
rep(i,2,lim) if (isprime[i]) pr.push_back(i);
return pr;
}
```

MillerRabin.h

Description: Miller-Rabin primality probabilistic test. Probability of failing one iteration is at most 1/4. 15 iterations should be enough for 50-bit numbers.
Time: 15 times the complexity of \(a^b \mod c\)  

```cpp
bool prime(ull p) { if (p == 2) return true;
if ((p-1) % 2 == 0) return false;
ull a = p - 1;
while (a % 2 == 0) a /= 2;
rep(i,0,a-1) if (mod_pow(a, a, a) == 1) break;
 ull m = a; a = m; m = modpow(a, (a - 1) / 2, p);
while (modpow(a, m, p) == 1) m *= 2;
return m == p - 1;
}
```

factor.h

Description: Pollard's rho algorithm. It is a probabilistic factorisation algorithm, whose expected time complexity is good. Before you start using it, run init(bites), where bites is the length of the numbers you use. Returns factors of the input without duplicates.
Time: Expected running time should be good enough for 50-bit numbers.

```cpp
vi vector_ui(vi pr) { ull t(ull a, ull n, ull nhas) { return (mod_mul(a, a, n) + nhas) % n;
}
vi vector_ui_factor(ull d) { return vector_ui(res); }
for (int i = 0; i < sz(pr) && pr[0] <= d; i++) if (mod_pow(d, pr[i], p) == 0)
    while (d % pr[i] == 0) d /= pr[i];
return d;
}
```

ModSum.h

Description: Sums of moded arithmetic progressions.

```cpp
while (modpow(n, (p-1) / 2, p) != 1 - p1); n1 = modpow(n, a, s, p);
(\sum_{i=0}^{n-1} (i + j) + [a[1] + b[1]]) \mod mod;
return c;
```
5.4 Divisibility

euclid

Description: Finds the Greatest Common Divisor to the integers a and b. Euclid also finds two integers x and y, such that ax + by = gcd(a, b).

```
ll gcd(ll a, ll b) { return _gcd(a, b); }
ll euclid(ll a, ll b, ll x, ll y) {
    if (b) { ll d = euclid(b, a % b, y, x);
        y -= a / b * x, d; }
    return x, y = a / b, x; }
```

5.5 Chinese remainder theorem

cinese

Description: Chinese Remainder Theorem.

cinese(a, m, b, n) returns a number x, such that x ≡ a (mod m) and x ≡ b (mod n). For not coprime m, n, use chinese common. Note that all numbers must be less than Z^n if you have Z = unsigned long long.

```
const int LIM = 5000000;
int phi[LIM];
void calculatePhi() {
    for(i = 3; i < LIM; i += 2)
        int phi[i] = i & 1 ? i : i / 2;
    for(i = 3; i < LIM; i++)
        for(j = i; j < LIM; j += i)
            phi[j] /= i;
}
```

5.6 Pythagorean Triples

The Pythagorean triples are uniquely generated by

\[ a = k \cdot (m^2 - n^2), \quad b = k \cdot (2mn), \quad c = k \cdot (m^2 + n^2), \]

with m > n > 0, k > 0, m,n,l, and either m or n even.

5.7 Primes

p = 962592769 is such that 2^p | p - 1, which may be useful. For hashing use 970592641 (31-bit number), 3144359979727 (45-bit), 3006703054067924923 (54-bit). There are 78498 primes less than 1,000,000.

5.8 Estimtes

\[ \sum_{d|n} d = O(n \log \log n) \]

The number of divisors of n is at most around 100 for n = 5e4, 500 for n = 1e7, 2000 for n = 1e10, 200,000 for n = 1e19.

6.1 The Twelvefold Way

Counts the # of functions f : N → K, |N| = n, |K| = k. The elements in N and K can be distinguishable or indistinguishable, while f can be injective (one-to-one) or surjective (onto).
Let the number of \( n \)-permutations whose cycle lengths all belong to the set \( S \) be denoted by \( g_S(n) \). Then
\[
\sum_{n=0}^{\infty} g_S(n) \frac{x^n}{n!} = \exp \left( \sum_{n \in S} \frac{x^n}{n!} \right)
\]

### 6.2.2 Cycles

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

### 6.2.3 Derangements

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

### Derangements.h

**description:** Generates the \( i \)th derangement of \( S_n \) (in lexicographical order).

