Federal University of Rio de Janeiro

UFRJ - Time Feliz ^_^

Chris Ciafrino e Letícia Freire

adapted from KTH ACM Contest Template Library

2019

Contest (1)

template.cpp

29 lines

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

using lint = long long;
using ldouble = long double;

const double PI = static_cast<double>(acosl(-1.0));

// Retorna -1 se a < b, 0 se a = b e 1 se a > b.
int cmp_double(double a, double b = 0, double eps = 1e-9) {
    return a + eps > b ? b + eps > a ? 0 : 1 : -1;
}

//be careful with cin optimization
string read_string() {
    char *str;
    scanf("%ms", &str);
    string result(str);
    free(str);
    return result;
}

int main() {
    ios_base::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    cin.exceptions(cin.failbit);

    return 0;
}
```

hash.sh

1 lines

```
tr -d '[:space:]' | md5sum
```

hash-cpp.sh

1 lines

```
cpp -P -fpreprocessed | tr -d '[:space:]' | md5sum
```

vimrc

29 lines

```
set nocp ai bs=2 hls ic is lbr ls=2 mouse=a nu ru sc scs
   ↪smd so=3 sw=4 ts=4
filetype plugin indent on
syn on
map gA m'ggVG"+y''

com -range=% -nargs=1 P exe "<line1>,<line2>!".<q-args> |y|
   ↪sil u|echom @"
com -range=% Hash <line1>,<line2>P tr -d '[:space:]' |
   ↪md5sum
au FileType cpp com! -buffer -range=% Hash <line1>,<line2>P
   ↪ cpp -dD -P -fpreprocessed | tr -d '[:space:]' |
   ↪md5sum

:autocmd BufNewFile *.cpp 0r /etc/vim/templates/cp.cpp

" shift+arrow selection
nmap <S-Up> v<Up>
nmap <S-Down> v<Down>
nmap <S-Left> v<Left>
nmap <S-Right> v<Right>
vmap <S-Up> <Up>
vmap <S-Down> <Down>
vmap <S-Left> <Left>
vmap <S-Right> <Right>
imap <S-Up> <Esc>v<Up>
imap <S-Down> <Esc>v<Down>
imap <S-Left> <Esc>v<Left>
imap <S-Right> <Esc>v<Right>
vmap <C-c> y<Esc>i
vmap <C-x> d<Esc>i
map <C-v> pi
imap <C-v> <Esc>pi
imap <C-z> <Esc>ui
```

troubleshoot.txt

62 lines

```
Pre-submit:
Write down your thoughts, even if they don't completely
   ↪solve the problem.
Stay organized (don't leave papers all over the place)!
Give your variables (and files) useful names!
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?
Remove debug output.
Make sure to submit the right file.
You should know what your code is doing ...

Wrong answer:
Read the full problem statement again.
Have you understood the problem correctly?
Are you sure your algorithm works?
Try writing a slow (but correct) solution.
Can your algorithm handle the whole range of input?
Did you consider corner cases (n=1) or other special cases?
Print your solution! Print debug output, as well.
Is your output format correct? (including whitespace)
Are you clearing all data structures between test cases?
Any uninitialized variables?
Any undefined behavior (array out of bounds)?
Any overflows or NaNs (shifting ll by 64 bits or more)?
Confusing N and M, i and j, etc.?
Confusing ++i and i++?
Correctly account for numbers close to (but not) zero.
For line sweeping over polygons, correctly deal with
   ↪vertices.
Are you sure the STL functions you use work as you think?
Add some assertions, maybe resubmit.
Create some test cases to run your algorithm on.
Go through the algorithm for a simple case.
Go through this list again.
Explain your algorithm to a teammate.
Ask the teammate to look at your code.
Go for a small walk, e.g. to the toilet.
Rewrite your solution from the start or let a teammate 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's your complexity? Large TL does not mean that
   ↪something simple (like NlogN) isn't intended.
Are you copying a lot of unnecessary data? (References)
Avoid vector, map. (use arrays/unordered_map)
How big is the input and output? (consider FastI)
What do your teammates think about your algorithm?

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

Mathematics (2)

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.

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

2.2 Master theorem

Given a recurrence of the form $T(n) = aT(\frac{n}{b}) + f(n)$ where $a \geq 1$, $b > 1$.

1) If $f(n) = \mathcal{O}(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$, then

$$T(n) = \Theta(n^{\log_b a})$$

2) If $f(n) = \Theta(n^{\log_b a})$, then

$$T(n) = \Theta(n^{\log_b a} \log n)$$

3) If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$ (and $af(\frac{n}{b}) \leq cf(n)$ for some $c < 1$ for all n sufficiently large), then

$$T(n) = \Theta(f(n))$$

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

2.4 Geometry

2.4.1 Triangles

Side lengths: a, b, c

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

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

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

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

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: $\frac{\sin\alpha}{a} = \frac{\sin\beta}{b} = \frac{\sin\gamma}{c} = \frac{1}{2R}$

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

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

Pick's: A polygon on an integer grid strictly containing i lattice points and having b lattice points on the boundary has area $i + \frac{b}{2} - 1$. (Nothing similar in higher dimensions)

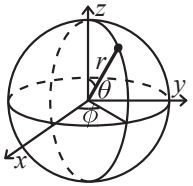
2.4.2 Quadrilaterals

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



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

2.4.4 Centroid of a polygon

The x coordinate of the centroid of a polygon is given by $\frac{1}{3A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_iy_{i+1} - x_{i+1}y_i)$, where A is twice the signed area of the polygon.

