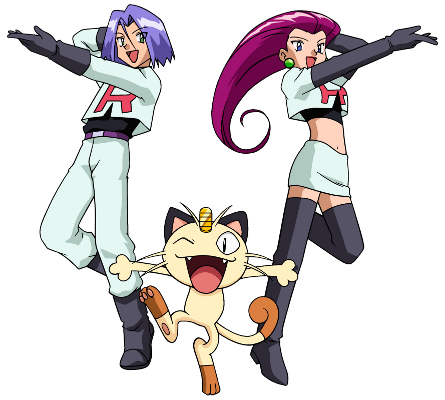


University of Bucharest

Echipa Lacheta

Lucian Bicsi, Andrei Popa, Alexandru Velea

ACM-ICPC World Finals 2018

April 2018

Contest (1)

template.cpp 15 lines

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

#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define trav(a, x) for(auto& a : x)
#define all(x) x.begin(), x.end()
#define sz(x) (int)(x).size()
typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;

int main() {
  cin.sync_with_stdio(0); cin.tie(0);
  cin.exceptions(cin.failbit);
}
```

.bashrc 3 lines

```
alias c='g++ -Wall -Wconversion -Wfatal-errors -g -std=c++14 \
  -fsanitize=undefined,address'
xmodmap -e 'clear lock' -e 'keycode 66=less greater' #caps = <>
```

.vimrc 2 lines

```
set cin aw ai is ts=4 sw=4 tm=50 nu noeb bg=dark ru cul
sy on   |   im jk <esc>   |   im kj <esc>   |   no ; :
```

troubleshoot.txt 52 lines

```
Pre-submit:
Write a few simple test cases, if sample is not enough.
Are time limits close? If so, generate max cases.
Is the memory usage fine?
Could anything overflow?
Make sure to submit the right file.

Wrong answer:
Print your solution! Print debug output, as well.
Are you clearing all datastructures between test cases?
Can your algorithm handle the whole range of input?
Read the full problem statement again.
Do you handle all corner cases correctly?
Have you understood the problem correctly?
Any uninitialized variables?
Any overflows?
Confusing N and M, i and j, etc.?
Are you sure your algorithm works?
What special cases have you not thought of?
Are you sure the STL functions you use work as you think?
Add some assertions, maybe resubmit.
Create some testcases to run your algorithm on.
Go through the algorithm for a simple case.
Go through this list again.
Explain your algorithm to a team mate.
Ask the team mate to look at your code.
Go for a small walk, e.g. to the toilet.
Is your output format correct? (including whitespace)
Rewrite your solution from the start or let a team mate do it.

Runtime error:
Have you tested all corner cases locally?
Any uninitialized variables?
Are you reading or writing outside the range of any vector?
Any assertions that might fail?
Any possible division by 0? (mod 0 for example)
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)
How big is the input and output? (consider scanf)
Avoid vector, map. (use arrays/unordered_map)
What do your team mates think about your algorithm?

Memory limit exceeded:
What is the max amount of memory your algorithm should need?
Are you clearing all datastructures between test cases?
```

Mathematics (2)

2.1 Equations

$$\begin{aligned} ax+by&=e \\ cx+dy&=f \end{aligned} \Rightarrow \begin{aligned} x&=\frac{ed-bf}{ad-bc} \\ y&=\frac{af-ec}{ad-bc} \end{aligned}$$

In general, given an equation $Ax = b$, the solution to a variable x_i is given by

$$x_i = \frac{\det A_i'}{\det A}$$

where A_i' is A with the i'th column replaced by b.

2.2 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. $a_n = (d_1 n + d_2) r^n$.

2.3 Trigonometry

$$\sin(v+w) = \sin v \cos w + \cos v \sin w$$
$$cos(v+w) = \cos v \cos w - \sin v \sin w$$

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

where V, W are lengths of sides opposite angles v, w.

$$a \cos x + b \sin x = r \cos(x - \phi)$$
$$a \sin x + b \cos x = r \sin(x + \phi)$$

where $r = \sqrt{a^2+b^2}, \phi = \operatorname{atan2}(b, a)$.

2.4 Geometry

2.4.1 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):

$$s_a = \sqrt{bc\left[1 - \left(\frac{a}{b+c}\right)^2\right]}$$

Law of sines: $\dfrac{\sin\alpha}{a} = \dfrac{\sin\beta}{b} = \dfrac{\sin\gamma}{c} = \dfrac{1}{2R}$

Law of cosines: $a^2 = b^2 + c^2 - 2bc\cos\alpha$

Law of tangents: $\dfrac{a+b}{a-b} = \dfrac{\tan\frac{\alpha+\beta}{2}}{\tan\frac{\alpha-\beta}{2}}$

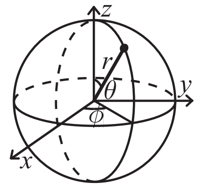
2.4.2 Quadrilaterals

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

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

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

2.4.3 Spherical coordinates



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

2.5 Linear algebra

2.5.1 Matrix inverse

The inverse of a 2x2 matrix:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

In general:

$$A^{-1} = \frac{1}{det(A)}A^*$$

where $A^*_{i,j} = (-1)^{i+j} * \Delta_{i,j}$ and $\Delta_{i,j}$ is the determinant of matrix A crossing out line i and column j.

2.6 Derivatives/Integrals

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

Integration by parts:

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

2.7 Sums

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

$$1+2+3+\cdots+n = \frac{n(n+1)}{2}$$
$$1^2+2^2+3^2+\cdots+n^2 = \frac{n(2n+1)(n+1)}{6}$$
$$1^3+2^3+3^3+\cdots+n^3 = \frac{n^2(n+1)^2}{4}$$
$$1^4+2^4+3^4+\cdots+n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

2.8 Series

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

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) $\mu = \mathbb{E}(X) = \sum_x xp_X(x)$ 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,

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

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 $\mathrm{Bin}(n,p)$, $n = 1,2,\ldots$, $0 \leq p \leq 1$.

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

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

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

First success distribution

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

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

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

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 $\mathrm{Po}(\lambda)$, $\lambda = t\kappa$.

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

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

2.9.2 Continuous distributions

Uniform distribution

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

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

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

Exponential distribution

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

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

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

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 $t_i = 1 + \sum_{k\in\mathbf{G}} p_{ki} t_k$.

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

16 lines

```
#include <bits/extc++.h>
using namespace __gnu_pbds;

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
}
```

SegmentTree.h

Description: Very fast and quick segment tree. Only useful for easy invariants. 0-indexed. Range queries are half-open.

18 lines

```
struct SegmTree {
  vector<int> T; int n;
  SegmTree(int n) : T(2 * n, (int)-2e9), n(n) {}

  void Update(int pos, int val) {
    for (T[pos += n] = val; pos > 1; pos /= 2)
      T[pos / 2] = max(T[pos], T[pos ^ 1]);
  }

  int Query(int b, int e) {
    int res = -2e9;
    for (b += n, e += n; b < e; b /= 2, e /= 2) {
      if (b % 2) res = max(res, T[b++]);
      if (e % 2) res = max(res, T[--e]);
    }
    return res;
  }
};
```

LineContainer.h

Description: Container where you can add lines of the form ax+b, and query maximum values at points x. For each line, also keeps a value p, which is the last (maximum) point for which the current line is dominant. (obviously, for the last line, p is infinity) Useful for dynamic programming.
Time: $\mathcal{O}(\log N)$

<bits/stdc++.h> 35 lines

```
using T = long long;
bool QUERY;
struct Line {
  mutable T a, b, p;
  T Eval(T x) const { return a * x + b; }
  bool operator<(const Line& o) const {
    return QUERY ? p < o.p : a < o.a;
  }
};

struct LineContainer : multiset<Line> {
  // for doubles, use kInf = 1/.0, div(a, b) = a / b
  const T kInf = numeric_limits<T>::max();
  T div(T a, T b) { // floored division
    return a / b - ((a ^ b) < 0 && a % b); }
  bool isect(iterator x, iterator y) {
    if (y == end()) { x->p = kInf; return false; }
    if (x->a == y->a) x->p = x->b > y->b ? kInf : -kInf;
    else x->p = div(y->b - x->b, x->a - y->a);
    return x->p >= y->p;
  }
  void InsertLine(T a, T b) {
    auto nx = insert({a, b, 0}), it = nx++, pv = it;
    while (isect(it, nx)) nx = erase(nx);
    if (pv != begin() && isect(--pv, it)) isect(pv, it = erase(
        it));
    while ((it = pv) != begin() && (--pv)->p >= it->p)
      isect(pv, erase(it));
  }
  T EvalMax(T x) {
    assert(!empty());
    QUERY = 1; auto it = lower_bound({0,0,x}); QUERY = 0;
    return it->Eval(x);
  }
};
```

ConvexTree.h

Description: Container where you can add lines of the form a * x + b, and query maximum values at points x. Useful for dynamic programming. To change to minimum, either change the sign of all comparisons, the initialization of T and max to min, or just add lines of form (-a)*X + (-b) instead and negate the result.
Time: $\mathcal{O}(\log(kMax - kMin))$

<bits/stdc++.h> 50 lines

```
using int64 = int64_t;

struct Line {
  int a; int64 b;
  int64 Eval(int x) { return 1LL * a * x + b; }
};
const int64 kInf = 2e18; // Maximum abs(A * x + B)
const int kMin = -1e9, kMax = 1e9; // Bounds of query (x)

struct ConvexTree {
  struct Node { int l, r; Line line; };
  vector<Node> T = { Node{0, 0, {0, -kInf}} };
  int root = 0;

  int update(int node, int b, int e, Line upd) {
    if (node == 0) {
      T.push_back(Node{0, 0, upd});
      return T.size() - 1;
    }

    auto& cur = T[node].line;
    if (cur.Eval(b)>=upd.Eval(b) && cur.Eval(e)>=upd.Eval(e))
      return node;
    if (cur.Eval(b)<=upd.Eval(b) && cur.Eval(e)<=upd.Eval(e))
      return cur = upd, node;
    int m = (b + e) / 2;
    if (cur.Eval(b) < upd.Eval(b)) swap(cur, upd);
    if (cur.Eval(m) >= upd.Eval(m)) {
      int res = update(T[node].r, m + 1, e, upd);
      T[node].r = res; // DO NOT ATTEMPT TO OPTIMIZE
    } else {
      swap(cur, upd);
      int res = update(T[node].l, b, m, upd);
      T[node].l = res; // DO NOT ATTEMPT TO OPTIMIZE
    }
    return node;
  }
  void AddLine(Line l) { root = update(root, kMin, kMax, l); }

  int64 query(int node, int b, int e, int x) {
    int64 ans = T[node].line.Eval(x);
    if (node == 0) return ans;
    int m = (b + e) / 2;
    if (x <= m) ans = max(ans, query(T[node].l, b, m, x));
    if (x > m) ans = max(ans, query(T[node].r, m + 1, e, x));
    return ans;
  }
  int64 QueryMax(int x) { return query(root, kMin, kMax, x); }
};
```

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. It can support several operations, including lazy propagation (sample reverse operation below). Can be made persistent, by making a copy at pull function To transform into ordered set, uncomment line at (A) and delete the subtraction logic at (B).
Time: $\mathcal{O}(\log N)$ expected time per operation 109 lines

```
namespace Treap {
  struct Node {
    int val, pri;
    int left = 0, right = 0, subsize = 0, lazy = 0;
    Node(int val, int pri) : val(val), pri(pri) {}
  };
  vector<Node> T(1, Node(-1, -1));

  int get_key(int node) {
    return T[T[node].left].subsize;
    // return T[node].val; (A)
  }

  int pull(int node) {
    if (node == 0) return 0;

    T[node].subsize = T[T[node].left].subsize
      + T[T[node].right].subsize + 1;

    return node;
  }

  int push(int node) {
    int& lazy = T[node].lazy;
    if (node == 0 or lazy == 0) return node;

    swap(T[node].left, T[node].right);
    T[T[node].left].lazy ^= lazy;
    T[T[node].right].lazy ^= lazy;
    lazy = 0;

    return node;
  }
// Splits into < key and >= key
pair<int, int> Split(int node, int key) {
  push(node);
  if (node == 0) return {0, 0};

  int l, r;
  if (get_key(node) < key) {              /* (B) */
    tie(l, r) = Split(T[node].right, key-get_key(node)-1);
    T[node].right = l;
    return {pull(node), r};
  } else {
    tie(l, r) = Split(T[node].left, key);
    T[node].left = r;
    return {l, pull(node)};
  }
}

// keys(node1) <= keys(node2) is REQUIRED
int Join(int node1, int node2) {
  push(node1); push(node2);
  if (!node1) return node2;
  if (!node2) return node1;

  if (T[node1].pri > T[node2].pri) {
    T[node1].right = Join(T[node1].right, node2);
    return pull(node1);
  } else {
    T[node2].left = Join(node1, T[node2].left);
    return pull(node2);
  }
}

// Can be any foreach function
void Dump(int node) {
  push(node);
  if (node == 0) return;

  Dump(T[node].left);
  cout << T[node].val << " ";
  Dump(T[node].right);
}

int Single(int value) {
  int node = T.size();
  T.push_back(Node(value, rand()));
  return pull(node);
}

// Only makes sense for cartesian tree
tuple<int, int, int> Slice(int node, int b, int e) {
  int l, m, r;
  tie(m, r) = Split(node, e);
  tie(l, m) = Split(m, b);
  return make_tuple(l, m, r);
}

int Find(int node, int key) {
  int l, m, r;
  tie(l, m, r) = Slice(node, key, key + 1);
  assert(node == Join(l, Join(m, r)));
  return m;
}

int Insert(int node, int key, int value) {
  int l, r, m = Single(value);
  tie(l, r) = Split(node, key);
  return Join(l, Join(m, r));
}
  int Reverse(int node) {
    T[node].lazy ^= 1;
    return push(node);
  }

}
```

FenwickTree.h

Description: Adds a value to a (half-open) range and computes the sum on a (half-open) range. Beware of overflows!
Time: Both operations are $\mathcal{O}(\log N)$. 27 lines

```
#define int long long
struct FenwickTree {
  int n;
  vector<int> T1, T2;

  FenwickTree(int n) : n(n), T1(n + 1, 0), T2(n + 1, 0) {}

  void Update(int b, int e, int val) {
    if (e != -1)
      return Update(e, -1, -val), Update(b, -1, +val);

    int c1 = val, c2 = val * (b - 1);
    for (int pos = b + 1; pos <= n; pos += (pos & -pos)) {
      T1[pos] += c1; T2[pos] += c2;
    }
  }

  int Query(int b, int e) {
    if (b != 0) return Query(0, e) - Query(0, b);

    int ans = 0;
    for (int pos = e; pos; pos -= (pos & -pos)) {
      ans += T1[pos] * (e - 1) - T2[pos];
    }
    return ans;
  }
};
```

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" 32 lines

```
struct Fenwick2D {
  vector<vector<int>> ys;
  vector<vector<int>> T;
  Fenwick2D(int n) : ys(n + 1) {}

  void FakeUpdate(int x, int y) {
    for (++x; x < (int)ys.size(); x += (x & -x))
      ys[x].push_back(y);
  }
  void Init() {
    for (auto& v : ys) {
      sort(v.begin(), v.end());
      T.emplace_back(v.size());
    }
  }
  int ind(int x, int y) {
    auto it = lower_bound(ys[x].begin(), ys[x].end(), y);
    return distance(ys[x].begin(), it);
  }
  void Update(int x, int y, int val) {
    for (++x; x < (int)ys.size(); x += (x & -x))
    for (int i = ind(x,y); i < (int)T[x].size(); i += (i & -i))
```