```c
#include <algorithm>
#include <vector>

template <class T, int N>
struct derangements {
  T dgen[N][N], choose[N][N], fac[N];
  derangements() {
    fac[0] = choose[0][0] = 1;
    memset(dgen, 0, sizeof(dgen));
    rep(m,1,N) {
      fac[m] = fac[m-1] * m;
      choose[m][0] = choose[m][m] = 1;
      rep(k,1,m) {
        choose[m][k] = choose[m-1][k-1] + choose[m-1][k];
      }
    }
  }
  T DGen(int n, int k) {
    T ans = 0;
    if (dgen[n][k]) return dgen[n][k];
    rep(i,0,k) {
      ans += choose[k][i] * fac[i-1] * DGen(n-1,k-i);
    }
    return dgen[n][k] = ans;
  }
  void generate(int n, T idx, int *res) {
    int vals[N];
    rep(i,0,n) vals[i] = i;
    rep(i,0,n) {
      int j, k = 0, m = n - 1;
      if (vals[j] > i) ++j;
      rep(j,0,m)
        T p = 0;
  }
}
```

### 6.2.4 Involutions

An involution is a permutation with maximum cycle length 2, and it is its own inverse.
\[
a(n) = a(n-1) + (n-1)a(n-2)
\]
\[
a(0) = a(1) = 1
\]

### 6.2.5 Stirling numbers of the first kind

\( s(n,k) \) is the unsigned Stirling numbers of the first kind, and they count the number of permutations on \( n \) items with \( k \) cycles.
\[
s(n,k) = s(n-1,k-1) - (n-1)s(n-1,k)
\]

### 6.2.6 Eulerian numbers

Number of permutations \( \pi \in S_n \) in which exactly \( k \) elements are greater than the previous element. \( k \) is s.t.
\[
\pi(j) > \pi(j+1), k+1 \text{ is s.t. } \pi(j) \geq j, k \text{ is s.t. } \pi(j) > j.
\]
\[
E(n,k) = (n-k)E(n-1,k-1) + (k+1)E(n-1,k)
\]
\[
E(n,0) = E(n,n-1) = 1
\]
\[
E(n,k) = \sum_{j=0}^{k} (-1)^j \binom{n+1}{j} (k+1-j)^n
\]

### 6.2.7 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 \( G \).
\[
g(n) = \frac{1}{n} \sum_{k=0}^{n-1} f(g(n,k)) = \frac{1}{n} \sum_{k|n} f(k)\phi(n/k).
\]

### 6.3 Partitions and subsets

#### 6.3.1 Partition function

Partitions of \( n \) with exactly \( k \) parts, \( p(n,k) \), i.e., writing \( n \) as a sum of \( k \) positive integers, disregarding the order of the summands.
\[
p(n,k) = p(n-1,k-1) + p(n-k,k)
\]
\[
p(0,0) = p(1,n) = p(n,n) = p(n,n-1) = 1
\]

For partitions with any number of parts, \( p(n) \) obeys
\[
p(0) = 1, \ p(n) = \sum_{k \in \mathbb{N}\setminus\{0\}} (-1)^{k+1}p(n-k(3k-1)/2)
\]
\[
p(n) \sim 0.145/n \cdot \exp(2.56\sqrt{n})
\]
\[
\frac{p(n)}{n} \sim \frac{1}{11} \ 1.234567892050100
\]
### 6.3.2 Binomials

**binomial.h**

**Description:** The number of \(k\)-element subsets of an \(n\)-element set, \(\binom{n}{k}\), is \(\mathcal{O}(\min(k, n-k))\) for \(\log n\) lines.

```c
ll chooseModP(ll n, ll m, int p) {
    ll c = 1, to = min(k, n-k);
    if (to < 0) return 0;
    rep(i,0,to) c = c * (n-i) / (1+i);
    return c;
}
```

**binomialModPrime.h**

**Description:** Lucas’ theorem. Let \(n, m\) be non-negative integers and \(p\) a prime. Write \(n = n_p p^k + \ldots + n_0 p + n_0\) and \(m = m_p p^k + \ldots + m_0 p + m_0\). Then \(\binom{n}{m} \equiv \prod \binom{n_k}{m_k} \pmod{p}\). fact and invfact must hold pre-computed factorials / inverse factorials, e.g. from ModInverse.h.

**Time:** \(\mathcal{O}(\log n)\) lines

```c
ll chooseModP(ll n, ll m, int p, vi6 fact, vi6 invfact) {
    ll c = 1;
    while (n || m) {
        ll a = n % p, b = m % p;
        if (a < b) return 0;
        c = c * fact[a] % p * invfact[b] % p * invfact[a-b] % p;
        n /= p; m /= p;
    }
    return c;
}
```

**RollingBinomial.h**

**Description:** \(\binom{n}{k}\) (mod m) in time proportional to the difference between \((n, k)\) and the previous \((n, k)\). 14 lines