2.5 Derivatives/Integrals

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

Integration by parts:

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

2.6 Sums

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

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

2.7 Series

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

2.8 Probability theory

Let X be a discrete random variable with probability $p_X(x)$ of assuming the value x. It will then have an expected value (mean) $\mu = \mathbb{E}(X) = \sum_x xp_X(x)$ 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,

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

2.8.1 Gambler's Ruin

Em um jogo no qual ganhamos cada aposta com probabilidade p e perdemos com probabilidade $q := 1 - p$, paramos quando ganhamos B ou perdemos A. Então $Prob(\text{ganhar B}) = \frac{1-(p/q)^B}{1-(p/q)^{A+B}}$.

2.8.2 Bertrand's ballot theorem

In an election where candidate A receives p votes and candidate B receives q votes with $p > q$, the probability that A will be strictly ahead of B throughout the count is $\frac{p-q}{p+q}$. If draw is a possible outcome, the probability will be equal to $\frac{p+1-q}{p+1}$, to find how many possible outcomes for both cases just multiply by $\binom{p+q}{q}$

2.8.3 Discrete distributions

Binomial distribution

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

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

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

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

First success distribution

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

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

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

Poisson distribution

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

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

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

2.8.4 Continuous distributions

Uniform distribution

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

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

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

Exponential distribution

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

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

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

Normal distribution

Most real random values with mean 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

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

2.9 Markov chains

A *Markov chain* is a discrete random process with the property that the next state depends only on the current state. Let $X_1, X_2, \ldots$ be 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 absorbing chain if
1. there is at least one absorbing state and
2. it is possible to go from any state to at least one absorbing state in a finite number of steps.
A Markov chain is an A-chain if the states can be partitioned into two sets **A** and **G**, such that all states in **A** are absorbing ($p_{ii} = 1$), and all states in **G** leads to an absorbing state in **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)

HashMap.h

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

```
#include <bits/extc++.h>
// To use most bits rather than just the lowest ones:
struct chash { // large odd number for C
  const uint64_t C = lint(4e18 * acos(0)) | 71;
  lint operator()(lint x) const { return __builtin_bswap64(
    ↪x*C); }
};
__gnu_pbds::gp_hash_table<lint,int,chash> h
  ↪({},{},{},{},{1<<16}); // hash-cpp-all = 52
  ↪e9426c73416cc4ff68f3f06c734558
```

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
} // hash-cpp-all = 782797f91ca134bf996558987dbf1924
```

DSU.h

Description: Disjoint-set data structure.
Time: $\mathcal{O}(\alpha(N))$

19 lines

```
struct UF {
    int n;
    vector<int> parent, rank;
    UF(int _n): n(_n), parent(n), rank(n, 0) {
        iota(parent.begin(), parent.end(), 0);
    }
    int find(int v) {
        if (parent[v] == v) return v;
        return parent[v] = find(parent[v]);
    }
    bool unite(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) return false;
        if (rank[a] > rank[b]) swap(a, b);
        parent[a] = b;
        if (rank[a] == rank[b]) ++rank[b];
        return true;
    }
}; // hash-cpp-all = 435133be76cd0916309585d854859fe3
```

DSURoll.h

Description: Disjoint-set data structure with undo.
Usage: `int t = uf.time(); ...; uf.rollback(t);`
Time: $\mathcal{O}(\log(N))$

21 lines

```
struct RollbackUF {
    vector<int> e; vector<pair<int,int>> st;
    RollbackUF(int n) : e(n, -1) {}
    int size(int x) { return -e[find(x)]; }
    int find(int x) { return e[x] < 0 ? x : find(e[x]); }
    int time() { return st.size(); }
    void rollback(int t) {
        for (int i = time(); i --> t;)
            e[st[i].first] = st[i].second;
        st.resize(t);
    }
    bool unite(int a, int b) {
        a = find(a), b = find(b);
        if (a == b) return false;
        if (e[a] > e[b]) swap(a, b);
        st.push_back({a, e[a]});
        st.push_back({b, e[b]});
        e[a] += e[b]; e[b] = a;
        return true;
    }
}; // hash-cpp-all = 7ddf1d63541b7bda1fc6daed3c938fb6
```

MinQueue.h

Description: Structure that supports all operations of a queue and get the minimum/maximum active value in the queue. Useful for sliding window 1D and 2D. For 2D problems, you will need to pre-compute another matrix, by making a row-wise traversal, and calculating the min/-max value beginning in each cell. Then you just make a column-wise traverse as they were each an independent array.

Time: $\mathcal{O}(1)$

24 lines