```
      trees[x][i] = trees[x][i] + val;
  }
  int Query(int x, int y) {
    int sum = 0;
    for (; x > 0; x -= (x & -x))
    for (int i = ind(x,y); i > 0; i -= (i & -i))
      sum = sum + T[x][i];
    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: $\mathcal{O}(|V|\log|V| + Q)$ 21 lines

```
template <class T>
struct RMQ {
  const int kInf = numeric_limits<T>::max();
  vector<vector<T>> rmq;

  RMQ(const vector<T>& V) {
    int n = V.size(), on = 1, depth = 1;
    while (on < n) on *= 2, ++depth;
    rmq.assign(depth, V);
    for (int i = 0; i < depth - 1; ++i)
      for (int j = 0; j < n; ++j) {
        jmp[i + 1][j] = min(jmp[i][j],
          jmp[i][min(n - 1, j + (1 << i))]);
    }

  T Query(int a, int b) {
    if (b <= a) return kInf;
    int dep = 31 - __builtin_clz(b - a); // log(b − a)
    return min(rmq[dep][a], rmq[dep][b - (1 << dep)]);
  }
};
```

Numerical (4)

FracBinarySearch.h

Description: Does binary search on fractions having an upper bound on the numerator, given a predicate pred. The predicate should be a monotonous function going from negative to 0 to positive. The function will find a RANDOM fraction f for which pred(f) = 0 or THROW exception 5.
Time: $\mathcal{O}(\log((b-a)/\epsilon))$ 34 lines

```
using int64 = int64_t;
struct Frac { int64 a, b; };

template<typename Predicate>
Frac FracBinarySearch(int64 max_num, Predicate pred) {
  // Range is OPEN ON BOTH ENDS
  Frac lo{0, 1}, hi{1, 1}; // set to {1, 0} for (0...max_num)
  int sign = 1;

  // Number of tries should be >= 2 * log(max_num)
  for (int tries = 0; tries < 135; ++tries) {
    int64 adv = 0;
    bool down = false;

    for (int64 step = 1; step; down ? step /= 2 : step *= 2) {
      adv += step;

      Frac mid{lo.a * adv + hi.a, lo.b * adv + hi.b};
      if (abs(mid.a) > kLim or mid.b > max_num) {
        adv -= step; down = true; continue;
      }

      int64 res = pred(mid);
      if (res == 0) return mid;
      if (res * sign < 0) { adv -= step; down = true; }
    }

    hi.a += lo.a * adv;
    hi.b += lo.b * adv;
    sign = -sign;
    swap(lo, hi);
  }
  throw 5;
}
```

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. 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 = GoldenSectionSearch(-1000,1000,func);
Time: $\mathcal{O}(\log((b-a)/\epsilon))$ 15 lines

```
template<typename Func>
double GoldenSectionSearch(double a, double b, Func f) {
  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 + b) / 2;
}
```

HillClimbing.h

Description: Poor man's optimization for unimodal functions. Finds minimum of a function func[Point] => double. To change with maximum, change the comparison at (*) 16 lines

```
struct Point { double x, y; };

template<typename Func>
pair<double, Point> HillClimb(Point p, Func func) {
  double best = func(p);

  for (double step = 1e9; step > 1e-20; step /= 2)
    for (int it = 0; it < 100; ++it)
      for (int dx = -1; dx <= 1; ++dx)
        for (int dy = -1; dy <= 1; ++dy) {
          Point q = p; q.x += dx * step; q.y += dy * step;
          double now = func(q);
          if (best > now) { best = now; p = q; } // (*)
        }
  return make_pair(best, p);
}
```

Polynomial.h

Description: Different operations on polynomials. Should work on any field.
<bits/stdc++.h> 114 lines

```
using TElem = double;
using Poly = vector<TElem>;

TElem Eval(const Poly& P, TElem x) {
  TElem val = 0;
  for (int i = (int)P.size() - 1; i >= 0; --i)
    val = val * x + P[i];
  return val;
}

// Differentiation
Poly Diff(Poly P) {
  for (int i = 1; i < (int)P.size(); ++i)
    P[i - 1] = i * P[i];
  P.pop_back();
  return P;
}

// Integration
Poly Integrate(Poly p) {
  P.push_back(0);
  for (int i = (int)P.size() - 2; i >= 0; --i)
    P[i + 1] = P[i] / (i + 1);
  P[0] = 0;
  return P;
}

// Division by (X − x0)
Poly DivRoot(Poly P, TElem x0) {
  int n = P.size();
  TElem a = P.back(), b; P.back() = 0;
  for (int i = n--; i--; )
    b = P[i], P[i] = P[i + 1] * x0 + a, a = b;
  P.pop_back();
  return P;
}

// Multiplication modulo X^sz
Poly Multiply(Poly A, Poly B, int sz) {
  static FFTSolver fft;
  A.resize(sz, 0); B.resize(sz, 0);
  auto R = fft.Multiply(A, B);
  R.resize(sz, 0);
  return r;
}

// Scalar multiplication
Poly Scale(Poly P, TElem s) {
  for (auto& x : P)
    x = x * s;
  return P;
}

// Addition modulo X^sz
Poly Add(Poly A, Poly B, int sz) {
  A.resize(sz, 0); B.resize(sz, 0);
  for (int i = 0; i < sz; ++i)
    A[i] = A[i] + B[i];
  return A;
}

// **************************************************
// For Invert, Sqrt, size of argument should be 2^k
// **************************************************

Poly inv_step(Poly res, Poly P, int n) {
  auto res_sq = Multiply(res, res, n);
  auto sub = Multiply(res_sq, P, n);
  res = Add(Scale(res, 2), Scale(sub, -1), n);
  return res;
}

// Inverse modulo X^sz
// EXISTS ONLY WHEN P[0] IS INVERTIBLE
Poly Invert(Poly P) {
  assert(P[0].Get() == 1);
  Poly res(1, 1);          // i.e., P[0]^(-1)

  int n = P.size();
  for (int step = 2; step <= n; step *= 2) {
    res = inv_step(res, P, step);
  }

  // Optional, but highly encouraged
  auto check = Multiply(res, P, n);
  for (int i = 0; i < n; ++i) {
    assert(check[i].Get() == (i == 0));
  }
  return res;
};

// Square root modulo X^sz
// EXISTS ONLY WHEN P[0] HAS SQUARE ROOT
Poly Sqrt(Poly P) {
  assert(P[0].Get() == 1);
  Poly res(1, 1);          // i.e., P[0]^(-1)
  Poly inv(1, 1);          // i.e., P[0]^(1/2)

  int n = P.size();
  for (int step = 2; step <= n; step *= 2) {
    auto now = inv_step(inv, res, step);
    now = Multiply(P, move(now), step);
    res = Add(res, now, step);
    res = Scale(res, (kMod + 1) / 2);
    inv = inv_step(inv, res, step);
  }

  // Optional, but highly encouraged
  auto check = Multiply(res, res, n);
  for (int i = 0; i < n; ++i) {
    assert(check[i].Get() == P[i].Get());
  }
  return res;
}
```

PolyRoots.h

Description: Finds the real roots to a polynomial.
Usage: `Poly p = {2, -3, 1} // x^2 - 3x + 2 = 0`
`auto roots = GetRoots(p, -1e18, 1e18); // {1, 2}`

<bits/stdc++.h>, "Polynomial.h" 26 lines

```
vector<double> GetRoots(Poly p, double xmin, double xmax) {
  if (p.size() == 2) { return {-p.front() / p.back()}; }
  else {
    Poly d = Diff(p);
    vector<double> dr = GetRoots(d, xmin, xmax);
    dr.push_back(xmin - 1);
    dr.push_back(xmax + 1);
    sort(dr.begin(), dr.end());

    vector<double> roots;
    for (auto i = dr.begin(), j = i++; i != dr.end(); j = i++){
      double lo = *j, hi = *i, mid, f;
      bool sign = Eval(p, lo) > 0;
      if (sign ^ (Eval(p, hi) > 0)) {
        // for (int it = 0; it < 60; ++it) {
        while (hi - lo > 1e-8) {
          mid = (lo + hi) / 2, f = Eval(p, mid);
          if ((f <= 0) ^ sign) lo = mid;
          else hi = mid;
        }
        roots.push_back((lo + hi) / 2);
      }
    }
    return roots;
  }
}
```

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 $x[k] = c * \cos(k/(n-1) * \pi), k = 0 \ldots n-1$.
Time: $\mathcal{O}(n^2)$

<bits/stdc++.h>, "Polynomial.h" 15 lines

```
Poly Interpolate(vector<TElem> x, vector<TElem> y) {
  int n = x.size();
  Poly res(n), temp(n);
  for (int k = 0; k < n; ++k)
    for (int i = k + 1; i < n; ++i)
    y[i] = (y[i] - y[k]) / (x[i] - x[k]);
  TElem last = 0; temp[0] = 1;
  for (int k = 0; k < n; ++k)
  for (int i = 0; i < n; ++i) {
    res[i] = res[i] + y[k] * temp[i];
    swap(last, temp[i]);
    temp[i] = temp[i] - last * x[k];
  }
  return res;
}
```

BerlekampMassey.h

Description: Recovers any n-order linear recurrence relation from the first 2*n terms of the recurrence. Very useful for guessing linear recurrences after brute-force / backtracking the first terms. Should work on any field. Numerical stability for floating-point calculations is not guaranteed.
Usage: `BerlekampMassey({0, 1, 1, 3, 5, 11}) => {1, 2}`

<bits/stdc++.h>, "ModOps.h" 29 lines

```
vector<ModInt> BerlekampMassey(vector<ModInt> s) {
  int n = s.size();
  vector<ModInt> C(n, 0), B(n, 0);
  C[0] = B[0] = 1;

  ModInt b = 1; int L = 0;
  for (int i = 0, m = 1; i < n; ++i) {

    ModInt d = s[i];
    for (int j = 1; j <= L; ++j)
      d = d + C[j] * s[i - j];

    if (d.get() == 0) { ++m; continue; }

    auto T = C; ModInt coef = d * inv(b);
    for (int j = m; j < n; ++j)
      C[j] = C[j] - coef * B[j - m];

    if (2 * L > i) { ++m; continue; }

    L = i + 1 - L; B = T; b = d; m = 1;
  }

  C.resize(L + 1); C.erase(C.begin());
  for (auto& x : C) x = ModInt(0) - x;
  return C;
}
```

LinearRecurrence.h

Description: Generates the k-th term of a n-th order linear recurrence given the first n terms and the recurrence relation. Faster than matrix multiplication. Useful to use along with Berlekamp Massey.
Usage: `LinearRec<double>({0, 1}, {1, 1}).Get(k) gives k-th Fibonacci number (0-indexed)`
Time: $\mathcal{O}\left(n^2 log(k)\right)$ per query

<bits/stdc++.h> 43 lines

```
template<typename T>
struct LinearRec {
  using Poly = vector<T>;
  int n; Poly first, trans;

  // Recurrence is S[i] = sum(S[i-j-1] * trans[j])
  // with S[0..(n-1)] = first
  LinearRec(const Poly &first, const Poly &trans) :
    n(first.size()), first(first), trans(trans) {}

  Poly combine(Poly a, Poly b) {
    Poly res(n * 2 + 1, 0);
    // You can apply constant optimization here to get a
    // ~10x speedup
    for (int i = 0; i <= n; ++i)
      for (int j = 0; j <= n; ++j)
        res[i + j] = res[i + j] + a[i] * b[j];

    for (int i = 2 * n; i > n; --i)
      for (int j = 0; j < n; ++j)
        res[i - 1 - j] = res[i - 1 - j] + res[i] * trans[j];

    res.resize(n + 1);
    return res;
  }

  // Consider caching the powers for multiple queries
  T Get(int k) {
    Poly r(n + 1, 0), b(r);
    r[0] = 1; b[1] = 1;

    for (++k; k; k /= 2) {
      if (k % 2)
        r = combine(r, b);
      b = combine(b, b);
    }

    T res = 0;
    for (int i = 0; i < n; ++i)
      res = res + r[i + 1] * first[i];
    return res;
  }
};
```

FFT.h

Description: Fast Fourier transform. Also includes a function for convolution: `conv(a, b) = c`, where $c[x] = \sum a[i]b[x-i]$. a and b should be of roughly equal size. Does about 1.2s for 10^6 elements. Rounding the results of conv works if $(|a| + |b|)\max(a, b) < \sim 10^9$ (in theory maybe 10^6); you may want to use an NTT from the Number Theory chapter instead.
Time: $\mathcal{O}(N \log N)$

<bits/stdc++.h> 76 lines

```
struct FFTSolver {
  using Complex = complex<double>;
  const double kPi = 4.0 * atan(1.0);
vector<int> rev;

int __lg(int n) { return n == 1 ? 0 : 1 + __lg(n / 2); }

void compute_rev(int n, int lg) {
  rev.resize(n); rev[0] = 0;
  for (int i = 1; i < n; ++i) {
    rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (lg - 1));
  }
}

vector<Complex> fft(vector<Complex> V, bool invert) {
  int n = V.size(), lg = __lg(n);
  if ((int)rev.size() != n) compute_rev(n, lg);

  for (int i = 0; i < n; ++i) {
    if (i < rev[i])
      swap(V[i], V[rev[i]]);
  }

  for (int step = 2; step <= n; step *= 2) {
    const double ang = 2 * kPi / step;
    Complex eps(cos(ang), sin(ang));
    if (invert) eps = conj(eps);

    for (int i = 0; i < n; i += step) {
      Complex w = 1;
      for (int a = i, b = i+step/2; b < i+step; ++a, ++b) {
        Complex aux = w * V[b];
        V[b] = V[a] - aux;
        V[a] = V[a] + aux;
        w *= eps;
      }
    }
  }

  return V;
}

vector<Complex> transform(vector<Complex> V) {
  int n = V.size();
  vector<Complex> ret(n);
  Complex div_x = Complex(0, 1) * (4.0 * n);

  for (int i = 0; i < n; ++i) {
    int j = (n - i) % n;
    ret[i] = (V[i] + conj(V[j]))
      * (V[i] - conj(V[j])) / div_x;
  }

  return ret;
}

vector<int> Multiply(vector<int> A, vector<int> B) {
  int n = A.size() + B.size() - 1;
  vector<int> ret(n);
  while (n != (n & -n)) ++n;

  A.resize(n); B.resize(n);
  vector<Complex> V(n);
  for (int i = 0; i < n; ++i) {
    V[i] = Complex(A[i], B[i]);
  }

  V = fft(move(V), false);
  V = transform(move(V));
  V = fft(move(V), true);

  for (int i = 0; i < (int)ret.size(); ++i)
    ret[i] = round(real(V[i]));
  return ret;
}
};
```

FST.h

Description: Fast Subset transform. Useful for performing the following convolution: R[a op b] += A[a] * B[b], where op is either of AND, OR, XOR. P has to have size $N = 2^n$, for some n.
Time: $\mathcal{O}(N \log N)$

<bits/stdc++.h> 16 lines

```
vector<int> Transform(vector<int> P, bool inv) {
  int n = P.size();
  for (int step = 1; step < n; step *= 2) {
    for (int i = 0; i < n; i += 2 * step) {
      for (int j = i; j < i + step; ++j) {
        int u = P[j], v = P[j + step];
        tie(P[j], P[j + step]) =
        inv ? make_pair(v - u, u) : make_pair(v, u + v); // AND
        inv ? make_pair(v, u - v) : make_pair(u + v, u); // OR
        make_pair(u + v, u - v);                         // XOR
      }
    }
  }
  // if (inv) for (auto& x : P) x /= n; // XOR only
  return P;
}
```

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.