```c
const ll mod = 1000000007;
vector<ll> invv; // precomputed up to max n, inclusively
struct Bin {
    int N = 0, K = 0; ll r = 1;
    void (ll a, ll b) { r = r * a % mod * invv[b] % mod; }
    ll choose(int n, int k) {
        if (k > n || k < 0) return 0;
        while (N < n) N += m(N, N-K);
        while (K < k) K += m(K, K+1, K);
        while (K > k) m(K, N-K+1), K;
        while (N > n) m(N-K, N), N = -N;
        return r;
    }
};
```

### 6.3.3 Stirling numbers of the second kind

**Partitions of \(n\) objects into \(k\) groups**

\[ S(n, k) = S(n-1, k-1) + kS(n-1, k) \]

\[ S(n, 1) = S(n, n) = 1 \]

\[ S(n, k) = \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^n \]

### 6.3.4 Bell numbers

**Total number of partitions of \(n\) distinct elements.**

\[ B(n) = \sum_{k=1}^{n} \binom{n-1}{k-1} B(n-k) = \sum_{k=1}^{n} S(n, k) \]

\[ B(0) = B(1) = 1 \]

The first are 1, 1, 2, 5, 15, 52, 203, 742900, 678570, 4213597. For a prime \(p\)

\[ B(p^n + m) \equiv mB(n) + B(n+1) \mod p \]

### 6.3.5 Triangles

**Given rods of length 1, \ldots, \(n\),**

\[ T(n) = \frac{1}{24} \sum_{k=1}^{n} \binom{n-1}{k-1} (n-1)(n-2)(n-3)(n-4) \]

\[ T(n) = \frac{1}{24} \binom{n-1}{k-1} (n-1)(n-2)(n-3)(n-4) \]

\[ T(n) = \frac{1}{24} \binom{n-1}{k-1} (n-1)(n-2)(n-3)(n-4) \]

is the number of distinct triangles (positive area) that can be constructed, i.e., the \# of 3-subsets of \([n]\) s.t. \(x \leq y \leq z\) and \(z \neq x + y\).

### 6.4 General purpose numbers

#### 6.4.1 Catalan numbers

\[ C_n = \frac{1}{n+1} \binom{2n}{n} = \frac{2n}{n+1} \frac{n}{n+1} \frac{(n-1)!}{2!} \]

\[ C_{n+1} = \frac{2(2n+1) C_n}{n+2} \]

\[ C_0 = 1, C_{n+1} = \sum_{i=0}^{C_n} C_i C_{n-i} \]

First few are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900.

- \# of monotonic lattice paths of a \(n \times n\)-grid which do not pass above the diagonal.
- \# of expressions containing \(n\) pairs of parenthesis which are correctly matched.
- \# of full binary trees with \(n+1\) leaves (0 or 2 children).
- \# of non-isomorphic ordered trees with \(n+1\) vertices.
- \# of ways a convex polygon with \(n+2\) sides can be cut into triangles by connecting vertices with straight lines.
- \# of permutations of \([n]\) with no three-term increasing subsequence.

#### 6.4.2 Super Catalan numbers

The number of monotonic lattice paths of a \(n \times n\)-grid that do not touch the diagonal.

\[ S(n) = \frac{3(2n-3) S(n-1) - (n-3) S(n-2)}{n} \]

\[ S(1) = S(2) = 1 \]

1, 1, 3, 11, 45, 197, 903, 4279, 20793, 103049, 518859

#### 6.4.3 Motzkin numbers

Number of ways of drawing any number of nonintersecting chords among \(n\) points on a circle. Number of lattice paths from \((0,0)\) to \((n,0)\) never going below the \(x\)-axis, using only steps \(NE, SE, E\).

\[ M(n) = \frac{3(n-1) M(n-2) + (2n+1) M(n-1)}{n+2} \]

\[ M(0) = M(1) = 1 \]

1, 1, 2, 4, 9, 21, 51, 127, 323, 835, 2188, 5798, 15511, 41835, 113634

#### 6.4.4 Narayana numbers

Number of lattice paths from \((0,0)\) to \((2n,0)\) never going below the \(x\)-axis, using only steps \(NE, SE, N, W\) and \(SE, E, W\) peak.

\[ N(n, k) = \frac{1}{n} \binom{n}{k} \binom{k}{1} \binom{n}{k-1} \]

\[ N(1, n) = N(n, n) = 1 \]
\[
\sum_{k=1}^{n} N(n, k) = C_n
\]

1, 1, 1, 1, 3, 1, 6, 1, 10, 1, 1, 15, 50

6.4.5 Schröder numbers

Number of lattice paths from (0, 0) to (n, n) using only steps N, NE, E, never going above the diagonal. Number of lattice paths from (0, 0) to (2n, 0) using only steps NE, SE, and double east EE, never going below the x-axis. Twice the Super Catalan number, except for the first term.

7 Graph (7)

7.1 Fundamentals

bellmanFord

Description: Calculates shortest path in a graph that might have negative edge distances. Propagates negative infinity distances (sets dist = -inf), and returns true if there is some negative cycle. Unreachable nodes get dist = inf.

Time: \(O(VE)\)

typedef int T; // or whatever
struct Edge { int src, dest; T weight; };
struct Node { T dist; }
struct Graph { vector<Node> nodes; vector<Edge> edges; };

int init = numeric_limits<T>::max();

Graph g { vector<Node> nodes; vector<Edge> edges; };

struct

BellmanFord(

vector<vector<ll>>& m)
{ void floydWarshall(vector<vector<ll>>& m) {
  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][j] < 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. Output is an ordering of vertices (array idx), such that there are edges only from left to right. The function returns false if there is a cycle in the graph.

Time: \(O(V + E)\)

7.2 Euler walk

EulerWalk.h

Description: Eulerian undirected/directed path/cycle algorithm. Returns a list of nodes in the Eulerian path in cycle with src at both start and end, or empty list if no cycle/path exists. To get edge indices back, also put it->second in s (and then return).

Time: \(O(E)\) where E is the number of edges.

struct V { int nins; // (dest, edge index)
  int nins = 0;
};

vi eulerWalk(vector<V> nodes, int nodes, int src=0) {
  int c = 0;
  trav(n, nodes) c += abs(n.nins - sz(n.outs));
  if (c > 2) return {};

  vector<vector<pair<int, int>>> its;
  for (auto& it = its[0]; it != end; ++it)
    if (it->n && H[it->n] == H[it->ndst] + 1) {
      HI = H[it->n];
      return;
    }

  vi ret = 0;
  trav(n, nodes) t = it->ndst;
  if (it->nins == t) ret = 1;

  return ret;
}

FloydWarshall.h

Description: Calculates all-pairs shortest path in a directed graph that might have negative edge distances. 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, int if no path, or -inf if the path goes through a negative-weight cycle.

Time: \(O(N^3)\)

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

  return;
}

7.3 Network flow