```
template<typename T>
struct minQueue {
  int lx, rx, sum;
  deque<pair<T, T>> q;
  minQueue() { lx = 1; rx = 0; sum = 0; }
  void clear() { lx = 1, rx = 0, sum = 0; q.clear(); }
  void push(T delta) {
     // q.back().first + sum <= delta for a maxQueue
    while(!q.empty() && q.back().first + sum >= delta)
      q.pop_back();
    q.emplace_back(delta - sum, ++rx);
  }
  void pop() {
    if (!q.empty() && q.front().second == lx++)
      q.pop_front();
  }
  void add(T delta) {
    sum += delta;
  }
  T getMin() {
    return q.front().first + sum;
  }
  int size() { return rx-lx+1; }
}; // hash-cpp-all = d40e772246502e3ab2ec99a1b0943803
```

SegTree.h

Description: Zero-indexed sum-tree. Bounds are inclusive to the left and exclusive to the right. Can be changed by modifying T, LOW and f.
Time: $\mathcal{O}(\log N)$

24 lines

```
template<typename T> struct tree_t {
    static const T LOW = -(1<<29);
    T f(T a, T b) { return max(a,b); } // (any associative
       ↪fn)
    vector<T> s; int n;
    tree_t() {}
    tree_t(int size, T def = LOW) : s(2*size, def), n(size)
       ↪ {}
    tree_t(const vector<T> &other) : n(other.size()), s(2*
       ↪other.size(), -(1<<29)) {
        copy(other.begin(), other.end(), s.begin() + n);
        for (int i = n; i-- > 1; )
            s[i] = f(s[i<<1], s[i<<1|1]);
    }
    void update(int pos, T val) {
        for (s[pos += n] = val; pos > 1; pos >>= 1)
            s[pos >> 1] = f(s[pos & ~1], s[pos | 1]);
    }
    T query(int l, int r) { // query [b, e)
        T ra = LOW, rb = LOW;
        for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
            if (l % 2) ra = f(ra, s[l++]);
            if (r % 2) rb = f(s[--r], rb);
        }
        return f(ra, rb);
    }
}; // hash-cpp-all = a729b414eb606758f023b71fb0cf462b
```

SparseSegTree.h

Description: Sparse Segment Tree with point update. Doesnt allocate storage for nodes with no data. Use BumpAllocator for better performance!

39 lines

```
const int SZ = 1<<19;
template<class T> struct node_t {
  T delta = 0; node_t<T>* c[2];
  node_t() { c[0] = c[1] = nullptr; }
  void upd(int pos, T v, int L = 0, int R = SZ-1) { // add
     ↪v
    if (L == pos && R == pos) { delta += v; return; }
    int M = (L + R)>>1;
    if (pos <= M) {
      if (!c[0]) c[0] = new node_t();
      c[0]->upd(pos, v, L, M);
    } else {
      if (!c[1]) c[1] = new node_t();
      c[1]->upd(pos, v, M+1, R);
    }
    delta = 0;
    for (int i = 0; i < 2; ++i) if (c[i]) delta += c[i]->
       ↪delta;
  }
  T query(int lx, int rx, int L = 0, int R = SZ-1) { //
     ↪query sum of segment
    if (rx < L || R < lx) return 0;
    if (lx <= L && R <= rx) return delta;
    int M = (L + R)>>1; T res = 0;
    if (c[0]) res += c[0]->query(lx, rx, L, M);
    if (c[1]) res += c[1]->query(lx, rx, M+1, R);
    return res;
  }
  void upd(int pos, node_t *a, node_t *b, int L = 0, int R
     ↪= SZ-1) {
    if (L != R) {
      int M = (L + R)>>1;
      if (pos <= M) {
        if (!c[0]) c[0] = new node_t();
        c[0]->upd(pos, a ? a->c[0] : nullptr, b ? b->c[0] :
           ↪ nullptr, L, M);
      } else {
        if (!c[1]) c[1] = new node_t();
        c[1]->upd(pos,a ? a->c[1] : nullptr, b ? b->c[1] :
           ↪nullptr, M+1, R);
      }
    }
    delta = (a ? a->delta : 0)+(b ? b->delta : 0);
  }
}; // hash-cpp-all = 014f68f7f91eb9db32df37e453e7fdd2
```

LazySegmentTree.h

Description: Better SegTree. Range Sum, can be extended to max/min/product/gcd, pay attention to propagate, f and update functions when extending. Be careful with each initialization aswell.
Time: $\mathcal{O}(\lg(N))$

44 lines

```
template<typename T, typename Q> struct segtree_t {
    int n;
    vector<T> tree;
    vector<Q> lazy, og;
    segtree_t(int N) : n(N), tree(4*N), lazy(4*N) {}
    segtree_t(const vector<Q> &other) : n(other.size()), og
       ↪(other),
    tree(4*n), lazy(4*n) { build(1, 0, n-1); }
    T f(const T &a, const T &b) { return (a + b); }
    T build(int v, int l, int r) {
        lazy[v] = 0;
        if (l == r) return tree[v] = og[l];
        int m = l + (r - l)/2;
        return tree[v] = f(build(v<<1, l, m), build(v<<1|1,
           ↪ m+1, r));
    }
    void propagate(int v, int l, int r) {
```