9 lines

```
template<typename Func>
double Quad(Func f, double a, double b) {
  const int n = 1000;
  double h = (b - a) / 2 / n;
  double v = f(a) + f(b);
  for (int i = 1; i < 2 * n; ++i)
    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 z, y;
double h(double x) { return x*x + y*y + z*z <= 1; }
double g(double y) { ::y = y; return Quad(h, -1, 1); }
double f(double z) { ::z = z; return Quad(g, -1, 1); }
double sphereVol = Quad(f, -1, 1), pi = sphereVol*3/4;

23 lines

```
template<typename Func>
double simpson(Func f, double a, double b) {
  double c = (a + b) / 2;
  return (f(a) + 4 * f(c) + f(b)) * (b - a) / 6;
}

template<typename Func>
double recurse(Func f, double a, double b,
               double eps, double S) {
  double c = (a + b) / 2;
  double S1 = simpson(f, a, c);
  double S2 = simpson(f, c, b);
  double T = S1 + S2;
  if (abs(T - S) < 15 * eps || b - a < 1e-10)
    return T + (T - S) / 15;
  return recurse(f, a, c, eps / 2, S1) +
         recurse(f, c, b, eps / 2, S2);
}

template<typename Func>
double Quad(Func f, double a, double b, double eps = 1e-8) {
  return recurse(f, a, b, eps, simpson(f, a, b));
}
```

Simplex.h

Description: Solves a general linear maximization problem: maximize $c^T x$ subject to $Ax \leq b$, $x \geq 0$. 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.
Usage: A = {{1,-1}, {-1,1}, {-1,-2}};
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.

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
#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define sz(x) (int)(x).size()

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;
  }
};
```

SolveLinear.h

Description: Solves $M * x = b$. If there are multiple solutions, returns a solution which has all free variables set to 0. To compute rank, count the number of values in pivot. vector which are not -1. For inverse modulo 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: $\mathcal{O}\left(vars^2 cons\right)$ 79 lines

```
// Transforms a matrix into its row echelon form
// Returns a vector of pivots for each variable
// vars is the number of variables to do echelon for
vector<int> ToRowEchelon(vector<vector<double>> &M, int vars) {
  int n = M.size(), m = M[0].size();
  vector<int> pivots(vars, -1);

  int cur = 0;
  for (int var = 0; var < vars; ++var) {
    if (cur >= n) break;

    for (int con = cur + 1; con < n; ++con)
      if (sgn(M[con][var]) != 0)
        swap(M[con], M[cur]);

    if (sgn(M[cur][var]) != 0) {
      pivots[var] = cur;
      auto aux = M[cur][var];

      for (int i = 0; i < m; ++i)
        M[cur][i] = M[cur][i] / aux;

      for (int con = 0; con < n; ++con) {
        if (con != cur) {
          auto mul = M[con][var];
          for (int i = 0; i < m; ++i) {
            M[con][i] = M[con][i] - mul * M[cur][i];
          }
        }
      }
      ++cur;
    }
  }

  return pivots;
}

// Computes the inverse of a nxn square matrix.
// Returns true if successful
bool Invert(vector<vector<double>> &M) {
  int n = M.size();
  for (int i = 0; i < n; ++i) {
    M[i].resize(2 * n, 0); M[i][n + i] = 1;
  }

  auto pivs = ToRowEchelon(M, n);
  for (auto x : pivs) if (x == -1) return false;

  for (int i = 0; i < n; ++i)
    M[i].erase(M[i].begin(), M[i].begin() + n);
  return true;
}

// Returns the solution of a system
// Will change matrix
// Throws 5 if inconsistent
vector<double> SolveSystem(vector<vector<double>> &M,
                           vector<double>& b) {
  int vars = M[0].size();
  for (int i = 0; i < (int)M.size(); ++i)
    M[i].push_back(b[i]);

  auto pivs = ToRowEchelon(M, vars);
  vector<double> solution(vars);
  for (int i = 0; i < vars; ++i) {
    solution[i] = (pivs[i] == -1) ? 0 : M[pivs[i]][vars];
  }

  // Check feasible (optional)
  for (int i = 0; i < (int)M.size(); ++i) {
    double check = 0;
    for (int j = 0; j < vars; ++j)
      check = check + M[i][j] * solution[j];
    if (sgn(check - M[i][vars]) != 0)
      throw 5;
  }

  return solution;
}
```

Tridiagonal.h

Description: Solves a linear equation system with a tridiagonal matrix with diagonal diag, subdiagonal sub and superdiagonal super, i.e., $x = \text{Tridiagonal}(d, p, q, b)$ solves the equation system

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

The size of diag and b should be the same and super and sub should be one element shorter. T is intended to be double.
This is useful for solving problems on the type

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

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

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

Usage: `int n = 1000000;`
```
vector<double> diag(n,-1), sup(n-1,.5), sub(n-1,.5), b(n,1);
vector<double> x = tridiagonal(diag, super, sub, b);
```
Time: $\mathcal{O}(N)$ 14 lines

```
template <typename T>
vector<T> Tridiagonal(vector<T> diag, const vector<T>& super,
    const vector<T>& sub, vector<T> b) {
  for (int i = 0; i < b.size() - 1; ++i) {
    diag[i + 1] -= super[i] * sub[i] / diag[i];
    b[i + 1] -= b[i] * sub[i] / diag[i];
  }
  for (int i = b.size(); --i > 0;) {
    b[i] /= diag[i];
    b[i - 1] -= b[i] * super[i - 1];
  }
  b[0] /= diag[0];
  return b;
}
```

Number theory (5)

5.1 Modular arithmetic

ModOps.h

Description: ModOps class and operations for easy modulo reduction. Quick to code, but not fast. 19 lines

```
const int kMod = 1e9 + 7;

struct ModInt {
  long long n;

  ModInt(long long n = 0) : n(n % kMod) {}
  ModInt operator+(const ModInt& oth) { return n + oth.n; }
  ModInt operator-(const ModInt& oth) { return n - oth.n; }
  ModInt operator*(const ModInt& oth) { return n * oth.n; }
  long long get() { return n < 0 ? n + kMod : n; }
};

ModInt lgpow(ModInt b, int e) {
  ModInt r;
  for (r = 1; e; e /= 2, b = b * b)
    if (e % 2) r = r * b;
  return r;
}
ModInt inv(ModInt x) { return lgpow(x, kMod - 2); }
```

ModInverse.h

Description: Pre-computation of modular inverses. Assumes lim < kMod and that kMod is a prime.
"ModOps.h" 7 lines

```
vector<ModInt> ComputeInverses(int lim) {
  vector<ModInt> inv(lim + 1); inv[1] = 1;
  for (int i = 2; i <= lim; ++i) {
    inv[i] = ModInt(0) - ModInt(kMod / i) * inv[kMod % i];
  }
  return inv;
}
```

ModSum.h

Description: Sums of mod'ed arithmetic progressions.
`modsum(to, c, k, m)` = $\sum_{i=0}^{to-1}(ki + c)\%m$. `divsum` is similar but for floored division.
Time: $\log(m)$, with a large constant. 21 lines

```
using ull = unsigned long long;
using ll = long long;

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) {
    ull to2 = (to * k + c) / m;
    res += to * to2;
    res -= DivSum(to2, m-1 - c, m, k) + to2;
  }
  return res;
}

ll ModSum(ull to, ll c, ll k, ll m) {
  c %= m; if (c < 0) c += m;
  k %= m; if (k < 0) k += 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 large c.
Time: $\mathcal{O}(64/bits \cdot \log b)$, where $bits = 64 - k$, if we want to deal with k-bit numbers.

19 lines

```
typedef unsigned long long ull;
const int bits = 10;
// if all numbers are less than 2^k, set bits = 64-k
const ull po = 1 << bits;
ull ModMul(ull a, ull b, ull &c) {
  ull x = a * (b & (po - 1)) % c;
  while ((b >>= bits) > 0) {
    a = (a << bits) % c;
    x += (a * (b & (po - 1))) % c;
  }
  return x % c;
}
ull ModPow(ull a, ull b, ull mod) {
  if (b == 0) return 1;
  ull res = ModPow(a, b / 2, mod);
  res = ModMul(res, res, mod);
  if (b & 1) return ModMul(res, a, mod);
  return res;
}
```

ModSqrt.h

Description: Tonelli-Shanks algorithm for modular square roots.
Time: $\mathcal{O}(\log^2 p)$ worst case, often $\mathcal{O}(\log p)$

"ModPow.h" 30 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);
  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;
  int r = 0;
  while (s % 2 == 0)
    ++r, s /= 2;
  ll n = 2; // find a non-square mod p
  while (modpow(n, (p - 1) / 2, p) != p - 1) ++n;
  ll x = modpow(a, (s + 1) / 2, p);
  ll b = modpow(a, s, p);
  ll g = modpow(n, s, p);
  for (;;) {
    ll t = b;
    int m = 0;
    for (; m < r; ++m) {
      if (t == 1) break;
      t = t * t % p;
    }
    if (m == 0) return x;
    ll gs = modpow(g, 1 << (r - m - 1), p);
    g = gs * gs % p;
    x = x * gs % p;
    b = b * g % p;
    r = m;
  }
}
```

5.2 Number theoretic transform

NTT.h

Description: Number theoretic transform. Can be used for convolutions modulo specific nice primes of the form $2^a b+1$, where the convolution result has size at most 2^a. For other primes/integers, use two different primes and combine with CRT. If NTT is not fast enough and you are multiplying a lot, consider doing naive solution for the small ones.
Time: $\mathcal{O}(N \log N)$

"ModPow.h" 65 lines

```
const int kMod = (119 << 23) + 1, kRoot = 3; // = 998244353
// For p < 2^30 there is also e.g. (5 << 25, 3), (7 << 26, 3),
// (479 << 21, 3) and (483 << 21, 5). The last two are > 10^9.

struct FFTSolver {
  vector<int> rev;

  int __lg(int n) { return n == 1 ? 0 : 1 + __lg(n / 2);  }

  void compute_rev(int n, int lg) {
    rev.resize(n); rev[0] = 0;
    for (int i = 1; i < n; ++i) {
      rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (lg - 1));
    }
  }

  vector<ModInt> fft(vector<ModInt> V, bool invert) {
    int n = V.size(), lg = __lg(n);
    if ((int)rev.size() != n) compute_rev(n, lg);

    for (int i = 0; i < n; ++i) {
      if (i < rev[i])
        swap(V[i], V[rev[i]]);
    }

    for (int step = 2; step <= n; step *= 2) {
      ModInt eps = lgpow(kRoot, (kMod - 1) / step);
      if (invert) eps = inv(eps);

      for (int i = 0; i < n; i += step) {
        ModInt w = 1;
        for (int a = i, b = i+step/2; b < i+step; ++a, ++b) {
          ModInt aux = w * V[b];
          V[b] = V[a] - aux;
          V[a] = V[a] + aux;
          w = w * eps;
        }
      }
    }

    return V;
  }

  vector<ModInt> Multiply(vector<ModInt> A, vector<ModInt> B) {
    int n = A.size() + B.size() - 1, sz = n;
    while (n != (n & -n)) ++n;
    A.resize(n, 0); B.resize(n, 0);

    A = fft(move(A), false);
    B = fft(move(B), false);

    vector<ModInt> ret(n);
    ModInt inv_n = inv(n);

    for (int i = 0; i < n; ++i) {
      ret[i] = A[i] * B[i] * inv_n;
    }

    ret = fft(move(ret), true);
    ret.resize(sz);

    return ret;
  }
};
```

5.3 Primality

Eratosthenes.h

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

15 lines

```
const int kMaxPr = 5000000;
bitset<kMaxPr> isprime;

vector<int> Sieve(int lim) {
  isprime.set(); isprime[0] = isprime[1] = 0;
  for (int i = 4; i < lim; i += 2) isprime[i] = 0;
  for (int i = 3; i*i < lim; i += 2) if (isprime[i])
    for (int j = i*i; j < lim; j += i*2) isprime[j] = 0;

  vector<int> pr;
  for (int i = 2; i < lim; ++i)
    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 $a^b \mod c$.

"ModMulLL.h" 18 lines

```
using ull = unsigned long long;

bool IsPrime(ull p) {
  if (p == 2) return true;
  if (p == 1 || p % 2 == 0) return false;
  ull s = p - 1;
  while (s % 2 == 0) s /= 2;
  for (int i = 0; i < 15; ++i) {
    ull a = rand() % (p - 1) + 1, tmp = s;
    ull mod = ModPow(a, tmp, p);
    while (tmp != p - 1 && mod != 1 && mod != p - 1) {
      mod = ModMul(mod, mod, p);
      tmp *= 2;
    }
    if (mod != p - 1 && tmp % 2 == 0) return false;
  }
  return true;
}
```

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(bits)`, where bits is the length of the numbers you use.
Time: Expected running time should be good enough for 50-bit numbers.

"MullerRabin.h", "Eratosthenes.h", "Euclid.h" 39 lines

```
using ull = unsigned long long;

vector<ull> pr;

ull f(ull a, ull n, ull &has) {
  return (ModMul(a, a, n) + has) % n;
}

vector<ull> Factorize(ull d) {
  vector<ull> res;
  for (size_t i = 0; i < pr.size() && pr[i]*pr[i] <= d; i++)
    if (d % pr[i] == 0) {
      while (d % pr[i] == 0) d /= pr[i];
      res.push_back(pr[i]);
    }
  //d is now a product of at most 2 primes.
  if (d > 1) {
    if (prime(d))
      res.push_back(d);
    else while (true) {
      ull has = rand() % 2321 + 47;
      ull x = 2, y = 2, c = 1;
      for (; c==1; c = gcd((y > x ? y - x : x - y), d)) {
        x = f(x, d, has);
        y = f(f(y, d, has), d, has);
      }
      if (c != d) {
        res.push_back(c); d /= c;
        if (d != c) res.push_back(d);
        break;
      }
    }
  }
  return res;
}

void Init(int bits) {//how many bits do we use?
  pr = Sieve(1 << ((bits + 2) / 3));
}
```

5.4 Divisibility

Euclid.h
Description: Finds the Greatest Common Divisor to the integers a and b. Euclid also finds two integers x and y, such that $ax + by = \gcd(a, b)$. If a and b are coprime, then x is the inverse of $a \pmod b$.

14 lines

```
using ll = long long;

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

ll ModInv(ll a, ll p) {
  ll x, y;
  assert(Euclid(a, p, x, y) == 1);
  return x;
}
```

5.4.1 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

$$\left(x + \frac{kb}{\gcd(a,b)}, y - \frac{ka}{\gcd(a,b)}\right), \quad k \in \mathbb{Z}$$

phiFunction.h
Description: *Euler's totient* or *Euler's phi* function is defined as $\phi(n) :=$ # of positive integers $\leq n$ that are coprime with n. The *cototient* is $n - \phi(n)$. $\phi(1) = 1$, 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 $\Rightarrow a^{p-1} \equiv 1 \pmod p$ $\forall a$.

11 lines

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

void ComputePhi() {
  for (int i = 0; i < kLim; ++i)
    phi[i] = (i % 2) ? i : i / 2;
  for (int i = 3; i < kLim; i += 2)
    if (phi[i] == i)
      for (int j = i; j < kLim; j += i)
        (phi[j] /= i) *= i - 1;
}
```

5.5 Chinese remainder theorem

CRT.h
Description: Chinese Remainder Theorem.
Find z such that $z\%m_1 = r_1, z\%m_2 = r_2$. Here, z is unique modulo M = lcm(m1, m2). The vector version solves a system of equations of type $z\%m_i = p_i$. On output, return { 0, -1 } . Note that all numbers must be less than 2^{31} if you have type unsigned long long.
Time: $\log(m + n)$

"Euclid.h" 17 lines

```
pair<int, int> CRT(int m1, int r1, int m2, int r2) {
  int s, t;
  int g = Euclid(m1, m2, s, t);
  if (r1 % g != r2 % g) return make_pair(0, -1);
  int z = (s * r2 * m1 + t * r1 * m2) % (m1 * m2);
  if (z < 0) z += m1 * m2;
  return make_pair(m1 * m2 / g, z / g);
}

pair<int, int> CRT(vector<int> m, vector<int> r) {
  pair<int, int> ret = make_pair(m[0], r[0]);
  for (int i = 1; i < m.size(); i++) {
    ret = CRT(ret.first, ret.second, m[i], r[i]);
    if (ret.second == -1) break;
  }
  return ret;
}
```

5.6 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 $m > n > 0$, $k > 0$, $m \perp n$, and either m or n even.