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.

Type: \(O(V^2 \sqrt{E})\)

template void add_flow(Edge e, Flow f) { if (e.cap < f) return;
  if (e.cap == f) e.cap -= f; e.f += f; e.rcap += f;
}

Flow maxflow(int s, int t) { Flow maxFlow = 0;
  if (s == t) return;

  PushRelabel(g, s, t, Flow cap, Flow rcap=0) {
    if (ec[u] > 0) {
      u = hs[ec[dest]]; push_back(dest); e.f += f; e.c -= f;
      back.f -= f; back.c += f; ec[back.dest] -= f;
    }
  }

  while (hs[hi].empty()) {
    if (hi == 0) { hi = H[0];
      for (int i = 0; i < sz(hs[hi].data()); ++i)
        if (hs[hi].empty()) return;

      int vi = sz(g)[0];
      while (hs[hi].empty()) if (hi == 0) return;

      if (ec[u] > 0) // discharge u
        if (H[i] < H[i]) H[i] = H[i];
        if (H[i] != H[i-1]) hi = i;
  }

  return maxFlow;
}
#include <bits/extc++.h>

typedef const Description: MinCostMaxFlow.h
Time: Approximately actual flow, look at positive values only.
that negative cost cycles are not allowed (that's NP-hard). To obtain the
edges are not. If costs can be negative, call setpi before maxflow, but note
pair<ll, ll> maxflow(

\[ \begin{align*}
\text{for } (i) & \text{ if } (r) \text{ flow}[p][x] \rightarrow r; \\
\text{else } & \text{ flow}[x][p] \rightarrow r; \\
& \text{ repr}(i,0,N) \text{ repr}(j,0,N) \text{ totcost} = \text{cost}([i][j]) \times \text{flow}([i][j]); \\
& \text{return} (\text{totflow}, \text{totcost}); \\
\end{align*} \]

// If some costs can be negative, call this before maxflow:
void setpi(int s) { // (otherwise, leave this out)
fill(all(pi), INF); pi[s] = 0; 
int it = N, ch = 1; ll v;
while (ch--) {
iter = iter - 1; 
if ((graph[p][y] -= inc) <= 0) graph[p].erase(y); 
p = par[y];
}

EdmondsKarp.h
Description: Flow algorithm with guaranteed complexity \(O(VE^2)\). To get
edge flow values, compare capacities before and after, and take the positive
values only.

pair<int, int> GetMinCut(vector<int>& weights) {
int N = sz(weights); 
vi used(N), cut, best_cut;
int best_weight = -1;
for (int phase = N-1; phase >= 0; phase--) {
vi w = weights[0], added = used; 
int prev, k = 0;
rep(i,0,phase) {
prev = k; 
k = -1;
rep(j,1,N) {
if ((added[j] && (k == -1 || w[j] > w[k])) k = j; 
if (j == phase) {
rep(j,0,N) weights[prev][j] = weights[k][j]; 
rep(j,0,N) weights[j][prev] = weights[j][prev]; 
used[k] = true;
put_back(k);
}
if (best_weight == -1 || w[k] < best_weight) {
best_cut = cut; 
best_weight = w[k];
}
}
else {
rep(j,0,N) w[j] = weights[k][j]; 
added[k] = true;
}
}
return (best_weight, best_cut); }

7.4 Matching
hopcroftKarp.h
Description: Find a maximum matching in a bipartite graph.
Usage: vi ba,n, -1); hopcroftKarp(g, ba); 
Time: \(O\left(VE^{\sqrt{3}}\right)\)

bool dfs(int a, int layer, const vector<int>& weights) {
int N = sz(weights); 
vi used(N), cut, best_cut;
int best_weight = -1;
for (int phase = N-1; phase >= 0; phase--) {
vi w = weights[0], added = used; 
int prev, k = 0;
rep(i,0,phase) {
prev = k; 
k = -1;
rep(j,1,N) {
if ((added[j] && (k == -1 || w[j] > w[k])) k = j; 
if (j == phase) {
rep(j,0,N) weights[prev][j] = weights[k][j]; 
rep(j,0,N) weights[j][prev] = weights[j][prev]; 
used[k] = true;
put_back(k);
}
if (best_weight == -1 || w[k] < best_weight) {
best_cut = cut; 
best_weight = w[k];
}
}
else {
rep(j,0,N) w[j] = weights[k][j]; 
added[k] = true;
}
}
return (best_weight, best_cut); }

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.

if (r) flow[p][x] \rightarrow r; 
else \text{ flow}[x][p] \rightarrow r; 
\text{ repr}(i,0,N) \text{ repr}(j,0,N) \text{ totcost} = \text{cost}([i][j]) \times \text{flow}([i][j]); 
\text{return} (\text{totflow}, \text{totcost}); 

// If some costs can be negative, call this before maxflow:
void setpi(int s) { // (otherwise, leave this out)
fill(all(pi), INF); pi[s] = 0; 
int it = N, ch = 1; ll v;
while (ch--) {
iter = iter - 1; 
if ((graph[p][y] -= inc) <= 0) graph[p].erase(y); 
p = par[y];
}

EdmondsKarp.h
Description: Flow algorithm with guaranteed complexity \(O(VE^2)\). To get
edge flow values, compare capacities before and after, and take the positive
values only.

pair<int, int> GetMinCut(vector<int>& weights) {
int N = sz(weights); 
vi used(N), cut, best_cut;
int best_weight = -1;
for (int phase = N-1; phase >= 0; phase--) {
vi w = weights[0], added = used; 
int prev, k = 0;
rep(i,0,phase) {
prev = k; 
k = -1;
rep(j,1,N) {
if ((added[j] && (k == -1 || w[j] > w[k])) k = j; 
if (j == phase) {
rep(j,0,N) weights[prev][j] = weights[k][j]; 
rep(j,0,N) weights[j][prev] = weights[j][prev]; 
used[k] = true;
put_back(k);
}
if (best_weight == -1 || w[k] < best_weight) {
best_cut = cut; 
best_weight = w[k];
}
}
else {
rep(j,0,N) w[j] = weights[k][j]; 
added[k] = true;
}
}
return (best_weight, best_cut); }

7.4 Matching
hopcroftKarp.h
Description: Find a maximum matching in a bipartite graph.
Usage: vi ba,n, -1); hopcroftKarp(g, ba); 
Time: \(O\left(VE^{\sqrt{3}}\right)\)

bool dfs(int a, int layer, const vector<int>& weights) {
int N = sz(weights); 
vi used(N), cut, best_cut;
int best_weight = -1;
for (int phase = N-1; phase >= 0; phase--) {
vi w = weights[0], added = used; 
int prev, k = 0;
rep(i,0,phase) {
prev = k; 
k = -1;
rep(j,1,N) {
if ((added[j] && (k == -1 || w[j] > w[k])) k = j; 
if (j == phase) {
rep(j,0,N) weights[prev][j] = weights[k][j]; 
rep(j,0,N) weights[j][prev] = weights[j][prev]; 
used[k] = true;
put_back(k);
}
if (best_weight == -1 || w[k] < best_weight) {
best_cut = cut; 
best_weight = w[k];
}
}
else {
rep(j,0,N) w[j] = weights[k][j]; 
added[k] = true;
}
}
return (best_weight, best_cut); }

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.
**Min cost bipartite matching. Negate costs for max cost.**

**weightedMatching.h**