```
        if (!lazy[v]) return;
        int m = l + (r - l)/2;
        tree[v<<1] += lazy[v] * (m - l + 1);
        tree[v<<1|1] += lazy[v] * (r - (m + 1) + 1);
        for (int i = 0; i < 2; ++i) lazy[v<<1|i] += lazy[v
            ↪];
        lazy[v] = 0;
    }
    T query(int a, int b) { return query(a, b, 1, 0, n-1);
        ↪}
    T query(int a, int b, int v, int l, int r) {
        if (b < l || r < a) return 0;
        if (a <= l && r <= b) return tree[v];
        propagate(v,l, r);
        int m = l + (r - l)/2;
        return f(query(a, b, v<<1,l, m), query(a, b, v
            ↪<<1|1, m+1, r));
    }
    T update(int a, int b, Q delta) { return update(a, b,
        ↪delta, 1, 0, n-1); }
    T update(int a, int b, Q delta, int v, int l, int r) {
        if (b < l || r < a) return tree[v];
        if (a <= l && r <= b) {
            tree[v] += delta * (r - l + 1);
            lazy[v] += delta;
            return tree[v];
        }
        propagate(v,l, r);
        int m = l + (r - l)/2;
        return tree[v] = f(update(a, b, delta, v<<1, l, m),
            update(a, b, delta, v<<1|1, m+1, r));
    }
}; // hash-cpp-all = f2afda22796df22ab2ce696bc83298c3
```

DynamicSegTree.h

Description: Dynamic Segment Tree with lazy propagation. Allows range query, range update (increment and assignment). For assignment change all += to = in push and update functions. If not using lazy, remove all push related function calls.
Usage: vector<int> a;
node *segtree = build(0, n, a);
Time: $\mathcal{O}(\lg(N))$

73 lines

```
struct node {
  node *l, *r;
  lint minv, sumv, lazy;
  int lx, rx;
};
void push(node *v) {
  if(v != nullptr && v->lazy) {
    v->minv += v->lazy;
    v->sumv += v->lazy * (v->rx - v->lx + 1);
    if(v->l) v->l->lazy += v->lazy;
    if(v->r) v->r->lazy += v->lazy;
    v->lazy = 0;
  }
}
void update(node *v, int lx, int rx, lint delta) {
  push(v);
  if(rx < v->lx || v->rx < lx) return;
  if(lx <= v->lx && v->rx <= rx) {
    v->lazy += delta;
    push(v);
    return;
  }
  update(v->l, lx, rx, delta);
  update(v->r, lx, rx, delta);
  push(v->l);
  v->minv = min(v->l->minv, v->r->minv);
  v->sumv = v->l->sumv + v->r->sumv;
}
// without propagation, way faster in practice
void upd(node *v, int lx, int rx, lint delta) {
  if(rx < v->lx || v->rx < lx) return;
  if(v->lx == v->rx) {
    v->lazy += delta;
    v->minv += delta;
    v->sumv += delta;
    return;
  }
  upd(v->l, lx, rx, delta);
  upd(v->r, lx, rx, delta);
  v->minv = min(v->l->minv, v->r->minv) + v->lazy;
  v->sumv = v->l->sumv + v->r->sumv + v->lazy * (v->rx - v
      ↪->lx + 1);
}
lint mquery(node *v, int lx, int rx) {
  push(v);
  if(rx < v->lx || v->rx < lx) return 1e19;
  if(lx <= v->lx && v->rx <= rx) return v->minv;
  return min(mquery(v->l, lx, rx), mquery(v->r, lx, rx));
}
lint squery(node *v, int lx, int rx) {
  push(v);
  if(rx < v->lx || v->rx < lx) return 0;
  if(lx <= v->lx && v->rx <= rx) return v->sumv;
  return squery(v->l, lx, rx) + squery(v->r, lx, rx);
}
node *build(int lx, int rx) {
  node *v = new node();
  v->lx = lx; v->rx = rx;
  if(lx == rx) {
    v->lazy = 0;
    v->l = v->r = nullptr;
    v->minv = v->sumv = 0;
  }
  else {
    v->l = build(lx, (lx + rx)/2);
    v->r = build((lx + rx)/2 + 1, rx);
    v->minv = min(v->l->minv, v->r->minv);
    v->sumv = v->l->sumv + v->r->sumv;
    v->lazy = 0;
  }
  return v;
}

// hash-cpp-all = 3c95db2fe22cac488cb5b184af79d992
```

SegTree2D.h

Description: 2D Segment Tree.

"SparseSegtree.h" 25 lines

```
template<class T> struct Node {
    node_t<T> seg; Node* c[2];
    Node() { c[0] = c[1] = nullptr; }
    void upd(int x, int y, T v, int L = 0, int R = SZ-1) {
        ↪// add v
        if (L == x && R == x) { seg.upd(y,v); return; }
        int M = (L+R)>>1;
        if (x <= M) {
            if (!c[0]) c[0] = new Node();
            c[0]->upd(x,y,v,L,M);
        } else {
            if (!c[1]) c[1] = new Node();
            c[1]->upd(x,y,v,M+1,R);
        }
        seg.upd(y,v); // only for addition
        // seg.upd(y,c[0]?&c[0]->seg:nullptr,c[1]?&c[1]->
            ↪seg:nullptr);
    }
    T query(int x1, int x2, int y1, int y2, int L = 0, int
        ↪R = SZ-1) { // query sum of rectangle
        if (x1 <= L && R <= x2) return seg.query(y1,y2);
        if (x2 < L || R < x1) return 0;
        int M = (L+R)>>1; T res = 0;
        if (c[0]) res += c[0]->query(x1, x2, y1, y2, L, M);
        if (c[1]) res += c[1]->query(x1, x2, y1, y2, M+1, R
            ↪);
        return res;
    }
}; // hash-cpp-all = 09098e83e696840f2ed51ab0d0eb300c
```

MergeSortTree.h

Description: Build segment tree where each node stores a sorted version of the underlying range.
Time: $\mathcal{O}(\log^2 N)$

36 lines