5.7 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 exist modulo any prime power p^a, except for $p = 2, a > 2$, and there are $\phi(\phi(p^a))$ many. For $p = 2, a > 2$, the group $\mathbb{Z}_{2^a}^{\times}$ is instead isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}_{2^{a-2}}$.

5.8 Estimates

$\sum_{d|n} d = O(n \log \log n)$.

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 $n < 1e19$.

Combinatorial (6)

6.1 The Twelvefold Way

Counts the # of functions $f : N \to K$, $|N| = n$, $|K| = k$. The elements in N and K can be distinguishable or indistinguishable, while f can be injective (one-to-one) of surjective (onto).

N	K	none	injective	surjective
dist	dist	k^n	$\frac{k!}{(k-n)!}$	$k!S(n,k)$
indist	dist	$\binom{n+k-1}{n}$	$\binom{k}{n}$	$\binom{n-1}{n-k}$
dist	indist	$\sum_{t=0}^{k} S(n,t)$	$[n \leq k]$	$S(n,k)$
indist	indist	$\sum_{t=1}^{k} p(n,t)$	$[n \leq k]$	$p(n,k)$

Here, $S(n,k)$ is the Stirling number of the second kind, and $p(n,k)$ is the partition number.

6.2 Permutations

6.2.1 Factorial

n	1 2 3	4	5	6	7	8	9	10
$n!$	1 2 6	24	120	720	5040	40320	362880	3628800
n	11	12	13	14	15	16	17	
$n!$	4.0e7	4.8e8	6.2e9	8.7e10	1.3e12	2.1e13	3.6e14	
n	20	25	30	40	50	100	150	171
$n!$	2e18	2e25	3e32	8e47	3e64	9e157	6e262	>DBL_MAX

intperm.h
Description: Permutations to/from integers. The bijection is order preserving.
Time: $\mathcal{O}(n^2)$ 20 lines

```
int factorial[] = {1, 1, 2, 6, 24, 120, 720, 5040}; // etc.
template <class Z, class It>
void perm_to_int(Z& val, It begin, It end) {
  int x = 0, n = 0;
  for (It i = begin; i != end; ++i, ++n)
    if (*i < *begin) ++x;
  if (n > 2) perm_to_int<Z>(val, ++begin, end);
  else val = 0;
  val += factorial[n-1]*x;
}
/* range [begin, end) does not have to be sorted. */
template <class Z, class It>
void int_to_perm(Z val, It begin, It end) {
  Z fac = factorial[end - begin - 1];
  // Note that the division result will fit in an integer!
  int x = val / fac;
  nth_element(begin, begin + x, end);
  swap(*begin, *(begin + x));
  if (end - begin > 2) int_to_perm(val % fac, ++begin, end);
}
```

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.

$$D(n) = (n-1)(D(n-1)+D(n-2)) = nD(n-1)+(-1)^n = \left\lfloor\frac{n!}{e}\right\rceil$$

derangements.h
Description: Generates the i:th derangement of S_n (in lexicographical order). 38 lines

```
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+1)
      ans += (i&1?-1:1) * choose[k][i] * fac[n-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 - i;
      rep(j,0,m) if (vals[j] > i) ++k;
      rep(j,0,m) {
        T p = 0;
        if (vals[j] > i) p = DGen(m-1, k-1);
        else if (vals[j] < i) p = DGen(m-1, k);
        if (idx <= p) break;
        idx -= p;
      }
      res[i] = vals[j];
      memmove(vals + j, vals + j + 1, sizeof(int)*(m-j-1));
    }
  }
};
```

6.2.4 Involutions

An involution is a permutation with maximum cycle length 2, and it is its own inverse.

$$a(n) = a(n-1) + (n-1)a(n-2)$$

$$a(0) = a(1) = 1$$

1, 1, 2, 4, 10, 26, 76, 232, 764, 2620, 9496, 35696, 140152

6.2.5 Stirling numbers of the first kind

$$s(n,k) = (-1)^{n-k}c(n,k)$$

$c(n,k)$ is the unsigned Stirling numbers of the first kind, and they count the number of permutations on n items with k cycles.

$$s(n,k) = s(n-1,k-1) - (n-1)s(n-1,k)$$

$$s(0,0) = 1, s(n,0) = s(0,n) = 0$$

$$c(n,k) = c(n-1,k-1) + (n-1)c(n-1,k)$$

$$c(0,0) = 1, c(n,0) = c(0,n) = 0$$

6.2.6 Eulerian numbers

Number of permutations $\pi \in S_n$ in which exactly k elements are greater than the previous element. k j:s s.t. $\pi(j) > \pi(j+1)$, $k+1$ j:s s.t. $\pi(j) \geq j$, k j:s 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 $G = \mathbb{Z}_n$ to get

$$g(n) = \frac{1}{n}\sum_{k=0}^{n-1} f(\gcd(n,k)) = \frac{1}{n}\sum_{k|n} f(k)\phi(n/k).$$

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

$$p(0) = 1,\ p(n) = \sum_{k\in\mathbb{Z}\setminus\{0\}}(-1)^{k+1}p(n-k(3k-1)/2)$$

$$p(n) \sim 0.145/n \cdot \exp(2.56\sqrt{n})$$

n	0	1	2	3	4	5	6	7	8	9	20	50	100
$p(n)$	1	1	2	3	5	7	11	15	22	30	627	~2e5	~2e8

6.3.2 Binomials

binomial.h
Description: The number of k-element subsets of an n-element set, $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
Time: $\mathcal{O}(\min(k, n-k))$ 6 lines

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

binomialModPrime.h
Description: Lucas' thm: Let n, m be non-negative integers and p a prime. Write $n = n_k p^k + ... + n_1 p + n_0$ and $m = m_k p^k + ... + m_1 p + m_0$. Then $\binom{n}{m} \equiv \prod_{i=0}^{k} \binom{n_i}{m_i} \pmod p$. fact and invfact must hold pre-computed factorials / inverse factorials, e.g. from ModInverse.h.
Time: $\mathcal{O}(\log_p n)$ 10 lines

```
ll chooseModP(ll n, ll m, int p, vi& fact, vi& 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

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

multinomial.h
Description: $\binom{\sum k_i}{k_1, k_2, \ldots, k_n} = \frac{(\sum k_i)!}{k_1! k_2! \ldots k_n!}$
Time: $\mathcal{O}((\sum k_i) - k_1)$ 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;
}
```

6.3.3 Stirling numbers of the second kind

Partitions of n distinct elements into exactly 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, 877, 4140, 21147, 115975, 678570, 4213597. For a prime p

$$B(p^m + n) \equiv mB(n) + B(n+1) \pmod p$$

6.3.5 Triangles

Given rods of length $1, \ldots, n$,

$$T(n) = \frac{1}{24}\begin{cases} n(n-2)(2n-5) & n \text{ even} \\ (n-1)(n-3)(2n-1) & n \text{ odd} \end{cases}$$

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

6.4 General purpose numbers

6.4.1 Catalan numbers

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

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

$$C_0 = 1, C_{n+1} = \sum C_i C_{n-i}$$

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 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, E, SE.

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

$$M(0) = M(1) = 1$$

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 and SE, and with k peaks.

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

$$N(n,1) = N(n,n) = 1$$

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

1, 1, 1, 1, 3, 1, 1, 6, 6, 1, 1, 10, 20, 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. 1, 2, 6, 22, 90, 394, 1806, 8558, 41586, 206098

Graph (7)

7.1 Euler walk

EulerWalk.h

Description: Eulerian undirected/directed path/cycle algorithm. For a directed / undirected graph. For each path/cycle found, calls a callback. You can check cycle by checking path endpoints. To transform into undirected, toggle comment on lines (*)
Time: $\mathcal{O}(E)$ where E is the number of edges. 46 lines

```
struct EulerWalk {
  int n;
  vector<multiset<int>> G;
  vector<int> deg;

  EulerWalk(int n) : n(n), G(n + 1), deg(n + 1, 0) {}

  void AddEdge(int a, int b) {
    G[b].insert(a);
    deg[a] += 1; deg[b] -= 1;
    // G[a].insert(b); (*)
  }

  vector<int> walk;
  void dfs(int node) {
    while (G[node].size()) {
      auto vec = *G[node].begin();
      G[node].erase(G[node].begin());
      // G[vec].erase(G[vec].find(node)); (*)
      dfs(vec);
    }
    walk.push_back(node);
  }

  template<typename Callback>
  void Solve(Callback cb) {
    for (int i = 1; i <= n; ++i) {
      while (deg[i] < 0) AddEdge(i, n); // (*)
      while (deg[i] > 0) AddEdge(n, i); // (*)
      // if (deg[i] % 2) AddEdge(i, n); (*)
    }
    // Paths
    vector<int> buff; dfs(n);
    for (auto node : walk) {
      if (node < n) buff.push_back(node);
      else if (buff.size()) {
        cb(buff); buff.clear();
      }
    }
    // Cycles
    for (int i = 0; i < n; ++i) {
      walk.clear(); dfs(i);
      if (walk.size() > 1) cb(walk);
    }
  }
};
```

7.2 Network flow

DinicFlow.h

Description: Quick flow algorithm.
Time: $\mathcal{O}(V^2 * E)$ or $\mathcal{O}(E * sqrt(E))$ on unit graphs 67 lines

```
struct Dinic {
  struct Edge { int to, cap, flow, nxt; };

  vector<Edge> edges;
  vector<int> graph, at, dist;
  int src = 0, dest = 1;

  Dinic(int n) : graph(n, -1), dist(n, -1) {}

  void add_edge(int from, int to, int cap) {
    edges.push_back(Edge {to, cap, 0, graph[from]});
    graph[from] = edges.size() - 1;
  }
  void AddEdge(int from, int to, int cap) {
    add_edge(from, to, cap);
    add_edge(to, from, 0);
  }

  bool bfs() {
    queue<int> q;
    fill(dist.begin(), dist.end(), -1);
    dist[src] = 0; q.push(src);

    while (!q.empty()) {
      int node = q.front(); q.pop();
      for (int i = graph[node]; i >= 0; i = edges[i].nxt) {
        const auto &e = edges[i];
        if (dist[e.to] == -1 && e.flow < e.cap) {
          dist[e.to] = dist[node] + 1;
          q.push(e.to);
        }
      }
    }

    return dist[dest] != -1;
  }

  int dfs(int node, int flow) {
    if (flow == 0) return 0;
    if (node == dest) return flow;

    while (at[node] != -1) {
      int eid = at[node]; const auto &e = edges[eid];

      if (dist[e.to] == dist[node] + 1) {
        if (int ret = dfs(e.to, min(flow, e.cap - e.flow))) {
          edges[ eid ].flow += ret;
          edges[eid^1].flow -= ret;
          return ret;
        }
      }

      at[node] = e.nxt;
    }
    return 0;
  }

  int Compute(int src, int dest) {
    this->src = src; this->dest = dest; int ret = 0;
    while (bfs()) {
      at = graph;
      while (int flow = dfs(src, 2e9))
        ret += flow;
    }
    return ret;
  }
};
```

EZFlow.h

Description: A slow, albeit very easy-to-implement flow algorithm.
Time: $\mathcal{O}(EF)$ where E is the number of edges and F is the maximum flow. 29 lines

```
struct EZFlow {
  vector<vector<int>> G;
  vector<bool> vis;
  int t;

  EZFlow(int n) : G(n), vis(n) {}

  bool dfs(int node) {
    if (node == t) return true;
    vis[node] = true;

    for (auto& vec : G[node]) {
      if (!vis[vec] && dfs(vec)) {
        G[vec].push_back(node);
        swap(vec, G[node].back());
        G[node].pop_back();
        return true;
      }
    }
    return false;
  }

  void AddEdge(int a, int b) { G[a].push_back(b); }
  int ComputeFlow(int s, int t) {
    this->t = t; int ans = 0;
    while (dfs(s)) {++ans; fill(vis.begin(),vis.end(), false);}
    return ans;
  }
};
```

MinCostMaxFlow.h

Description: Min-cost max-flow with potentials technique. If costs can be negative, call SetPi before Compute, but note that negative cost cycles are not allowed (that's NP-hard). To obtain the actual flow, look at positive values only.
Time: Approximately $\mathcal{O}(E^2)$. Another upper bound is $\mathcal{O}(FE \log E)$
<bits/stdc++.h>, <bits/extc++.h> 90 lines

```
using T = int;
const T kInf = numeric_limits<T>::max() / 4;

struct MFMC {
  struct Edge { int to, nxt; T flow, cap, cost; };
  vector<Edge> edges;

  int n;
  vector<T> dist, pi;
  vector<int> par, graph;

  MFMC(int n) :
```

```
  n(n), dist(n), pi(n, 0), par(n), graph(n, -1) {}

  void _addEdge(int from, int to, T cap, T cost) {
    edges.push_back(Edge{to, graph[from], 0, cap, cost});
    graph[from] = edges.size() - 1;
  }
  void AddEdge(int from, int to, T cap, T cost) {
    _addEdge(from, to, cap, cost);
    _addEdge(to, from, 0, -cost);
  }

  bool dijkstra(int s, int t) {
    fill(dist.begin(), dist.end(), kInf);
    fill(par.begin(), par.end(), -1);

    __gnu_pbds::priority_queue<pair<T, int>> q;
    vector<decltype(q)::point_iterator> its(n);

    dist[s] = 0; q.push({0, s});
    while (!q.empty()) {
      int node; T d;
      tie(d, node) = q.top(); q.pop();
      if (dist[node] != -d) continue;
      for (int i = graph[node]; i >= 0; ) {
        const auto &e = edges[i];
        T now = dist[node] + pi[node] - pi[e.to] + e.cost;
        if (e.flow < e.cap && now < dist[e.to]) {
          dist[e.to] = now;
          par[e.to] = i;
          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});
        }
        i = e.nxt;
      }

    }
    for (int i = 0; i < n; ++i)
    pi[i] = min(pi[i] + dist[i], kInf);
    return par[t] != -1;
  }

  pair<T, T> Compute(int s, int t) {
    T flow = 0, cost = 0;
    while (dijkstra(s, t)) {
      T now = kInf;
      for (int node = t; node != s; ) {
        int ei = par[node];
        now = min(now, edges[ei].cap - edges[ei].flow);
        node = edges[ei ^ 1].to;
      }
      for (int node = t; node != s; ) {
        int ei = par[node];
        edges[ei].flow += now;
        edges[ei ^ 1].flow -= now;
        cost += edges[ei].cost * now;
        node = edges[ei ^ 1].to;
      }
      flow += now;
    }
    return {flow, cost};
  }

  // If some costs can be negative, call this before maxflow:
  void SetPi(int s) { // (otherwise, leave this out)
    fill(pi.begin(), pi.end(), kInf); pi[s] = 0;
    int it = n, ch = 1; T v;
    while (ch-- && it--)
      for (int i = 0; i < n; ++i) if (pi[i] != kInf)
        for (int ei = graph[i]; ei >= 0; ) {
          const auto& e = edges[ei];
          if (e.cap && (v = pi[i] + e.cost) < pi[e.to])
            pi[e.to] = v, ch = 1;
          ei = e.nxt;
        }
    assert(it >= 0); // negative cost cycle
  }
};
```