```cpp
O \(N^2\)
rep(i,0,n) if (!found[i]) cover.push_back(i); repro(0,n,0) if (seen[i]) cover.push_back(n+i); assert(sz(cover) == res); return cover;

7.6 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, {i}vis 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: O(V+E)

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

Usage: TwoSat ts(number of boolean variables);
TwoSat bicomps(F f) {
    num.assign(sz(ed), 0);
    rep(i,0,sz(ed))
        if (comp[i] < 0) dfs(i, g, f);
    for each edge (a,b) Usage:
i.e., not part of any cycle.
In a biconnected component there
Finds all biconnected components in an undirected graph,
Description:
BiconnectedComponents.h

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, i.e., not part of any cycle.
Usage: int eid = 0; ed.resize(N);
    for each edge (a,b) { ed[a].emplace_back(b, eid); ed[b].emplace_back(a, eid+1); } bicomps[](const vis &edgelist) [...]};
Time: O(E + V)

LCA.h
Description: Lowest common ancestor. Finds the lowest common ancestor in a tree (with 0 as root). C should be an adjacency list of the tree, either in a tree (with 0 as root). C should be an adjacency list of the tree, either
Lowest common ancestor. Finds the lowest common ancestor
LCA.h

TreePower.h
Description: Calculate power of two jumps in a tree. Assumes the root node points to itself.
Time: O(V|log|V|)

LCA.h
Description: Lowest common ancestor. Finds the lowest common ancestor in a tree (with 0 as root). C should be an adjacency list of the tree, either directed or undirected. Can also find the distance between two nodes.
Usage: LCA lca(undirGraph);
lca.query(firstNode, secondNode);
lca.distance(firstNode, secondNode);
Time: O(V|log|V| + Q)

typedef vector<pair<int, int>> tspi;
typedef vector<pair<int, int>> tsp;
typedef vector<T> tsp;

tspi tspi;
tsp tsp;
tsp tsp;

LCA(graph C) : time(sz(C) - 99), dist(sz(C)), rmq(dfs(C)) {}
Decomposes a tree into vertex disjoint heavy paths and light edges such that the path from any leaf to the root contains at most \( \log(n) \) edges.

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 \( \{ \text{par}, \text{val} \} \) representing a tree rooted at 0. The root points to itself.

**Time:** \( O(|S| \log(n)) \)

**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 \( \{ \text{par}, \text{val} \} \) representing a tree rooted at 0. The root points to itself.

**Time:** \( O(|S| \log(n)) \)
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.)

template <class T>
struct Point {
    typedef Point P;
    explicit Point(T x=0, T y=0) : x(x), y(y) {}
    T x, y;

    struct { const P& e1, const P& e2, int a1, int a2; }
    lineIntersectionP const; // intersection, dummy;
    int segmentIntersect; // segments intersect at * or section intersect && end;

    template <class P>
    double lineDist(const P& s1, const P& e1, const P& s2, const P& e2, P& r1, P& r2) {
        if (a1 == a2) {
            if (e1 == e2) { r1 = e1; return 1; } // all equal
            else return 0; // different point segments
        }
        return segmentIntersection(s2, e2, s1, e1, r1, r2); //swap
    }
}

Geometry (8)

7.8 Matrix tree theorem

MatrixTree.h
Description: To count the number of spanning trees in an undirected graph G: create an N x N matrix mat and for each edge (a, b) in G, do mat[a][a]++ , mat[b][b]++ , mat[a][b]-- , mat[b][a]-- . Remove the last row and column, and take the determinant.
8.2 Circles

Circles

The intersection point exists if Point P lies on the line segment from s to e. In-

tended for use with e.g. Point A and Point B.

Usage:

circleIntersection(A, B, C) // returns Point P or -1

Description:

Computes a pair of points at which two circles intersect.

Usage:

circleIntersection(A, B, C) // returns Point P or -1

Description:

Computes the minimum circle that encloses a set of points.

Usage:

MinimumEnclosingCircle(S) // returns Point A and Point B

Description:

Computes the minimum circle that encloses a set of points.

Usage:

MinimumEnclosingCircle(S) // returns Point A and Point B
8.3 Polygons

**insidePolygon.h**

Description: Returns true if p lies within the polygon described by the points between iterators begin and end. If strict false is returned when p is on the edge of the polygon. Answer is calculated by counting the number of intersections between the polygon and a line going from p to infinity in the positive x-direction. The algorithm uses products in intermediate steps so watch out for overflows. If points within an edge from an edge should be considered as on the edge replace the line "if (onSegment...)" with the comment below (this will cause overflows for int and long).

Usage: typedef Point<int> P;
vector<P> v; v.push_back(pi(4,4));

**ConvexHull.h**

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

Usage: vector<P> convexHull(vector<P>& poly);

**PointInsideHull.h**

Description: Determine whether a point t lies inside a given polygon (counter-clockwise order). The polygon must be such that every point on the circumference is visible from the first point in the vector. It returns 0 for points outside, 1 for points on the circumference, and 2 for points inside.

Time: \( O(n \log n) \)

**PolygonDiameter.h**

Description: Calculates the max squared distance of a set of points. For points outside, 1 for points on the circumference, and 2 for points inside.

Time: \( O(n + \log n) \)

**PolygonDiameter**

```
vector<P> res;
rep(i,0,sz(v)-1) a += v[i].cross(v[i+1]);
if (side = (s.cross(e, prev) < 0)) {
  res.push_back(cur);
  lineIntersection(a, e, prev, res.back());
}
if (side)
  res.push_back(cur);
return res;
```

**PointInsideHull**

```
if (len == 2) {
  int sa = sideOf(H[0], H[1], p);
  int sb = sideOf(H[1], H[2], p);
  int sc = sideOf(H[0], H[2], p);
  if (sa < 0 || sb < 0 || sc < 0) return 0;
  if (sa == 0 && sb == 0 && sc == 0) return 1;
}
return 2;
```

**LineHullIntersection**

Description: Line-convex polygon intersection. The polygon must be ccw and have no colinear points. isct(a, b) returns a pair describing the intersection of a line with the polygon: \( (x, 1) \) if a corner \( \cdot \) \((x, 1)\) if along side (\( x, 1 \) \) if crossing sides \((x, 1)\) and \((x, 1)\) in the last case, if a corner is crossed, this is treated as happening on side \((x, 1)\). The points are returned in the same order as the line hits the polygon.