```
struct MergeSortTree {
    vector<int> v, id;
    vector<vector<int>> tree;
    MergeSortTree(vector<int> &v) : v(v), tree(4*(v.size()
        ↪+1)) {
        for(int i = 0; i < v.size(); ++i) id.push_back(i);
        sort(id.begin(), id.end(), [&v](int i, int j) {
            ↪return v[i] < v[j]; });
        build(1, 0, v.size()-1);
    }
    void build(int id, int left, int right) {
        if (left == right) tree[id].push_back(id[left]);
        else {
            int mid = (left + right)>>1;
            build(id<<1, left, mid);
            build(id<<1|1, mid+1, right);
            tree[id] = vector<int>(right - left + 1);
            merge(tree[i<<1].begin(), tree[i<<1].end(),
                tree[id<<1|1].begin(), tree[id<<1|1].end(),
                tree[id].begin());
        }
    }
    // how many elements in this node have id in the range
        ↪[a,b]
    int how_many(int id, int a, int b) {
        return (int)(upper_bound(tree[id].begin(), tree[id
            ↪].end(), b)
            - lower_bound(tree[id].begin(), tree[id].end(),
                ↪ a));
    }
    int query(int id, int left, int right, int a, int b,
        ↪int x) {
        if (left == right) return v[tree[id].back()];
        int mid = (left + right)>>1;
        int lcount = how_many(id<<1, a, b);
        if (lcount >= x) return query(id<<1, left, mid, a,
            ↪b, x);
        else return query(id<<1|1, mid+1, right, a, b, x -
            ↪lcount);
    }
    int kth(int a, int b, int k) {
        return query(1, 0, v.size()-1, a, b, k);
    }
}; // hash-cpp-all = 342dec8e345cf97b7fd22b5637258ebe
```

Mo.h

Description: Mo's algorithm example problem: Count how many elements appear at least two times in given range $[l, r]$. For path queries on trees, flatten the tree by DFSing and pushing even-depth nodes at entry and odd-depth nodes at exit. If you need to squeeze Mo's in the TL and Q is greater than N, consider Hilbert Curves. Will work much faster.

Time: $(n+q)sqrt(n)$

32 lines

```
struct query_t { int l, r, id; };
int n, m, total = 0; // elements, queries, result.
const int sqn = sqrt(n), maxv = 1000000;
vector<int> values(n), freq(2*maxv), result(m);
vector<query_t> queries(m);
sort(queries.begin(), queries.end(), [sqn](const query_t &a
    ↪, const query_t &b) {
  if (a.l/sqn != b.l/sqn) return a.l < b.l;
  return a.r < b.r;
});
int l = 0, r = -1;
for(query_t &q : queries) {
  auto add = [&](int i) {
    // Change if needed
    ++freq[values[i]];
    if (freq[values[i]] == 2) total += 2;
    else if (freq[values[i]] > 2) ++total;
  };
  auto del = [&](int i) {
    // Change if needed
    --freq[values[i]];
    if (freq[values[i]] == 1) total -= 2;
    else if (freq[values[i]] > 1) --total;
  };
  while(r < q.r) add(++r);
  while(l > q.l) add(--l);
  while(r > q.r) del(r--);
  while(l < q.l) del(l++);
  result[q.id] = total;
}

// hash-cpp-all = 33f45f767453beb8f0b1c28702606ed7
```

RMQ.h

Description: Range Minimum/Maximum Queries on an array. Returns min(V[a], V[a + 1], ... V[b - 1]) in constant time. Returns a pair that holds the answer, first element is the value and the second is the index.

Usage: `rmq_t<int> rmq(values);`
`rmq.query(inclusive, inclusive);`
`rmq_t<int, greater<pair<int,int>>> rmq(values) //max query`

Time: $\mathcal{O}(|V| \log |V| + Q)$

21 lines

```
// change cmp for max query or similar
template<typename T, typename Cmp=less<pair<T, int>>>
struct rmq_t {
    Cmp cmp; vector<vector<pair<T, int>>> table;
    rmq_t() {}
    rmq_t(const vector<T> &values) {
        int n = values.size();
        table.resize(__lg(n)+1);
        table[0].resize(n);
        for (int i = 0; i < n; ++i) table[0][i] = {values[i
            ↪], i};
        for (int l = 1; l < (int)table.size(); ++l) {
            table[l].resize(n - (1<<l) + 1);
            for (int i = 0; i + (1<<l) <= n; ++i)
                table[l][i] = min(table[l-1][i], table[l
                    ↪-1][i+(1<<(l-1))], cmp);
        }
    }
    pair<T, int> query(int a, int b) {
        int l = __lg(b-a+1);
        return min(table[l][a], table[l][b-(1<<l)+1], cmp);
    }
}; // hash-cpp-all = 4fc7f76162d2b064f3cc7907da50e5c4
```

RSQ.h

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

Usage: `rsq_t<int> rsq(values);`
`rsq.query(inclusive, inclusive);`

Time: $\mathcal{O}(|V| \log |V| + Q)$

24 lines