CycleCancelFlow.h

Description: Cycle-cancelling algorithm for minimum cost circulation or minimum cost flow (uncomment lines for flow and remove for in iterate) 71 lines

```
struct CCFlow {
  const int kInf = 1e9;
  struct Edge {
    int to, f, c, k;
    int res() { return c - f; }
  };
  vector<Edge> es;
  vector<vector<int>> G;
  long long cost;

  CCFlow(int n) : G(n), in(n), dist(n) {}

  void add_edge(int a, int b, int c, int k) {
    G[a].push_back(es.size());
    es.push_back(Edge{b, 0, c, k});
  }
  void AddEdge(int a, int b, int c, int k) {
    add_edge(a, b, c, k);
    add_edge(b, a, 0, -k);
  }

  int start, aug;
  vector<int> in;
  vector<long long> dist;

  int dfs(int node, int f) {
    if (in[node]) {
      start = node;
      aug = f;
      return 1;
    }

    in[node] = true;
    for (auto ei : G[node]) {
      auto& e = es[ei];
      if (dist[e.to] <= dist[node] + e.k or e.res() == 0)
        continue;

      dist[e.to] = dist[node] + e.k;

      int rec = dfs(e.to, min(f, e.res()));
      if (rec == 2) return 2;
      if (rec == 1) {
        es[ei].f += aug; es[ei ^ 1].f -= aug;
        cost += 1LL * aug * es[ei].k;
        return 1 + (node == start);
      }
    }
    in[node] = false;
    return 0;
  }

  bool iterate() {
    bool ok = false;
    for (int s = 0; s < n; ++s) {
      fill(in.begin(), in.end(), 0);
      fill(dist.begin(), dist.end(), 1LL * kInf * kInf);
      dist[s] = 0;
      ok |= dfs(s, kInf);
    }
    return ok;
  }
  long long Solve() {
    // AddEdge(t, s, kInf, -kInf);
    while (iterate());
    // int flow = es[G[t].back()].f;
    // cost += 1LL * flow * kInf;
    return cost;
  }

};
```

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. 1 lines

GlobalMinCut.h

Description: Find a global minimum cut in an undirected graph, as represented by an adjacency matrix.
Time: $\mathcal{O}(V^3)$
<bits/stdc++.h> 40 lines

```
using T = long long;
pair<T, vector<int>> GetMinCut(vector<vector<T>> weights) {
  int n = weights.size();
  vector<int> used(n), best_cut, cut;
  T best_weight = numeric_limits<T>::max();

  for (int phase = n - 1; phase > 0; phase--) {
    auto w = weights[0];
    auto added = used;
    int prev, k = 0;

    for (int i = 0; i < phase; ++i) {
      prev = k; k = -1;

      for (int j = 1; j < n; ++j)
        if (!added[j] && (k == -1 || w[j] > w[k]))
          k = j;

      if (i != phase - 1) {
        for (int j = 0; j < n; ++j)
          w[j] += weights[k][j];
        added[k] = true;
        continue;
      }

      for (int j = 0; j < n; ++j)
          weights[prev][j] += weights[k][j];
      for (int j = 0; j < n; ++j)
          weights[j][prev] = weights[prev][j];

      used[k] = true; cut.push_back(k);

      if (w[k] < best_weight) {
        best_cut = cut;
        best_weight = w[k];
      }
    }
  }
  return {best_weight, best_cut};
}
```

GomoryHu.h

Description: Computes Gomory-Hu tree of a graph
Time: $\mathcal{O}(V)$ calls of flow algorithm

14 lines

```
void GomoryHu() {
    int parent[n]; //initialized to 0
    int answer[n][n]; //initialize this one to infinity
    for(int i=1;i<n;++i){
        //Compute the minimum cut between i and parent[i].
        //Let the i-side of the min cut be S, and the value of
             the min-cut be F
        for (int j=i+1;j<n;++j)
            if ((j is in S) && parent[j]==parent[i])
                parent[j]=i;
        answer[i][parent[i]]=answer[parent[i]][i]=F;
        for (int j=0;j<i;++j)
            answer[i][j]=answer[j][i]=min(F,answer[parent[i]][j
                ]);
    }
}
```

7.3 Matching

DFSMatching.h

Description: This is a simple matching algorithm but should be just fine in most cases. n is the size of the left partition and m is the size of the right partition. If you want to get the matched pairs, $match[i]$ contains match for vertex i on the right side or -1 if it's not matched. Performance can be significantly improved by splitting the condition at (*) and/or by randomizing the order at (**).
Time: $\mathcal{O}(EV)$ where E is the number of edges and V is the number of vertices.

<bits/stdc++.h> 55 lines

```
struct BipartiteMatcher {
  int n, m;
  vector<vector<int>> G;
  vector<int> L, R, vis;

  BipartiteMatcher(int n, int m) :
    n(n), m(m), G(n), L(n, -1), R(m, -1), vis(n) {}

  void AddEdge(int a, int b) {
    G[a].push_back(b);
  }

  bool match(int node) {
    if (vis[node])
      return false;
    vis[node] = true;

    for (auto vec : G[node]) {
      if (R[vec] == -1 || match(R[vec])) { // (*)
        L[node] = vec; R[vec] = node; return true;
      }
    }
    return false;
  }

  int Solve() {
    int ok = 1;
    while (ok--) {
      fill(vis.begin(), vis.end(), 0);
      for (int i = 0; i < n; ++i) // (**)
        if (L[i] == -1)
          ok |= match(i);
    }
    return n - count(L.begin(), L.end(), -1);
  }

  // Only include if you want vertex cover
  vector<bool> CL, CR;

  void cover(int node) {
    for (auto vec : G[node]) if (!CR[vec]) {
      CR[vec] = true;
      CL[R[vec]] = false;
      cover(R[vec]);
    }
  }

  int VertexCover() {
    int ret = Solve();
    CL.assign(n, false); CR.assign(m, false);
    for (int i = 0; i < n; ++i) if (L[i] != -1) CL[i] = true;
    for (int i = 0; i < n; ++i) if (L[i] == -1) cover(i);
    return ret;
  }
};
```

WeightedMatching.h

Description: Min cost perfect bipartite matching. Negate costs for max cost.
Time: $\mathcal{O}(N^3)$

57 lines

```
template<typename T>
int MinAssignment(const vector<vector<T>> &c) {
  int n = c.size(), m = c[0].size();          // assert(n <= m);
  vector<T> v(m), dist(m);                    // v: potential
  vector<int> L(n, -1), R(m, -1);             // matching pairs
  vector<int> index(m), prev(m);
  iota(index.begin(), index.end(), 0);

  auto residue = [&](int i, int j) { return c[i][j] - v[j]; };
  for (int f = 0; f < n; ++f) {
    for (int j = 0; j < m; ++j) {
      dist[j] = residue(f, j); prev[j] = f;
    }
    T w; int j, l;
    for (int s = 0, t = 0;;) {
      if (s == t) {
        l = s; w = dist[index[t++]];
        for (int k = t; k < m; ++k) {
          j = index[k]; T h = dist[j];
          if (h <= w) {
            if (h < w) { t = s; w = h; }
            index[k] = index[t]; index[t++] = j;
          }
        }
        for (int k = s; k < t; ++k) {
          j = index[k];
          if (R[j] < 0) goto aug;
        }
      }
      int q = index[s++], i = R[q];
      for (int k = t; k < m; ++k) {
        j = index[k];
        T h = residue(i,j) - residue(i,q) + w;
        if (h < dist[j]) {
          dist[j] = h; prev[j] = i;
          if (h == w) {
            if (R[j] < 0) goto aug;
            index[k] = index[t]; index[t++] = j;
          }
        }
      }
    }
  aug:
    for(int k = 0; k < l; ++k)
      v[index[k]] += dist[index[k]] - w;
    int i;
    do {
      R[j] = i = prev[j];
      swap(j, L[i]);
    } while (i != f);
  }
  T ret = 0;
  for (int i = 0; i < n; ++i) {
    ret += c[i][L[i]]; // (i, L[i]) is a solution
  }
  return ret;
}
```

GeneralMatching.h

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

"../numerical/MatrixInverse-mod.h" 40 lines

```
vector<pii> generalMatching(int N, vector<pii>& ed) {
  vector<vector<ll>> mat(N, vector<ll>(N)), A;
  trav(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: $\mathcal{O}(E+V)$ 30 lines

```
vector<int> val, comp, stk, cont;
int timer, ncomps;

template<class Graph, class Func>
int dfs(int node, Graph& G, Func f) {
  int low = val[node] = ++timer, x; stk.push_back(node);
  for (auto vec : G[node]) if (comp[vec] < 0)
    low = min(low, val[vec] ?: dfs(vec, G, f));

  if (low == val[node]) {
    do {
      x = stk.back(); stk.pop_back();
      comp[x] = ncomps;
      cont.push_back(x);
    } while (x != node);
    f(cont); cont.clear();
    ncomps++;
  }
  return val[node] = low;
}

template<class Graph, class Func>
void SCC(Graph& G, Func f) {
  int n = G.size();
  val.assign(n, 0); comp.assign(n, -1);
  timer = ncomps = 0;
  for (int i = 0; i < n; ++i)
    if (comp[i] < 0)
      dfs(i, G, f);
}
```

BiconnectedComponents.h

Description: Finds all biconnected components in an undirected multigraph, 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. HOWEVER, note that we are outputting bridges as BCC's here, because we might be interested in vertex bcc's, not edge bcc's.
To get the articulation points, look for vertices that are in more than 1 BCC. To get the bridges, look for biconnected components with one edge
Time: $\mathcal{O}(E+V)$ 54 lines

```
struct BCC {
  vector<pair<int, int>> edges;
  vector<vector<int>> G;
  vector<int> enter, low, stk;

  BCC(int n) : G(n), enter(n, -1) {}

  int AddEdge(int a, int b) {
    int ret = edges.size();
    edges.emplace_back(a, b);
    G[a].push_back(ret);
    G[b].push_back(ret);
    return ret;
  }

  template<typename Iter>
  void Callback(Iter bg, Iter en) {
    for (Iter it = bg; it != en; ++it) {
      auto edge = edges[*it];
      // Do something useful
    }
  }

  void Solve() {
    for (int i = 0; i < (int)G.size(); ++i)
      if (enter[i] == -1) {
        dfs(i, -1);
      }
  }

  int timer = 0;
  int dfs(int node, int pei) {
    enter[node] = timer++;
    int ret = enter[node];

    for (auto ei : G[node]) if (ei != pei) {
      int vec = (edges[ei].first ^ edges[ei].second ^ node);
      if (enter[vec] != -1) {
        ret = min(ret, enter[vec]);
        if (enter[vec] < enter[node])
          stk.push_back(ei);
      } else {
        int sz = stk.size(), low = dfs(vec, ei);
        ret = min(ret, low);
        stk.push_back(ei);
        if (low >= enter[node]) {
          Callback(stk.begin() + sz, stk.end());
          stk.resize(sz);
        }
      }
    }
    return ret;
  }
};
```

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 $(a \lor b) \land (!a \lor c) \land (d \lor !b) \land \ldots$ becomes true, or reports that it is unsatisfiable. THROWS 5 IF NO SOLUTION Negated variables are represented by bit-inversions (~x).
Usage:
```
TwoSat sat(4); // number of variables
sat.Either(0, ~3); // Var 0 is true or var 3 is false
sat.SetValue(2); // Var 2 is true
sat.AtMostOne({0,~1,2}); // <= 1 of vars 0, ~1 and 2 are true
sat.Solve(); // Returns solution or throws
```
Time: $\mathcal{O}(N+E)$, where N is the number of boolean variables, and E is the number of clauses. 70 lines

```
struct TwoSat {
  int n;
  vector<vector<int>> G;
  vector<int> values; // 0 = false, 1 = true

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

  int AddVar() { // (optional)
    G.emplace_back();
    G.emplace_back();
    return n++;
  }

  void Implies(int a, int b) {
    a = (a >= 0 ? 2*a : -1-2*a);
    b = (b >= 0 ? 2*b : -1-2*b);
    G[a].push_back(b);
  }
  void Either(int a, int b) {
    Implies(~a, b);
    Implies(~b, a);
  }
  void SetValue(int x) {
    Either(x, x);
  }
  void AtMostOne(const vector<int>& vals) { // (optional)
    if (vals.size() <= 1) return;
    int cur = ~vals[0];
    for (int i = 2; i < (int)vals.size(); ++i) {
      int nxt = AddVar();
      Either(cur, ~vals[i]);
      Either(cur, nxt);
      Either(~vals[i], nxt);
      cur = ~nxt;
    }
    Either(cur, ~vals[1]);
  }

  vector<int> enter, comp, stk;
  int timer = 0;
  int dfs(int node) {
    int low = enter[node] = ++timer, x;
    stk.push_back(node);

    for (auto vec : G[node]) if (!comp[vec])
      low = min(low, enter[vec] ?: dfs(vec));
    ++timer;
    if (low == enter[node]) do {
      x = stk.back(); stk.pop_back();
      comp[x] = timer;
      if (values[x >> 1] == -1)
        values[x >> 1] = 1 - x & 1;
    } while (x != node);
    return enter[node] = low;
  }

  vector<int> Solve() {
    values.assign(n, -1);
    enter.assign(2 * n, 0); comp = enter;
    for (int i = 0; i < 2 * n; ++i) {
      if (!comp[i])
        dfs(i);
    }
    for (int i = 0; i < n; ++i) {
      if (comp[2 * i] == comp[2 * i + 1])
        throw 5;
    }
    return values;
  }
};
```

7.5 Trees

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: $\mathcal{O}(|V|\log|V|+Q)$
"../data-structures/RMQ.h" 43 lines

```
const pair<int, int> kInf{1 << 29, -1};

struct LCA {
  vector<int> enter, depth;
  vector<vector<int>> G;
  vector<pair<int, int>> linear;
  RMQ<pair<int, int>> rmq;
  int timer = 0;

  LCA(int n) : enter(n, -1), depth(n), G(n), linear(2 * n) {}

  void dfs(int node, int dep) {
    linear[timer] = {dep, node};
    enter[node] = timer++;
    depth[node] = dep;

    for (auto vec : G[node])
    if (enter[vec] == -1) {
      dfs(vec, dep + 1);
      linear[timer++] = {dep, node};
    }
  }

  void AddEdge(int a, int b) {
    G[a].push_back(b);
    G[b].push_back(a);
  }

  void Build(int root) {
    dfs(root, 0);
    rmq.Build(linear);
  }

  int Query(int a, int b) {
    a = enter[a], b = enter[b];
    return rmq.Query(min(a, b), max(a, b) + 1).second;
  }

  int Distance(int a, int b) {
    return depth[a] + depth[b] - 2 * depth[Query(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 the nodes of the reduced tree, while at the same time populating a link array that stores the new parents. The root points to -1.
Time: $\mathcal{O}(|S| * (\log|S| + LCA\_Q))$

"LCA.h" 18 lines

```
vector<int> CompressTree(vector<int> v, LCA& lca,
                         vector<int>& link) {
  auto cmp = [&](int a, int b) {
    return lca.enter[a] < lca.enter[b];
  };
  sort(v.begin(), v.end(), cmp);
  v.erase(unique(v.begin(), v.end()), v.end());

  for (int i = (int)v.size() - 1; i > 0; --i)
    v.push_back(lca.Query(v[i - 1], v[i]));

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

  for (int i = 0; i < (int)v.size(); ++i)
    link[v[i]] = (i == 0 ? -1 : lca.Query(v[i - 1], v[i]));
  return v;
}
```

HLD.h

Description: 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) light edges.