Time: \( O(n + \log n) \)

```c++
for(i = 0; i < N; i++)
  if (cnt[i] == 1)
    res.push_back(cur);
for(i = N; i--;
  if (cnt[i] == 1)
    res.push_back(cur);
return res;
```
8.4 Misc. Point Set Problems

closestPair

description: i1, i2 are the indices to the closest pair of points in the point
vector p after the call. the distance is returned.
time: $O(n \log n)$

```cpp
// returns $i < j$ if $i < j$;
bool isect(P a, P b) {
    int t = b.x - a.y, s = bs(b-b-a);
    if ((a.x > b.y) || (t < s)) return false;
    if ((b.x < a.y) || (t > s)) return true;
    int x = b.x > a.y ? b.x : a.y;
    int y = b.x < a.y ? b.x : a.y;
    return x < y;
}
```

delauNayTriangulation

description: Computes the Delaunay triangulation of a set of points.

```
templated <class F, class P>
void delaunay(vector<P>& ps, F trifun) {
    if (sz(ps) >= 3) {
        Node* root = 0;
        for(P p : ps) root = make_pair(p, root);
        for(P p : ps) make_pair(p, root);
        for (Node* node : ps) {
            if (node->pt == p) return;
            Node* f = node->first, *s = node->second;
            if ((f->distance(p) <= s->distance(p)) && (p != node->pt)) {
                if (f->distance(p) > s->distance(p)) {
                    Node* tmp = f;
                    f = s;
                    s = tmp;
                }
                if (p.first > f->second) {
                    swap(f->second, s->first);
                    swap(f, s);
                }
                trifun(f->first, s->first, p);
                trifun(f->second, s->second, p);
            }
        }
    }
}
```

delaunayTriangulation.h

description: Computes the Delaunay triangulation of a set of points.
Each circumsphere contains none of the input points. If any three points are
colinear or any four are on the same circle, behavior is undefined.
time: $O(n^2)$

```
templated <class F, class P>
void delaunay(vector<P*>& ps, T trifun) {
    if (sz(ps) >= 3) {
        Node* root = 0;
        for(P p : ps) root = make_pair(p, root);
        for(P p : ps) make_pair(p, root);
        for (Node* node : ps) {
            if (node->pt == p) return;
            Node* f = node->first, *s = node->second;
            if ((f->distance(p) <= s->distance(p)) && (p != node->pt)) {
                if (f->distance(p) > s->distance(p)) {
                    Node* tmp = f;
                    f = s;
                    s = tmp;
                }
                if (p.first > f->second) {
                    swap(f->second, s->first);
                    swap(f, s);
                }
                trifun(f->first, s->first, p);
                trifun(f->second, s->second, p);
            }
        }
    }
}
```

8.5 3D

polyhedronVolume

description: Magic formula for the volume of a polyhedron. Faces should point
outwards.
time: $O(n^3)$

```
templated <class V, class T>
double signedPolyVolume(const V* v, const L* trilist) {
    int v = 0;
    for (L* l : trilist) {
        v += cross(l[0], l[1], l[2]).dot(l[3].c);
    }
    return v / 6;
}
```

Compute all faces of the 3-dimension hull of a point set.

**Description:**
- Includes `Point3D.h` and `Point3D.h`.
- The class `Point3D` has a member function `Point3D P3;`.
- It provides methods for operations like `normal()`, `unit()`, `theta()`, `dist()`, `dist2()`, `cross()`, `dot()`, and more.
- The `Point3D` class defines the basic Euclidean geometry operations for 3D points.

```cpp
// returns unit vector normal to
// returns point rotated 'angle' radians ccw around axis
// Zenith angle (latitude) to the z
const P normal(P p)
const P unit()
double theta()
double dist()
double dist2()
const T dist2();

P cross(R p)
const T dot(R p)
P P

bool operator == (R p)
explicit Point3D(T x=0, T y=0, T z=0) : x(x), y(y), z(z) {}
T x, y, z;
typedef const T P3;
```

3D Hull Description:
- Computes all faces of the 3-dimensional hull of a point set.
- Takes a vector of `Point3D` objects as input.
- The implementation uses the `Point3D` class defined in `Point3D.h`.
- The `3dHull.h` header file contains the implementation details.

```cpp
void rotate(vector<point3d> &v, double angle)
```

Spherical Distance Description:
- Returns the shortest distance on the sphere with radius `ra` between two points on the x-axis.
- The distance is calculated using the spherical distance formula.

```cpp
//makes dist() = 1
(radius*2)*asin(d/2);
```

KMP Description:
- Finds the longest prefix of a string `s` that is also a suffix of the string.
- Uses the KMP algorithm to achieve linear-time complexity.

```cpp
//this is just 3 times faster than stl sort for N=10^6
int mask = (1 << bits) - 1;
```

Strings (9)

**KMP**
- **Description:**
- Computes the length of the longest prefix of `s` that ends at `x`, other than `s[0..x-1]`.
- This is used by find to find all occurrences of a string.

**Usage:**
- `vi pi = pi(pattern);`
- `vi occ = find(word, pi);`

**Time:**
- `O(pattern)` for `pi`, `O(word + pattern)` for find

**Strings (9)**
- **KMP**
- **Description:**
- Computes the length of the longest prefix of `s` that ends at `x`, other than `s[0..x-1]`.
- This is used by find to find all occurrences of a string.