```
template<typename T>
struct rsq_t {
    vector<vector<T>> table;
    rsq_t() {}
    rsq_t(const vector<T> &values) {
        int n = values.size();
        table.resize(__lg(n)+1); table[0].resize(n);
        for (int i = 0; i < n; ++i) table[0][i] = values[i
            ↪];
        for (int l = 1; l < (int)table.size(); ++l) {
            table[l].resize(n - (1<<l) + 1);
            for (int i = 0; i + (1<<l) <= n; ++i)
                table[l][i] = table[l-1][i] + table[l-1][i
                    ↪+(1<<(l-1))];
        }
    }
    T query(int a, int b) {
        int l = b - a + 1; T ret{};
        for (int i = (int)table.size(); i >= 0; --i)
            if ((1 << i) <= l) {
                ret += table[i][a]; a += (1<<i);
                l = b - a + 1;
            }
        return ret;
    }
}; // hash-cpp-all = 74c8911b858ee79594a61fb4ef973a41
```

FenwickTree.h

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

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

22 lines

```
template<typename T> struct FT {
  vector<T> s;
  FT(int n) : s(n) {}
  void update(int pos, T dif) { // a[pos] += dif
    for (; pos < (int)s.size(); pos |= pos + 1) s[pos] +=
        ↪dif;
  }
  T query(int pos) { // sum of values in [0, pos)
    T res = 0;
    for (; pos > 0; pos &= pos - 1) res += s[pos-1];
    return res;
  }
  int lower_bound(T sum) {// min pos st sum of [0, pos] >=
      ↪sum
    // Returns n if no sum is >= sum, or -1 if empty sum is
        ↪.
    if (sum <= 0) return -1;
    int pos = 0;
    for (int pw = 1 << 25; pw; pw >>= 1) {
      if (pos + pw <= (int)s.size() && s[pos + pw-1] < sum)
        pos += pw, sum -= s[pos-1];
    }
    return pos;
  }
}; // hash-cpp-all = d5645d6d4e6c95c755d9cf007a508be3
```

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

"FenwickTree.h" 21 lines

```
template<typename T> struct FT2 {
  vector<vector<int>> ys; vector<FT<T>> ft;
  FT2(int limx) : ys(limx) {}
  void fakeUpdate(int x, int y) {
    for (; x < (int)ys.size(); x |= x + 1) ys[x].push_back(
        ↪y);
  }
  void init() {
    for(auto v : ys) sort(v.begin(), v.end()), ft.
        ↪emplace_back(v.size());
  }
  int ind(int x, int y) {
    return (int)(lower_bound(ys[x].begin(), ys[x].end(), y)
        ↪ - ys[x].begin()); }
  void update(int x, int y, T dif) {
    for (; x < ys.size(); x |= x + 1)
      ft[x].update(ind(x, y), dif);
  }
  T query(int x, int y) {
    T sum = 0;
    for (; x; x &= x - 1) sum += ft[x-1].query(ind(x-1, y))
        ↪;
    return sum;
  }
}; // hash-cpp-all = 4b694aa5311c5911b57f95cad9db49ca
```

MisofTree.h

Description: A simple treedata structure for inserting, erasing, and querying the n^{th} largest element.

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

15 lines

```
const int BITS = 15;
struct misof_tree{
    int cnt[BITS][1<<BITS];
    misof_tree() {memset(cnt, 0, sizeof cnt);}
    void add(int x, int dv) {
        for (int i = 0; i < BITS; cnt[i++][x] += dv, x >>=
            ↪1); }
    void del(int x, int dv) {
        for (int i = 0; i < BITS; cnt[i++][x] -= dv, x >>=
            ↪1); }
    int nth(int n) {
        int r = 0, i = BITS;
        while(i--) if (cnt[i][r <<= 1] <= n)
            n -= cnt[i][r], r |= 1;
        return r;
    }
}; // hash-cpp-all = 8c50f4c6f10e1ba44cd8a7679881cc1b
```

LineContainer.h

Description: Container where you can add lines of the form kx+m, and query maximum values at points x. Useful for dynamic programming ("convex hull trick").

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

29 lines

```
struct Line {
  mutable lint k, m, p;
  bool operator<(const Line& o) const { return k < o.k; }
  bool operator<(lint x) const { return p < x; }
};
struct LineContainer : multiset<Line, less<>> {
  // (for doubles, use inf = 1/.0, div(a,b) = a/b)
  const lint inf = LLONG_MAX;
  lint div(lint a, lint b) { // floored division
    return a / b - ((a ^ b) < 0 && a % b); }
  bool isect(iterator x, iterator y) {
    if (y == end()) { x->p = inf; return false; }
    if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
    else x->p = div(y->m - x->m, x->k - y->k);
    return x->p >= y->p;
  }
  void add(lint k, lint m) {
    auto z = insert({k, m, 0}), y = z++, x = y;
    while (isect(y, z)) z = erase(z);
    if (x != begin() && isect(--x, y)) isect(x, y = erase(y
      ↪));
    while ((y = x) != begin() && (--x)->p >= y->p)
      isect(x, erase(y));
  }
  lint query(lint x) {
    assert(!empty());
    auto l = *lower_bound(x);
    return l.k * x + l.m;
  }
}; // hash-cpp-all = a7bd5b8f8949a839ae0fe9c5bf830667
```

Matrix.h

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

28 lines

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

Wavelet.h

Description: Segment tree on values instead of indices. kth return the largest number in 0-indexed interval. count return the number of elements of a[i, j) that belong in [x, y].
Time: $\mathcal{O}(\log(n))$

25 lines