69 lines

```
struct HeavyLight {
  struct Node {
    int jump, subsize, depth, lin, parent;
    vector<int> leg;
  };
  vector<Node> T;
  bool processed = false;

  HeavyLight(int n) : T(n) {}

  void AddEdge(int a, int b) {
    T[a].leg.push_back(b);
    T[b].leg.push_back(a);
  }

  void Preprocess() {
    dfs_sub(0, -1); dfs_jump(0, 0);
    processed = true;
  }

  // Gets the position in the HL linearization
  int GetPosition(int node) {
    assert(processed);
    return T[node].lin;
  }

  // Gets an array of ranges of form [li...ri)
  // that correspond to the ranges you would need
  // to query in the underlying structure
  vector<pair<int, int>> GetPathRanges(int a, int b) {
    assert(processed);
    vector<pair<int, int>> ret;
    while (T[a].jump != T[b].jump) {
      if (T[T[a].jump].depth < T[T[b].jump].depth)
      swap(a, b);

      ret.emplace_back(T[T[a].jump].lin, T[a].lin + 1);
      a = T[T[a].jump].parent;
    }
    if (T[a].depth < T[b].depth) swap(a, b);
    ret.emplace_back(T[b].lin, T[a].lin + 1);
    return ret;
  }

  int dfs_sub(int x, int par) {
    auto &node = T[x];
    node.subsize = 1; node.parent = par;
    if (par != -1) {
      node.leg.erase(find(node.leg.begin(),
                          node.leg.end(), par));
      node.depth = 1 + T[par].depth;
    }
    for (auto vec : node.leg)
      node.subsize += dfs_sub(vec, x);
    return node.subsize;
  }

  int timer = 0;
  void dfs_jump(int x, int jump) {
    auto &node = T[x];
    node.jump = jump; node.lin = timer++;
    iter_swap(node.leg.begin(), max_element(node.leg.begin(),
      node.leg.end(), [&](int a, int b) {
        return T[a].subsize < T[b].subsize;
    }));
    for (auto vec : node.leg)
      dfs_jump(vec, vec == node.leg.front() ? jump : vec);
  }
};
```

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 $\mathcal{O}(\log N)$.

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

7.6 Matrix tree theorem

MatrixTree.h

Description: To count the number of spanning trees in an undirected graph G: create an $N \times N$ 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.

1 lines

Geometry (8)

8.1 Geometric primitives

Point.h

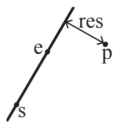
Description: Point declaration, and basic operations.

32 lines

```
using Point = complex<double>;

const double kPi = 4.0 * atan(1.0);
const double kEps = 1e-9; // Good eps for long double is ~1e-11

#define X() real()
#define Y() imag()

double dot(Point a, Point b) { return (conj(a) * b).X(); }
double cross(Point a, Point b) { return (conj(a) * b).Y(); }
double dist(Point a, Point b) { return abs(b - a); }
Point perp(Point a) { return Point{-a.Y(), a.X()}; } // +90deg
double rotateCCW(Point a, double theta) {
  return a * polar(1.0, theta); }
double det(Point a, Point b, Point c) {
  return cross(b - a, c - a); }

// abs() is norm (length) of vector
// norm() is square of abs()
// arg() is angle of vector
// det() is twice the signed area of the triangle abc
// and is > 0 iff c is to the left as viewed from a towards b.
// polar(r, theta) gets a vector from abs() and arg()

void ExampleUsage() {
  Point a{1.0, 1.0}, b{2.0, 3.0};
  cerr << a << " " << b << endl;
  cerr << "Length of ab is: " << dist(a, b) << endl;
  cerr << "Angle of a is: " << arg(a) << endl;
  cerr << "axb is: " << cross(a, b) << endl;
}
```

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, although don't rely on that. Also works in 3D. 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.

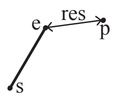


"Point.h" 8 lines

```
double LineDistance(Point a, Point b, Point p) {
  return det(a, b, p) / abs(b - a);
}

// Projects point p on line (a, b)
Point ProjectPointOnLine(Point p, Point a, Point b) {
  return a + (b - a) * dot(p - a, b - a) / norm(b - a);
}
```

SegmentDistance.h

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



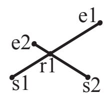
Usage: `Point a{0, 0}, b{2, 2}, p{1, 1};`
`bool onSegment = SegmentDistance(p, a, b) < kEps;`

"Point.h" 15 lines

```
double SegmentDistance(Point p, Point a, Point b) {
  if (a == b) return abs(p - a); // Beware of precision!!!
  double d = norm(b - a);
  double t = min(d, max(.0, dot(p - a, b - a)));
  return abs((p - a) * d - (b - a) * t) / d;
}

// Projects point p on segment [a, b]
Point ProjectPointOnSegment(Point p, Point a, Point b) {
  double d = norm(b - a);
  if (d == 0) return a; // Beware of precision!!!

  double r = dot(p - a, b - a) / d;
  return (r < 0) ? a : (r > 1) ? b : (a + (b - a) * r);
}
```

SegmentIntersection.h

Description:
If a unique intersection point between the line segments going from s1 to e1 and from s2 to e2 exists r1 is set to this point and 1 is returned. If no intersection point exists 0 is returned and if infinitely many exists 2 is returned and r1 and r2 are set to the two ends of the common line. The wrong position will be returned if P is Point<int> 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. Use segmentIntersectionQ to get just a true/false answer.



Usage: `Point<double> intersection, dummy;`
`if (segmentIntersection(s1,e1,s2,e2,intersection,dummy)==1)`
`cout << "segments intersect at " << intersection << endl;`

"Point.h" 27 lines

```
template <class P>
int segmentIntersection(const P& s1, const P& e1,
    const P& s2, const P& e2, P& r1, P& r2) {
  if (e1==s1) {
    if (e2==s2) {
      if (e1==e2) { r1 = e1; return 1; } //all equal
      else return 0; //different point segments
    } else return segmentIntersection(s2,e2,s1,e1,r1,r2);//swap
  }
  //segment directions and separation
  P v1 = e1-s1, v2 = e2-s2, d = s2-s1;
  auto a = v1.cross(v2), a1 = v1.cross(d), a2 = v2.cross(d);
  if (a == 0) { //if parallel
    auto b1=s1.dot(v1), c1=e1.dot(v1),
         b2=s2.dot(v1), c2=e2.dot(v1);
    if (a1 || a2 || max(b1,min(b2,c2))>min(c1,max(b2,c2)))
      return 0;
    r1 = min(b2,c2)<b1 ? s1 : (b2<c2 ? s2 : e2);
    r2 = max(b2,c2)>c1 ? e1 : (b2>c2 ? s2 : e2);
    return 2-(r1==r2);
  }
  if (a < 0) { a = -a; a1 = -a1; a2 = -a2; }
  if (0<a1 || a<-a1 || 0<a2 || a<-a2)
    return 0;
  r1 = s1-v1*a2/a;
  return 1;
}
```

LineIntersectionCheck.h

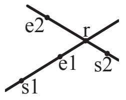
Description: Checks if two lines intersect, and returns 1 if one intersection, 0 if lines are parallel (no intersection), and -1 if they coincide (infinite intersections).

"Point.h" 6 lines

```
int LineIntersection(Point a, Point b, Point p, Point q) {
  double c1 = det(a, b, p), c2 = det(a, b, q);
  if (sgn(c1 - c2)) return 1;
  if (sgn(c1) == 0) return -1;
  return 0;
}
```

LineIntersection.h

Description:
Returns the intersection between non-parallel lines. If unsure if lines are concurrent, check with LineIntersectionCheck.
The wrong position will be returned if P is complex<int> 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.



"Point.h" 5 lines

```
Point LineIntersection(Point a, Point b, Point p, Point q) {
  double c1 = det(a, b, p), c2 = det(a, b, q);
  assert(sgn(c1 - c2)); // undefined if parallel
  return (q * c1 - p * c2) / (c1 - c2);
}
```

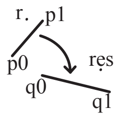
OnSegment.h

Description: Returns true iff p lies on the line segment from s to e. Intended for use with e.g. Point<long long> where overflow is an issue. Use (SegDist(s, e, p) < kEps) instead when using Point<double>.

"Point.h" 4 lines

```
bool OnSegment(Point s, Point e, Point p) {
  Point ds = p - s, de = p - e;
  return cross(ds, de) == 0 && dot(ds, de) <= 0;
}
```

LinearTransformation.h



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

"Point.h" 6 lines

```
Point LinearTransformation(Point p0, Point p1,
                           Point q0, Point q1, Point r) {
  Point dp = p1 - p0, dq = q1 - q0,
        num = dp * conj(dq);
  return q0 + (r - p0) * conj(num) / norm(dp);
}
```

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.
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`

34 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 a) const { return {x-a.x, y-a.y, t}; }
  int quad() const {
    assert(x || y);
    if (y < 0) return (x >= 0) + 2;
    if (y > 0) return (x <= 0);
    return (x <= 0) * 2;
  }
  Angle t90() const { return {-y, x, t + (quad() == 3)}; }
  Angle t180() const { return {-x, -y, t + (quad() >= 2)}; }
  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.quad(), 1LL * a.y * b.x) <
         make_tuple(b.t, b.quad(), 1LL * a.x * 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) { // where b is a vector
  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;
}
```

Caliper.h

Description: A class for simulating the rotating calipers technique. Calipers will rotate covering the smallest arc. To change that, edit the code at (*) to add 2*kPi if return value is < 0

"Point.h" 18 lines

```
struct Caliper {
  Point pivot; double angle;

  double AngleTo(Point oth) {
    double new_ang = arg(oth - pivot);
    return remainder(new_ang - angle, 2.0 * kPi); // (*)
  }

  void RotateCCW(double ang) { angle += ang; }
  void ChangePivot(Point oth) { pivot = oth; }

  // Need to have same angle
  double DistanceTo(Caliper oth) {
    Point a = RotateCCW(pivot, -angle);
    Point b = RotateCCW(oth.pivot, -angle);
    return abs(a.imag() - b.imag());
  }
};
```

8.2 Circles

Circle.h

Description: Circle

"Point.h" 1 lines

```
struct Circle { Point c; double r; };
```

CircleIntersection.h

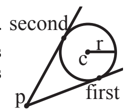
Description: Computes the intersection between two circles and other circle-related geometry

"Circle.h" 30 lines

```
// Computes the intersection of two circles.
// Can be 0(non-intersecting), 1(tangent), or 2 points
void CircleCircleIntersect(Circle c, Circle d,
                           vector<Point>& inter) {
  Point a = c.c, b = d.c, delta = b - a;
  double r1 = c.r, r2 = d.r;
  if (sgn(norm(delta)) == 0) return;
  double r = r1 + r2, d2 = norm(delta);
  double p = (d2 + r1 * r1 - r2 * r2) / (2.0 * d2);
  double h2 = r1 * r1 - p * p * d2;
  if (sgn(d2 - r * r) > 0 || sgn(h2) < 0) return;
  Point mid = a + delta * p,
        per = perp(delta) * sqrt(abs(h2) / d2);
  inter.push_back(mid - per);
  if (sgn(per) != 0)
    inter.push_back(mid + per);
}

// Computes the intersection between a line pq and a circle
// Can be 0(non-intersecting), 1(tangent), or 2 points
void LineCircleIntersect(Circle c, Point p, Point q,
                         vector<Point>& inter) {
  Point mid = ProjectPointOnLine(c.c, p, q);
  double d2 = norm(mid - c.c), dist = c.r * c.r - d2;
  if (sgn(dist) < 0) return;
  Point dir = (q - p) * sqrt(dist) / abs(q - p);
  inter.push_back(mid - dir);
  if (sgn(dist) != 0)
    inter.push_back(mid + dir);
}
```

CircleTangents.h

Description:



Returns a pair of the two points on the circle with radius r centered around c whose tangent lines intersect p. If p lies within the circle NaN-points are returned. The first point is the one to the right as seen from the p towards c.

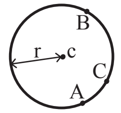
Usage: `auto p = Tangents(Point(100, 2), Point(0, 0), 2);`

"Circle.h" 6 lines

```
pair<Point, Point> Tangents(Point p, Circle c) {
  p -= c.c;
  double x = c.r * c.r / norm(p), y = sqrt(x - x * x);
  return make_pair(c.c + p * x + perp(p) * y,
                   c.c + p * x - perp(p) * y);
}
```

Circumcircle.h

Description:



The circumcirle of a triangle is the circle intersecting all three vertices. CircumRadius returns the radius of the circle going through points a, b and c and CircumCenter returns the center of the same circle.

"Circle.h" 12 lines

```
double CircumRadius(Point a, Point b, Point c) {
  return dist(a, b) * dist(b, c) * dist(c, a) /
    abs(det(a, b, c)) / 2.;
}
Point CircumCenter(Point a, Point b, Point c) {
  c -= a; b -= a;
  return a + perp(c*norm(b) - b*norm(c)) / cross(c, b) / 2.;
}
Circle CircumCircle(Point a, Point b, Point c) {
  Point p = CircumCenter(a, b, c);
  return {p, abs(p - a)};
}
```

MinimumEnclosingCircle.h

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

"Circumcircle.h" 21 lines

```
// IMPORTANT: random_shuffle(pts.begin(), pts.end())
Circle MEC(vector<Point>& pts, vector<Point> ch = {}) {
  if (pts.empty() || ch.size() == 3) {
    switch (ch.size()) {
      case 0: return {0, -1};
      case 1: return {ch[0], 0};
      case 2: return {(ch[0] + ch[1])/2, abs(ch[0] - ch[1])/2};
      case 3: return CircumCircle(ch[0], ch[1], ch[2]);
      default: assert(false);
    }
  }

  auto p = pts.back(); pts.pop_back();
  auto c = MEC(pts, ch);
  if (sgn(abs(p - c.c) - c.r) > 0) {
    ch.push_back(p);
    c = MEC(pts, ch);
  }
  pts.push_back(p);
  return c;
}
```

8.3 Polygons

InsidePolygon.h

Description: Returns true if p lies within the polygon described by the points between iterators begin and end. Returns 0 if on polygon, 1 if inside polygon and -1 if outside. 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 overflow. If points within epsilon from an edge should be considered as on the edge replace the line "if (onSegment..." with the comment bellow it (this will cause overflow for int and long long).
Usage: typedef Point<int> pi;
vector<pi> v; v.push_back(pi(4,4));
v.push_back(pi(1,2)); v.push_back(pi(2,1));
bool in = insidePolygon(v.begin(),v.end(), pi(3,4), false);
Time: $\mathcal{O}(n)$

"Point.h", "OnSegment.h" 10 lines