**Usage:**
- `vi pi = pi(pattern);`
- `vi occ = find(word, pi);`

**Time:**
- `O(pattern)` for `pi`, `O(word + pattern)` for find

```cpp
// KMP
```

**Suffix Array**
- **Description:**
- Builds a suffix array for a string, where `a[n]` is the starting index of the suffix which is `a[i]` in the input string.
- The returned vector is of size `n+1`, and `w[0]=n`.

**Memory:**
- `O(N)`

**Time:**
- `O(N log^2 N)` where `N` is the length of the string for creation of the SA. `O(N)` for longest common prefixes.

```cpp
//this is just 3 times faster than stl sort for N=10^6
int mask = (1 << bits) - 1;
```

**SuffixArray**
- **Description:**
- Builds a suffix array for a string, where `a[n]` is the starting index of the suffix which is `a[i]` in the input string.
- The returned vector is of size `n+1`, and `w[0]=n`.

**Memory:**
- `O(N)`

**Time:**
- `O(N log^2 N)` where `N` is the length of the string for creation of the SA. `O(N)` for longest common prefixes.
int q = 8;
while (I <= q <= N) q++;
for (int moc = 0; moc < N; moc++)
    count_sort(b, q); // sort(all(b)) can be used as well
a[b][second] = 0;
rep(i,1,N)
a[i].second = a[i - 1].second + 
(b[i].first == b[i].first);
}
for (int moc = 0; moc < N; moc++)
    count_sort(b, q); // sort(all(b)) can be used as well
a[b][second] = 0;
rep(i,1,N)
a[i].second = a[i - 1].second + 
(b[i].first == b[i].first);
}
for (int moc = 0; moc < N; moc++)
    count_sort(b, q); // sort(all(b)) can be used as well
a[b][second] = 0;
rep(i,1,N)
a[i].second = a[i - 1].second + 
(b[i].first == b[i].first);
}
for (int moc = 0; moc < N; moc++)
    count_sort(b, q); // sort(all(b)) can be used as well
a[b][second] = 0;
rep(i,1,N)
a[i].second = a[i - 1].second + 
(b[i].first == b[i].first);
}
Various (10)

10.1 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 \([\text{inclusive}, \text{exclusive})\).
Time: \(O(\log N)\) 25 lines

```cpp
// template< class T >
auto addInterval(set<pair<T, T>> &is, T L, T R) {
    auto it = is.lower_bound({L, R});
    if (it != is.end()) {
        int mid = (it->first + it->second) / 2;
        rec(mid+1, to, f, g, i, p, f(mid));
        rec(from, mid, f, g, i, p, f(mid));
    }
    is.insert({L, R});
    return is.end();
}
```

IntervalCover.h
Description: Compute indices of smallest set of intervals covering another interval. Should be [inclusive, exclusive]. To support [inclusive, exclusive] change (A) to add \([\text{inclusive}, \text{exclusive})\). To support \([\text{inclusive}, \text{exclusive})\).
Time: \(O(N \log N)\) 19 lines

```cpp
template<class T, class C> const C lcs(T S, const C &T, const C &T) {
    int a = sz(S), b = sz(T);
    vector<vi> dp(a+1, vi(b+1));
    rep(i,1,a+1) rep(j,1,b+1) {
        dp[i][j] = X[i-1]==Y[j-1] ? dp[i-1][j-1]+1 : max(dp[i-1][j],dp[i][j-1]);
    }
    return dp;  // (A)
}
```

10.2 Misc. algorithms

TernarySearch.h
Description: Find the smallest \(i\) in \([a,b]\) that maximizes \(f(i)\), assuming that \(f(a) < \cdots < f(i) \geq \cdots \geq f(b)\). To reverse which of the sides allows non-strict inequalities, change the \(\geq\) to \(\leq\), and reverse the loop at (B). To minimize \(f\), change it to \(>\), also at (B).
Usage: \(\text{int ind = ternsearch}(0,n-1,\{\text{int i}\} \{\text{return a[i]}\});\)
Time: \(O(\log n)\) 13 lines

```cpp
template<class T, class F, class G, class C> const C ternSearch(int i, int j, T &f, G &g, C &c) {
    int id = ternSearch(0,n-1,\{\text{int i}\} \{\text{return a[i]}\});
    return a[id];
}
```

Karatsuba.h
Description: Faster-than-naive convolution of two sequences: \(c[x] = \sum_{a+b=x} aX^b \cdot X^{d} = X^x + bd + \{a+b+n\}d-bdX\). Doesn’t handle sequences of very different length well. See also FFT, under the Numerical chapter.
Time: \(O(N \log 3)\) 1 lines

```cpp
template<class T> vi lis(vector<T> &v) {
    vi prev(sz(S));
    typedef pair<T, int> p;
    vector<p> res;
    rep(i,0,sz(S)) {
        p el { S[i], i };  // (A)
        if (it == res.end()) res.push_back(el), it = --res.end();
        if (it->first != res.end()) it->second = L;
        if (it->first == L) it = res.begin();
        if (it != res.begin()) it = (--it)->second;
        int L = sz(res), cur = res.back().second;
        if (cur < L) return res;
        return ans;
    }
```
10.3 Dynamic programming

DivideAndConquerDP.h

Description: Given $s[i] = \min_{k<i} \{ s[0] + c_{k,i} (f(i,k)) \}$ where the (minimal) $k$ increases with $i$, computes $s[i]$ for $i = L \ldots R - 1$.

Time: $O((N + (hi-lo)) \log N)$

struct DP // Modify at will:
int lo(int ind) { return 0; }
int hi(int ind) { return 1; }
nl f(int ind, k) return dp[ind][k];
void store(int ind, k, nl v) { res[ind] = pii(k, v); }

void rec(int l, int r, int lo, int hi) {
if (L >= R) return;
mid = (l + R) >> 1;

if (L >= R) return;
mid = (l + R) >> 1;
pair<nl, int> best = make_pair(f(mid, k), mid);

best = min(best, make_pair(f(mid, k), mid));

rec(lo, mid, hi, best.second);
rec(mid, R, best.second+1);

return (best.first, best.second);
}

void solve(int l, int r) { rec(l, R, INT_MIN, INT_MAX); }

KnuthDP.h

Description: When doing DP on intervals: KnuthDP.h

DP {
// Modify at will:

// Either globally or in a single class:

static char buf[450 << 20];
void operator new[](size_t s) {
static size_t i = 0;

struct ptr {
unsigned ind;
T operator()(T* p = 0) const { return *(T*) (buf + buf_ind); }
T operator[](int i) const { return *(T*) (buf + buf_ind); }
}

secret_small() {}
public static T* operator new[](size_t s) {
return (T*) (buf + buf_ind); }
}

select_vectorized code, but causes crashes on old machines.