```
template<int SZ> struct Wavelet {
  vector<int> L[SZ], R[SZ];
  void build(vector<int> &a, int v=1, int l=0, int r=SZ-1)
    ↪{
    if (l == r) return;
    L[v] = R[v] = {0};
    vector<int> A[2]; int m = l + (r-l)/2;
    for(auto &t : a) {
      A[t>m].push_back(t);
      L[v].push_back(A[0].size()), R[v].push_back(A[1].size
        ↪());
    }
    build(A[0],2*v,l,m), build(A[1],2*v+1,m+1,r);
  }
  int kth(int i,int j,int k,int v=1,int l=0,int r=SZ-1) {
    ↪// [i, j)!!
    if (l == r) return l;
    int m = l + (r - l)/2, t = L[v][j]-L[v][i];
    if (t >= k) return kth(L[v][i], L[v][j],k,2*v,l,m);
    return kth(R[v][i], R[v][j],k-t,2*v+1,m+1,r);
  }
  int count(int i,int j,int x, int y, int v=1,int l=0,int r
    ↪=SZ-1) {
    if (y < l || r < x) return 0; //count(i, j, x, y)
      ↪retorna o numero de elementos
    if (x <= l && r <= y) return j - i; // de a[i, j) que
      ↪pertencem a [x, y]
    int m = l + (r - l)/2;
    return count(L[v][i], L[v][j], x, y, 2*v, l, m) + count
      ↪(i-L[v][i], j-L[v][j], x, y, 2*v+1, m+1, r);
  }
}; // hash-cpp-all = ccafbcdb45780a07a57206128b3d5c80
```

Numerical (4)

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 = gss(-1000,1000,func);`
Time: $\mathcal{O}(\log((b-a)/\epsilon))$

14 lines

```
double gss(double a, double b, double (*f)(double)) {
  double r = (sqrt(5)-1)/2, eps = 1e-7;
  double x1 = b - r*(b-a), x2 = a + r*(b-a);
  double f1 = f(x1), f2 = f(x2);
  while (b-a > eps)
    if (f1 < f2) { //change to > to find maximum
      b = x2; x2 = x1; f2 = f1;
      x1 = b - r*(b-a); f1 = f(x1);
    } else {
      a = x1; x1 = x2; f1 = f2;
      x2 = a + r*(b-a); f2 = f(x2);
    }
  return a;
} // hash-cpp-all = 31d45b514727a298955001a74bb9b9fa
```

Polynomial.h

17 lines

```
struct Poly {
  vector<double> a;
  double operator()(double x) const {
    double val = 0;
    for(int i = a.size(); i--;) (val *= x) += a[i];
    return val;
  }
  void diff() {
    for(int i = 1; i < a.size(); ++i) a[i-1] = i*a[i];
    a.pop_back();
  }
  void divroot(double x0) {
    double b = a.back(), c; a.back() = 0;
    for(int i = a.size()-1; i--;) c = a[i], a[i] = a[i+1]*
      ↪x0+b, b=c;
    a.pop_back();
  }
}; // hash-cpp-all = 84593c332febd5a0502a84570aa64c30
```

PolyRoots.h

Description: Finds the real roots to a polynomial.
Usage: `poly_roots({{2,-3,1}},-1e9,1e9) // solve x^2-3x+2 = 0`
Time: $\mathcal{O}\left(n^2 \log(1/\epsilon)\right)$

"Polynomial.h"

23 lines

```
vector<double> poly_roots(Poly p, double xmin, double xmax)
  ↪ {
  if ((p.a).size() == 2) { return {-p.a[0]/p.a[1]}; }
  vector<double> ret;
  Poly der = p;
  der.diff();
  auto dr = poly_roots(der, xmin, xmax);
  dr.push_back(xmin-1);
  dr.push_back(xmax+1);
  sort(dr.begin(), dr.end());
  for(int i = 0; i < dr.size()-1; ++i) {
    double l = dr[i], h = dr[i+1];
    bool sign = p(l) > 0;
    if (sign^(p(h) > 0)) {
      for(int it = 0; it < 60; ++it) { // while (h - l > 1e
        ↪-8)
        double m = (l + h) / 2, f = p(m);
        if ((f <= 0) ^ sign) l = m;
        else h = m;
      }
      ret.push_back((l + h) / 2);
    }
  }
  return ret;
} // hash-cpp-all = 49396af6a482b97394e6b2e412a6069c
```

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

13 lines

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

```
  double last = 0; temp[0] = 1;
  for(int k = 0; k < n; ++k) for(int i = 0; i < n; ++i) {
    res[i] += y[k] * temp[i];
    swap(last, temp[i]);
    temp[i] -= last * x[k];
  }
  return res;
} // hash-cpp-all = 97a266204931196ab2c1a2081e6f2f60
```

Lagrange.h

Description: Lagrange Polynomials.
Time: $\mathcal{O}(N)$

"ModPow.h", "ModInv.h", "Factorial.h" — 29 lines