```
int InsidePolygon(vector<Point> P, const Point& p) {
  int ic = 0, n = P.size();
  for (int i = 0, j = n - 1; i < n; j = i++) {
    if (OnSegment(P[i], P[j], p)) return 0;
    ic += (max(P[i].Y(), P[j].Y()) > p.Y() &&
           min(P[i].Y(), P[j].Y()) <= p.Y() &&
          (det(P[i], P[j], p) > 0) == (P[i].Y() <= p.Y()));
  }
  return ic % 2 ? 1 : -1; //inside if odd number of
      intersections
}
```

InsidePolygonMulti.h

Description: Given a (possibly non-convex) polygon P and Q query points, computes if the points are inside P or not. Returns -1 for strictly outside, 0 for edge, 1 for strictly inside. If no points are on the polygon, you can remove the events of type 2 completely.
Time: $\mathcal{O}((N+Q)\log N)$

<bits/stdc++.h>, <bits/extc++.h> 57 lines

```
using namespace __gnu_pbds;

vector<int> PointsInPolygon(vector<Point> P, vector<Point> Q) {
  int n = P.size(), q = Q.size();

  // Step 1: add events to sweepline
  vector<tuple<Point, int, int, int>> events;
  auto process = [&](int i, int j) {
    events.emplace_back(P[i], 2, i, i);
    if (P[j] < P[i]) swap(i, j);
    if (P[i].real() == P[j].real()) {
      events.emplace_back(P[i], 2, i, j);
    } else {
      events.emplace_back(P[i], 1, i, j);
      events.emplace_back(P[j], 0, i, j);
    }
  };
  for (int i = 0; i < n; ++i) process(i, (i + 1) % n);
  for (int i = 0; i < q; ++i)
    events.emplace_back(Q[i], 3, i, -1);

  // Step 2: Prepare sweepline status
  sort(events.begin(), events.end());
  auto cmp = [](pair<Point, Point> p1, pair<Point, Point> p2) {
    Point a, b, p, q; tie(a, b) = p1; tie(p, q) = p2;

    int v = sgn(det(a, b, p)) + sgn(det(a, b, q));
    if (v != 0) return v > 0;
    return sgn(det(p, q, a)) + sgn(det(p, q, b)) < 0;
  };

  tree<pair<Point, Point>, null_type, decltype(cmp),
    rb_tree_tag, tree_order_statistics_node_update> s(cmp);
  vector<int> ans(q);
  Point vert{-1, -1};
  vert *= (int)(2e9);

  // Step 3: Solve
  for (auto itr : events) {
    int tp, i, j; tie(ignore, tp, i, j) = itr;

    if (tp == 0) s.erase({P[i], P[j]});
    if (tp == 1) s.insert({P[i], P[j]});
    if (tp == 2) vert = max(vert, P[j]);
    if (tp == 3) {
      auto q = Q[i];
      auto it = s.lower_bound({q, q});
      int dist = s.order_of_key({q, q});
      ans[i] = (dist % 2 ? 1 : -1);

      if ((it != s.end() && det(it->first, it->second, q) == 0)
          || (vert.real() == q.real() && vert.imag() >= q.imag
            ()))
        ans[i] = 0;
    }
  }
  return ans;
}
```

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" 6 lines

```
double SignedArea(const vector<Point> &P) {
  double area = cross(P.back(), P.front());
  for (int i = 1; i < (int)P.size(); ++i)
    area += cross(P[i - 1], P[i]);
  return area; // Divide by 2 for proper area
}
```

PolygonCenter.h

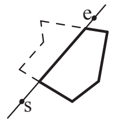
Description: Returns the center of mass for a polygon.

"Point.h" 8 lines

```
Point PolygonCenter(vector<Point>& P) {
  int n = P.size(); Point res{0, 0}; double area = 0;
  for (int i = 0, j = n - 1; i < n; j = i++) {
    res += (P[i] + P[j]) * cross(P[j], P[i]);
    area += cross(P[j], P[i]);
  }
  return res / area / 3.0;
}
```

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<Point> p = ...;
p = PolygonCut(p, Point(0, 0), Point(1, 0));

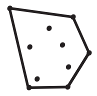


"Point.h", "LineIntersection.h" 13 lines

```
vector<Point> PolygonCut(vector<Point>& P, Point s, Point e) {
    vector<Point> res;
    for (int i = 0; i < (int)P.size(); ++i) {
        Point cur = P[i], prev = i ? P[i - 1] : P.back();
        int side1 = sgn(det(s, e, cur));
        int side2 = sgn(det(s, e, prev));
        if (side1 * side2 == -1) {
            res.push_back(LineIntersection(s, e, cur, prev));
        }
        if (side1 <= 0) res.push_back(cur);
    }
    return res;
}
```

ConvexHull.h

Description:
Returns a pair (upper_hull, lower_hull). Points on the edge of the hull between two other points are considered part of the hull. To change that, change the signs at (1) and (2) to make them non-strict.
Time: $\mathcal{O}(n\log n)$



"Point.h" 21 lines

```
using Poly = vector<Point>;
pair<Poly, Poly> ConvexHull(Poly P) {
  sort(P.begin(), P.end(), [](Point a, Point b) {
    return make_pair(a.x(), a.y()) < make_pair(b.x(), b.y());
  });
  P.erase(unique(P.begin(), P.end()), P.end());

  Poly up, dw;
  for (auto p : P) {
    while (up.size() >= 2 &&
           det(up[up.size() - 2], up.back(), p) > 0) // (1)
      up.pop_back();
    up.push_back(p);

    while (dw.size() >= 2 &&
          det(dw[dw.size() - 2], dw.back(), p) < 0) // (2)
      dw.pop_back();
    dw.push_back(p);
  }
  return tie(up, dw);
}
```

Voronoi.h

Description: Determines the voronoi cell of a point with a list of other points. If the cell is unbounded, check for points with very high coordinates.
Time: $\mathcal{O}(N^2)$

"Point.h", "PolygonCut.h", <bits/stdc++.h> 18 lines

```
const double kInf = 1e9;

// To the right of mediator is region closer to b
pair<Point, Point> Mediator(Point a, Point b) {
  Point m = (a + b) * .5;
  return make_pair(m, m + perp(b - a));
}

vector<Point> VoronoiCell(Point p, vector<Point> P) {
  vector<Point> ret = {{-kInf, -kInf}, {kInf, -kInf},
    {kInf, kInf}, {-kInf, kInf}};

  for (auto oth : P) {
    Point a, b; tie(a, b) = Mediator(p, oth);
    ret = PolygonCut(ret, b, a);
  }
  return ret;
}
```

PolygonDiameter.h

Description: Calculates the max squared distance of a set of points.

"ConvexHull.h" 19 lines

```
vector<pii> antipodal(const vector<P>& S, vi& U, vi& L) {
  vector<pii> ret;
  int i = 0, j = sz(L) - 1;
  while (i < sz(U) - 1 || j > 0) {
    ret.emplace_back(U[i], L[j]);
    if (j == 0 || (i != sz(U)-1 && (S[L[j]] - S[L[j-1]])
          .cross(S[U[i+1]] - S[U[i]]) > 0)) ++i;
    else --j;
  }
  return ret;
}

pii polygonDiameter(const vector<P>& S) {
  vi U, L; tie(U, L) = ulHull(S);
  pair<ll, pii> ans;
  trav(x, antipodal(S, U, L))
    ans = max(ans, {(S[x.first] - S[x.second]).dist2(), x});
  return ans.second;
}
```

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 -1 for points outside, 0 for points on the circumference, and 1 for points inside.
Time: $\mathcal{O}(\log N)$

"Point.h", "sideOf.h", "onSegment.h" 22 lines

```
typedef Point<ll> P;
int insideHull2(const vector<P>& H, int L, int R, const P& p) {
  int len = R - L;
  if (len == 2) {
    int sa = sideOf(H[0], H[L], p);
    int sb = sideOf(H[L], H[L+1], p);
    int sc = sideOf(H[L+1], H[0], p);
    if (sa < 0 || sb < 0 || sc < 0) return -1;
    if (sb==0 || (sa==0 && L == 1) || (sc == 0 && R == sz(H)))
      return 0;
    return 1;
  }
  int mid = L + len / 2;
  if (sideOf(H[0], H[mid], p) >= 0)
    return insideHull2(H, mid, R, p);
  return insideHull2(H, L, mid+1, p);
}

int insideHull(const vector<P>& hull, const P& p) {
  if (sz(hull) < 3) return onSegment(hull[0], hull.back(), p);
  else return insideHull2(hull, 1, sz(hull), p);
}
```

HalfplaneSet.h

Description: Data structure that dynamically keeps track of the intersection of halfplanes. Use is straigntforward. Area should be able to be kept dynamically with some modifications.
Usage: HalfplaneSet hs;
hs.Cut({0, 0}, {1, 1});
double best = hs.Maximize({1, 2});
Time: $\mathcal{O}(n)$

"Point.h", "LineIntersection.h", "Angle.h" 62 lines

```
struct HalfplaneSet : multimap<Angle, Point> {
  using Iter = multimap<Angle, Point>::iterator;

  HalfplaneSet() {
    insert({{+1, 0}, {-kInf, -kInf}});
    insert({{0, +1}, {+kInf, -kInf}});
    insert({{-1, 0}, {+kInf, +kInf}});
    insert({{0, -1}, {-kInf, +kInf}});
  }

  Iter get_next(Iter it) {
    return (next(it) == end() ? begin() : next(it)); }
  Iter get_prev(Iter it) {
    return (it == begin() ? prev(end()) : prev(it)); }
  Iter fix(Iter it) { return it == end() ? begin() : it; }

  // Cuts everything to the RIGHT of a, b
  // For LEFT, just swap a with b
  void Cut(Angle a, Angle b) {
    if (empty()) return;
    int old_size = size();

    auto eval = [&](Iter it) {
      return sgn(det(a.p(), b.p(), it->second)); };
    auto intersect = [&](Iter it)  {
      return LineIntersection(a.p(), b.p(),
          it->second, it->first.p() + it->second);
    };

    auto it = fix(lower_bound(b - a));
    if (eval(it) >= 0) return;

    while (size() && eval(get_prev(it)) < 0)
      fix(erase(get_prev(it)));
    while (size() && eval(get_next(it)) < 0)
      it = fix(erase(it));

    if (empty()) return;

    if (eval(get_next(it)) > 0) it->second = intersect(it);
    else it = fix(erase(it));
    if (old_size <= 2) return;
    it = get_prev(it);
    insert(it, {b - a, intersect(it)});
    if (eval(it) == 0) erase(it);
  }

  // Maximizes dot product
  double Maximize(Angle c) {
    assert(!empty());
    auto it = fix(lower_bound(c.t90()));
    return dot(it->second, c.p());
  }

  double Area() {
    if (size() <= 2) return 0;
    double ret = 0;
    for (auto it = begin(); it != end(); ++it)
      ret += cross(it->second, get_next(it)->second);
    return ret;
  }
};
```

8.4 Misc. Point Set Problems

closestPair.h

Description: Returns the indices to the closest pair of points in the point vector *pts* after the call. The distance can be easily computed. Might fail when using floating point (distance should be arbitrarily close though).
Time: $\mathcal{O}(n \log n)$

"Point.h" 41 lines

```
using T = long long;
using Point = complex<T>;

pair<int, int> ClosestPair(vector<Point> pts) {
  int n = pts.size();

  vector<int> order(n);
  iota(order.begin(), order.end(), 0);
  sort(order.begin(), order.end(), [&](int a, int b) {
    return pts[a].real() < pts[b].real();
  });
  set<pair<T, int>> s;

  T best_dist = numeric_limits<T>::max();
  pair<int, int> sol;

  int ii = 0, jj = 0;
  while (ii < n) {
    T d = ceil(sqrt(best_dist));
    int i = order[ii], j = order[jj];

    if (i != j && pts[i].real() - pts[j].real() >= best_dist) {
      s.erase({pts[j].imag(), j});
      jj += 1;
    } else {
      auto it1 = s.lower_bound({pts[i].imag() - d, -1});
      auto it2 = s.upper_bound({pts[i].imag() + d, n});

      for (auto it = it1; it != it2; ++it) {
        T now_dist = norm(pts[i] - pts[it->second]);
        if (best_dist > now_dist) {
          best_dist = now_dist;
          sol = {i, it->second};
        }
      }
      s.insert({pts[i].imag(), i});
      ii += 1;
    }
  }
  return sol;
}
```

kdTree.h

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

"Point.h" 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 the box is wider than high (not best
          heuristic...)
      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);
  }
};
```

DelaunayTriangulation.h

Description: Computes the Delaunay triangulation of a set of points. Each circumcircle contains none of the input points. If any three points are colinear or any four are on the same circle, behavior is undefined.
Time: $\mathcal{O}\left(n^2\right)$

"Point.h", "3dHull.h" 10 lines

```
template<class P, class F>
void delaunay(vector<P>& ps, F trifun) {
  if (sz(ps) == 3) { int d = (ps[0].cross(ps[1], ps[2]) < 0);
    trifun(0,1+d,2-d); }
  vector<P3> p3;
  trav(p, ps) p3.emplace_back(p.x, p.y, p.dist2());
  if (sz(ps) > 3) trav(t, hull3d(p3)) if ((p3[t.b]-p3[t.a]).
      cross(p3[t.c]-p3[t.a]).dot(P3(0,0,1)) < 0)
    trifun(t.a, t.c, t.b);
}
```

8.5 3D

PolyhedronVolume.h

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

6 lines

```
template <class V, class L>
double signed_poly_volume(const V& p, const L& trilist) {
  double v = 0;
  trav(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.

28 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 norm() const { return x*x + y*y + z*z; }
  double abs() const { return sqrt((double)norm()); }
  P unit() const { return *this / (T)abs(); } //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" 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);
    }
  }
  trav(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: Conversions to/from spherical coordinates and great circle distance formula

18 lines

```
Point3D FromSpherical(double r, double lat, double lon) {
  return Point3D{
    r * cos(lat) * cos(lon),
    r * cos(lat) * sin(lon),
    r * sin(lat)};
}

void ToSpherical(Point3D p, double& r,
                 double& lat, double& lon) {
  r = p.abs(); lat = asin(p.z / r); lon = atan2(p.y, p.x);
}

double SphericalDistance(double r, double lat1, double lon1,
                         double lat2, double lon2) {
  double d = (FromSpherical(1.0, lat1, lon1)
         - FromSpherical(1.0, lat2, lon2)).abs();
  return 2 * r * asin(d / 2);
}
```

Strings (9)

KMP.h

Description: pi[x] is the length of the longest prefix of s that ends at x (exclusively), other than s[0..x) itself. This is used by Match() to find all occurences of a string.
Usage: ComputePi("alabala") => {-1, 0, 0, 1, 0, 1, 2, 3}
Match("atoat", "atoatoat") => {4, 7}
Time: $\mathcal{O}\left(N\right)$

24 lines

```
vector<int> ComputePi(string s) {
  int n = s.size();
  vector<int> pi(n + 1, -1);

  for (int i = 0; i < n; ++i) {
    int j = pi[i];
    while (j != -1 && s[j] != s[i]) j = pi[j];
    pi[i + 1] = j + 1;
  }

  return pi;
}

vector<int> Match(string text, string pat) {
  vector<int> pi = ComputePi(pat), ret;
  int j = 0;

  for (int i = 0; i < (int)text.size(); ++i) {
    while (j != -1 && pat[j] != text[i]) j = pi[j];
    if (++j == pat.size())
      ret.push_back(i), j = pi[j];
  }
  return ret;
}
```

ZFunction.h

Description: Given a string s, computes the length of the longest common prefix of s[i..] and s[0..] for each i > 0 !!
Usage: `Zfunction("abacaba") => {0, 0, 1, 0, 3, 0, 1}`
Time: $\mathcal{O}(N)$

<bits/stdc++.h> 21 lines