#pragma GCC optimize("Ofast") will make GCC auto-vectorize for loops and optimizes floating point better (assumes associativity and turns off denormals).

#pragma GCC target ("avx,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).

BumpAllocator.h

Description: When you need to dynamically allocate many objects and
don't care about freeing them. "New X" otherwise has an overhead of something like 0.05ms + 16 bytes per allocation.

template<T> struct ptr {
unsigned ind;
T operator()(T* p = 0) const { return *(T*) (buf + buf_ind); }
T operator[](int i) const { return *(T*) (buf + buf_ind); }
}

explicit operator bool() const { return ind; }
}

class BumpAllocatorSTL {

public:

// Methods that work with most data types (append e.g. _epi32):

// andnot, abs, min, max, sign(z), cmpf, eq, unpack(0,0)
int sum32(mi m) { union {int v[4]; mi m;} u; u.m = acc; rep(i,0,4) r += u.v[i];
return ret; }
bool all_zero(mi m) { return _mm256_testz_si256(m, m); }
bool all_one(mi m) { return _mm256_testz_si256(m, m); }

10.4 Debugging tricks

• signal(SIGSEGV, [int] { _exit(0); });

• sends segfaults into Wrong Answers. Similarly one

can catch SIGABRT (assertion failures) and SIGFPE

(zero divisions).

• feasibleeless(29); kills the program on NaNs

(1), 0-divs (4), infinities (8) and denormals (16).

10.5 Optimization tricks

10.5.1 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).

• $c = \lfloor x \rfloor, r = x + \lfloor c \rfloor$; $((r-x) > 2)/c$ or $r$

is the next number after $x$ with the same number of bits set.
Techniques (A)

Recursion

Divide and conquer
Finding interesting points in N log N
Algorithm analysis

Master theorem
Amortized time complexity

Greedy algorithm
Scheduling
Max contiguous subvector sum
Invariants

Huffman encoding

Graph theory

Dynamic graphs (extra book-keeping)
Breadth first search
Depth first search
- Normal trees / DFS trees
- Djikstra's algorithm
MST: Prím's algorithm
Bellman-Ford
König's theorem and vertex cover
Min-cost max flow
Lovasz togle
Matrix tree theorem
Maximal matching, general graphs
Hopcroft-Karp
Hall's marriage theorem

Graphical sequences

Floyd-Warshall
Eulercykler
Flow networks
- Augmenting paths
- Edmonds-Karp
Bipartite matching
Min. path cover
Topological sorting

Strongly connected components

2-SAT
Cutvertices, cutedges och biconnected components

Edge coloring
- Trees
- Vertex coloring
- Bipartite graphs (=> trees)
- 3^n (special case of set cover)

Diameter and centroid
K'th shortest path

Shortest cycle

Dynamic programming

Knapsack
Coin change

Longest common subsequence
Longest increasing subsequence
Number of paths in a dag
Shortest path in a dag
Dynprog over intervals
Dynprog over subsets
Dynprog over probabilities
Dynprog over trees

3^n set cover
Divide and conquer
Knuth optimization
Convex hull optimizations
RMQ (sparse table a.k.a 2^k-jumps)

Bitonic cycle
Log partitioning (loop over most restricted)

Combinatorics

Computation of binomial coefficients
Pigeon-hole principle
Inclusion/exclusion
Catalan number
Pick's theorem
Number theory
Integer parts
Divisibility
Euclidean algorithm
Modular arithmetic
- Modular multiplication
- Modular inverses
- Modular exponentiation by squaring
Chinese remainder theorem
Fermat's small theorem
Euler's theorem
Phi function

Froebenius number
Quadratic reciprocity
Pollard-Rho
Miller-Rabin
Hensel lifting
Vieta root jumping

Game theory

Combinatorial games
Game trees
Mini-max
Nim
Games on graphs
Games on graphs with loops
Grundy numbers
Bipartite games without repetition
General games without repetition
Alpha-beta pruning

Probability theory

Optimization

Binary search
Ternary search

Unimodality and convex functions

Binary search on derivative

Numerical methods

Numeric integration
Newton's method
Root-finding with binary/ternary search
Golden section search

Matrices

Gaussian elimination
Exponentiation by squaring

Sorting

Radix sort

Geometry

Coordinates and vectors
- Cross product
- Scalar product
Convex hull
Polygon cut
Closest pair
Coordinate-compression
Quadtrees
KD-trees

All segment-segment intersection

Sweeping

Discretization (convert to events and sweep)
Angle sweeping
Line sweeping

Discrete second derivatives

Strings

Longest common substring
Palindrome subsequences

Knuth-Morris-Pratt
Tries
Rolling polynomial hashes
Suffix array
Suffix tree
Aho-Corasick

Manacher's algorithm
Letter position lists

Combinatorial search
Meet in the middle
Brute-force with pruning
Best-first (A*)

Bidirectional search
Iterative deepening DFS / A*

Data structures

LCA (2^k-jumps in trees in general)
Full/push-technique on trees
Heavy-light decomposition
Centroid decomposition
Lazy propagation
Self-balancing trees

Convex hull trick (wclpeg.com/wiki/Convex_hull_trick)

Monotone queues / monotone stacks / sliding queues
Sliding queue using 2 stacks

Persistent segment tree