```
vector<int> ZFunction(string s) {
  int n = s.size();
  vector<int> z(n, 0);
  int L = 0, R = 0;
  for (int i = 1; i < n; i++) {
    if (i > R) {
      L = R = i;
      while (R < n && s[R - L] == s[R]) R++;
      z[i] = R - L; R--;
    } else {
      int k = i-L;
      if (z[k] < R - i + 1) z[i] = z[k];
      else {
        L = i;
        while (R < n && s[R - L] == s[R]) R++;
        z[i] = R - L; R--;
      }
    }
  }
  return z;
}
```

Manacher.h

Description: Given a string s, computes the length of the longest palindromes centered in each position (for parity == 1) or between each pair of adjacent positions (for parity == 0).
Usage: `Manacher("abacaba", 1) => {0, 1, 0, 3, 0, 1, 0}`
`Manacher("aabbaa", 0) => {1, 0, 3, 0, 1}`
Time: $\mathcal{O}(N)$

14 lines

```
vector<int> Manacher(string s, bool parity) {
  int n = s.size(), z = parity, l = 0, r = 0;
  vector<int> ret(n - !z, 0);

  for (int i = 0; i < n - !z; ++i) {
    if (i + !z < r) ret[i] = min(r - i, ret[l + r - i - !z]);
    int L = i - ret[i] + !z, R = i + ret[i];
    while (L - 1 >= 0 && R + 1 < n && s[L - 1] == s[R + 1])
      ++ret[i], --L, ++R;
    if (R > r) l = L, r = R;
  }

  return ret;
}
```

PalindromicTree.h

Description: A trie-like structure for keeping track of palindromes of a string s. It has two roots, 0 (for even palindromes) and 1 (for odd palindromes). Each node stores the length of the palindrome, the count and a link to the longest "aligned" subpalindrome. Can be made online from left to right
Time: $\mathcal{O}(N)$

53 lines

```
struct PalTree {
  struct Node {
    map<char, int> leg;
    int link, len, cnt;
  };
  vector<Node> T;
  int nodes = 2;

  PalTree(string str) : T(str.size() + 2) {
    T[1].link = T[1].len = 0;
    T[0].link = T[0].len = -1;

    int last = 0;
    for (int i = 0; i < (int)str.size(); ++i) {
      char now = str[i];

      int node = last;
      while (now != str[i - T[node].len - 1])
        node = T[node].link;

      if (T[node].leg.count(now)) {
        node = T[node].leg[now];
        T[node].cnt += 1;
        last = node;
        continue;
      }

      int cur = nodes++;
      T[cur].len = T[node].len + 2;
      T[node].leg[now] = cur;

      int link = T[node].link;
      while (link != -1) {
        if (now == str[i - T[link].len - 1] &&
            T[link].leg.count(now)) {
          link = T[link].leg[now];
          break;
        }
        link = T[link].link;
      }
      if (link <= 0) link = 1;

      T[cur].link = link;
      T[cur].cnt = 1;

      last = cur;
    }

    for (int node = nodes - 1; node > 0; --node) {
      T[T[node].link].cnt += T[node].cnt;
    }
  }
};
```

MinRotation.h

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

11 lines

```
int MinRotation(string s) {
  int a = 0, n = s.size(); s += s;
  for (int b = 0; b < n; ++b)
    for (int i = 0; i < n; ++i) {
      if (a + i == b || s[a + i] < s[b + i]) {
        b += max(0, i - 1); break;
      }
      if (s[a + i] > s[b + i]) { a = b; break; }
    }
  return a;
}
```

SuffixArray.h

Description: Builds suffix array for a string. $a[i]$ is the starting index of the suffix which is i-th in the sorted suffix array. The `lcp` function calculates longest common prefixes for indices. Can also sort cyclic permutations
Memory: $\mathcal{O}(N)\,/\mathcal{O}(N\log N)$
Time: $\mathcal{O}(N\log N)$ where N is the length of the string for creation of the SA. $\mathcal{O}(log)$ for LCP.

75 lines

```
struct SuffixArray {
  int n, csz;
  vector<vector<int>> classes;
  vector<int> cnt, order, oldc, newc, left;
  string str;

  SuffixArray(string s, bool cyclic) :
    n(s.size() + !cyclic), csz(max(n, 256)), cnt(csz),
    order(n), oldc(n), newc(n), left(n), str(s) {
      if (!cyclic) str += '\0';
  }

  vector<int> Build() {
    for (int i = 0; i < n; ++i) {
      oldc[i] = newc[i] = str[i];
      order[i] = left[i] = i;
    }

    for (int step = 1; step <= 2 * n; step *= 2) {
      // Counting sort (can be replaced by sort with left)
      // although not trivial
      fill(cnt.begin(), cnt.end(), 0);
      for (int i = 0; i < n; ++i) ++cnt[oldc[left[i]]];
      for (int i = 1; i < csz; ++i) cnt[i] += cnt[i - 1];
      for (int i = n - 1; i >= 0; --i)
        order[--cnt[oldc[left[i]]]] = left[i];

      newc[order[0]] = 0;

      for (int i = 1; i < n; ++i) {
        int now1 = order[i], last1 = order[i - 1],
            now2 = (now1 + step / 2) % n,
            last2 = (last1 + step / 2) % n;

        newc[now1] = newc[last1] + (oldc[now1] != oldc[last1]
               or oldc[now2] != oldc[last2]);
      }

      classes.push_back(newc);
      swap(oldc, newc);

      for (int i = 0; i < n; ++i) {
        left[i] = (order[i] + n - step) % n;
      }
    }

    return order;
  }

  int Compare(int i, int j, int len) {
    for (int step = 0; len; ++step, len /= 2) {
      if (len % 2 == 0) continue;

      int ret = classes[step][i] - classes[step][j];
      if (ret != 0) return ret < 0 ? -1 : 1;

      i = (i + (1 << step)) % n;
      j = (j + (1 << step)) % n;
    }
    return 0;
  }

  int GetLCP(int i, int j) {
    if (i == j) return str.back() == '\0' ? n - i - 1 : n;
    int ans = 0;
    for (int step = classes.size() - 1; step >= 0; --step) {
      if (classes[step][i] == classes[step][j]) {
        i = (i + (1 << step)) % n;
        j = (j + (1 << step)) % n;
        ans += (1 << step);
      }
    }
    return min(ans, n); // if cyclic
  }
};
```

SuffixAutomaton.h

Description: Builds an automaton of all the suffixes of a given string (online from left to right). For each character c, do sa.ConsumeChar(c) You can change char to int and add negative numbers to support multiple strings.
Time: $\mathcal{O}(log(sigma))$ amortized per character added

<bits/stdc++.h> 58 lines

```
struct SuffixAutomaton {
  struct Node {
    int link, len;
    map<char, int> leg;
  };
  vector<Node> T;
  int last = 0, nodes = 1;

  SuffixAutomaton(int sz) : T(2 * sz + 1) {
    T[0].link = -1;
    T[0].len = 0;
  }

  // Adds another character to the automaton
  // and returns the node of the whole new string
  // (the suffixes of that are parents in the link tree)
  int ConsumeChar(char c) {
    // Add state for whole string
    int cur = nodes++, node = last;
    T[cur].len = T[last].len + 1;
    T[cur].link = 0;

    // Add transitions to all suffixes which do not have one
    // already
    while (node != -1 && T[node].leg.count(c) == 0) {
      T[node].leg[c] = cur;
      node = T[node].link;
    }

    if (node != -1) {
      int old = T[node].leg[c];

      if (T[old].len == T[node].len + 1) {
        T[cur].link = old;
      } else {
        int clone = nodes++;

        T[clone].leg = T[old].leg;
        T[clone].len = T[node].len + 1;
        T[clone].link = T[old].link;
        T[old].link = T[cur].link = clone;

        while (node != -1 && T[node].leg[c] == old) {
          T[node].leg[c] = clone;
          node = T[node].link;
        }
      }
    }
    return last = cur;
  }
```

```
  // Runs through the automaton
  int Go(int node, char c) {
    while (node != -1 && T[node].leg.count(c) == 0)
      node = T[node].link;
    return (node == -1 ? 0 : T[node].leg[c]);
  }
};
```

AhoCorasick.h

Description: Aho-Corasick algorithm builds an automaton for multiple pattern string matching
Time: $\mathcal{O}(N * log(sigma))$ where N is the total length

<bits/stdc++.h> 48 lines

```
struct AhoCorasick {
  struct Node {
    int link;
    map<char, int> leg;
  };
  vector<Node> T;
  int root = 0, nodes = 1;

  AhoCorasick(int sz) : T(sz) {}

  // Adds a word to trie and returns the end node
  int AddWord(const string &word) {
    int node = root;
    for (auto c : word) {
      auto &nxt = T[node].leg[c];
      if (nxt == 0) nxt = nodes++;
      node = nxt;
    }
    return node;
  }

  // Advances from a node with a character (like an automaton)
  int Advance(int node, char chr) {
    while (node != -1 && T[node].leg.count(chr) == 0)
      node = T[node].link;
    if (node == -1) return root;
    return T[node].leg[chr];
  }

  // Builds links
  void BuildLinks() {
    queue<int> Q;
    Q.push(root);
    T[root].link = -1;

    while (!Q.empty()) {
      int node = Q.front();
      Q.pop();

      for (auto &p : T[node].leg) {
        int vec = p.second;
        char chr = p.first;
        T[vec].link = Advance(T[node].link, chr);
        Q.push(vec);
      }
    }
  }
};
```

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 [inclusive, exclusive).
Time: $\mathcal{O}(\log N)$

35 lines

```
struct IntervalContainer {
  map<int, int> s;
  using Iter = map<int, int>::iterator;

  Iter AddInterval(int l, int r) {
    if (l == r) return s.end();
    Iter it = s.lower_bound(l);
    while (it != s.end() && it->first <= r) {
      r = max(r, it->second);
      it = s.erase(it);
    }
    while (it != s.begin() && (--it)->second >= l) {
      l = min(l, it->first);
      r = max(r, it->second);
      it = s.erase(it);
    }
    return s.insert({l, r}).first;
  }

  Iter FindInterval(int x) {
    auto it = s.upper_bound(x);
    if (it == s.begin() or (--it)->second <= x)
      return s.end();
    return it;
  }

  void RemoveInterval(int l, int r) {
    if (l == r) return;
    auto it = AddInterval(l, r);
    int l2 = it->first, r2 = it->second;
    s.erase(it);
    if (l != l2) s.insert({l2, l});
    if (r != r2) s.insert({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: $\mathcal{O}(N \log N)$

19 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 cb for each such interval.
Usage: `ConstantIntervals(0, v.size(), [&](int x){return v[x];}, [&](int lo, int hi, T val){...});`
Time: $\mathcal{O}\left(k \log \frac{n}{k}\right)$ 20 lines

```
template<class Func, class Callback, class T>
void recurse(int from, int to, Func f, Callback cb,
             int& i, T& p, T q) {
  if (p == q) return;
  if (from == to) {
    cb(i, to, p);
    i = to; p = q;
  } else {
    int mid = (from + to) / 2;
    recurse(from, mid, f, cb, i, p, f(mid));
    recurse(mid + 1, to, f, cb, i, p, q);
  }
}
template<class Func, class Callback>
void ConstantIntervals(int from, int to, Func f, Callback cb) {
  if (to <= from) return;
  int i = from; auto p = f(i), q = f(to - 1);
  recurse(from, to - 1, f, cb, i, p, q);
  cb(i, to, q);
}
```

10.2 Misc. algorithms

TernarySearch.h
Description: Find the smallest i in $[a, b]$ that maximizes $f(i)$, assuming that $f(a) < \ldots < f(i) \geq \cdots \geq f(b)$. To reverse which of the sides allows non-strict inequalities, change the < marked with (A) to <=, and reverse the loop at (B). To minimize f, change it to >, also at (B).
Usage: `int ind = TernarySearch(0,n-1,[&](int i){return a[i];});`
Time: $\mathcal{O}(\log(b-a))$ 12 lines

```
template<class Func>
int TernarySearch(int a, int b, Func f) {
  assert(a <= b);
  while (b - a >= 5) {
    int mid = (a + b) / 2;
    if (f(mid) < f(mid + 1)) a = mid; // (A)
    else b = mid + 1;
  }
  for (int i = a + 1; i <= b; ++i)
    if (f(a) < f(i)) a = i; // (B)
  return a;
}
```

AlphaBeta.h
Description: Uses the alpha-beta pruning method to find score values for states in games (minimax) 8 lines

```
int AlphaBeta(state s, int alpha, int beta) {
  if (s.finished()) return s.score();
  for (state t : s.next()) {
    alpha = max(alpha, -AlphaBeta(t, -beta, -alpha));
    if (alpha >= beta) break;
  }
  return alpha;
}
```

10.3 Dynamic programming

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

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 $f(b, c) \leq f(a, d)$ and $f(a, c) + f(b, d) \leq f(a, d) + f(b, c)$ for all $a \leq b \leq c \leq d$. Consider also: LineContainer (ch. Data structures), monotone queues, ternary search.
Time: $\mathcal{O}\left(N^2\right)$ 1 lines

10.4 Debugging tricks

- `signal(SIGSEGV, [](int) { _Exit(0); });` converts segfaults into Wrong Answers. Similarly one can catch SIGABRT (assertion failures) and SIGFPE (zero divisions). `_GLIBCXX_DEBUG` violations generate SIGABRT (or SIGSEGV on gcc 5.4.0 apparently).
- `feenableexcept(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 = 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.

10.5.2 Pragmas

- `#pragma GCC optimize ("Ofast")` will make GCC auto-vectorize for loops and optimizes floating points 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).

Unrolling.h 5 lines

```
#define F {...; ++i;}
int i = from;
while (i&3 && i < to) F // for alignment, if needed
while (i + 4 <= to) { F F F F }
while (i < to) F
```

Techniques (A)

techniques.txt

159 lines

```
Recursion
Divide and conquer
  Finding interesting points in N log N
Algorithm analysis
  Master theorem
  Amortized time complexity
Greedy algorithm
  Scheduling
  Max contigous subvector sum
  Invariants
  Huffman encoding
Graph teory
  Dynamic graphs (extra book-keeping)
  Breadth first search
  Depth first search
  * Normal trees / DFS trees
  Dijkstra's algoritm
  MST: Prim's algoritm
  Bellman-Ford
  Konig's theorem and vertex cover
  Min-cost max flow
  Lovasz toggle
  Matrix tree theorem
  Maximal matching, general graphs
  Hopcroft-Karp
  Hall's marriage theorem
  Graphical sequences
  Floyd-Warshall
  Eulercykler
  Flow networks
  * Augumenting 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 programmering
  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
  Euklidean algorithm
  Modular arithmetic
  * Modular multiplication
  * Modular inverses
  * Modular exponentiation by squaring
  Chinese remainder theorem
  Fermat's small theorem
  Euler's theorem
  Phi function
  Frobenius 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 polynom 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)
  Pull/push-technique on trees
  Heavy-light decomposition
  Centroid decomposition
  Lazy propagation
  Self-balancing trees
  Convex hull trick (wcipeg.com/wiki/Convex_hull_trick)
  Monotone queues / monotone stacks / sliding queues
  Sliding queue using 2 stacks
  Persistent segment tree
```