```
template<typename T> struct Lagrange {
  const int n;
  vector<T> f, den;
  Lagrange(vector<T> other) : f(other), n(other.size()) {
    den.resize(n);
    for(int i = 0; i < n; ++i) {
      f[i] = (f[i] % mod + mod) % mod;
      den[i] = ifact[n-i-1] * ifact[i] % mod;
      if((n-i-1) % 2 == 1)
        den[i] = (mod - den[i]) % mod;
    }
  }
  T interpolate(T x) {
    x %= mod;
    vector<T> l, r;
    l.resize(n); r.resize(n);
    l[0] = r[n-1] = 1;
    for (int i = 1; i < n; ++i)
      l[i] = l[i-1] * (x - (i-1) + mod) % mod;
    for (int i = n-2; i >= 0; --i)
      r[i] = r[i+1] * (x - (i+1) + mod) % mod;
    T ans = 0;
    for (int i = 0; i < n; ++i) {
      T coef = l[i] * r[i] % mod;
      ans = (ans + coef * f[i] % mod * den[i]) % mod;
    }
    return ans;
  }
}; // hash-cpp-all = 31ad4afb396146045d8ab630178912af
```

BerlekampMassey.h

Description: Recovers any n-order linear recurrence relation from the first $2n$ terms of the recurrence. Useful for guessing linear recurrences after brute-forcing the first terms. Should work on any field, but numerical stability for floats is not guaranteed. Output will have size $\leq n$.
Usage: `BerlekampMassey({0, 1, 1, 3, 5, 11}) // {1, 2}`
Time: $\mathcal{O}(N^2)$

"ModularArithmetic.h" — 19 lines

```
template <typename num>
vector<num> BerlekampMassey(const vector<num>& s) {
  int n = int(s.size()), L = 0, m = 0;
  vector<num> C(n), B(n), T;
  C[0] = B[0] = 1;
  num b = 1;
  for(int i = 0; i < n; i++) { ++m;
    num d = s[i];
    for (int j = 1; j <= L; j++) d += C[j] * s[i - j];
    if (d == 0) continue;
    T = C; num coef = d / b;
    for (int j = m; j < n; j++) C[j] -= coef * B[j - m];
    if (2 * L > i) continue;
    L = i + 1 - L; B = T; b = d; m = 0;
  }
  C.resize(L + 1); C.erase(C.begin());
  for (auto& x : C) x = -x;
  return C;
} // hash-cpp-all = 4c4a480dc5edfaf7dec76cfa95895967
```

LinearRecurrence.h

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

"ModularArithmetic.h" — 22 lines

```
template <typename num>
num linearRec(const vector<num>& S, const vector<num>& tr,
    ↪lint k) {
  int n = int(tr.size());
  assert(S.size() >= tr.size());
  auto combine = [&](vector<num> a, vector<num> b) {
    vector<num> res(n * 2 + 1);
    for (int i = 0; i <= n; i++) for (int j = 0; j <= n; 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] * tr[j];
    res.resize(n + 1);
    return res;
  };
  vector<num> pol(n + 1), e(pol);
  pol[0] = e[1] = 1;
  for (++k; k; k /= 2) {
    if (k % 2) pol = combine(pol, e);
    e = combine(e, e);
  }
  num res = 0;
  for (int i = 0; i < n; i++) res += pol[i + 1] * S[i];
  return res;
} // hash-cpp-all = 0baa7bd4eee0281c35a994f18b73ddc2
```

HillClimbing.h

Description: Poor man's optimization for unimodal functions. — 14 lines

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

template<class F> pair<double, P> hillClimb(P start, F f) {
  pair<double, P> cur(f(start), start);
  for (double jmp = 1e9; jmp > 1e-20; jmp /= 2) {
    for(int j = 0; j < 100; ++j) for(int dx = -1; dx < 2;
      ↪++dx) for(int dy = -1; dy < 2; ++dy) {
      P p = cur.second;
      p[0] += dx*jmp;
      p[1] += dy*jmp;
      cur = min(cur, {f(p), p});
    }
  }
  return cur;
} // hash-cpp-all = 47a385e94ef7ce0c87e3c8de8882a81f
```

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. — 7 lines

```
template<class F>
double quad(double a, double b, F f, const int n = 1000) {
  double h = (b - a) / 2 / n, v = f(a) + f(b);
  for(int i = 1; i < n*2; ++i)
    v += f(a + i*h) * (i&1 ? 4 : 2);
  return v * h / 3;
} // hash-cpp-all = a89f8870779936ce986c40a84367af33
```

IntegrateAdaptive.h

Description: Fast integration using an adaptive Simpson's rule.
Usage: `double sphereVolume = quad(-1, 1, [](double x) { return quad(-1, 1, [&](double y) { return quad(-1, 1, [&](double z) { return x*x + y*y + z*z < 1; });});});` — 15 lines

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

template <class F>
d rec(F& f, d a, d b, d eps, d S) {
  d c = (a + b) / 2;
  d S1 = S(a, c), S2 = S(c, b), T = S1 + S2;
  if (abs(T - S) <= 15 * eps || b - a < 1e-10)
    return T + (T - S) / 15;
  return rec(f, a, c, eps / 2, S1) + rec(f, c, b, eps / 2,
    ↪S2);
}
template<class F>
d quad(d a, d b, F f, d eps = 1e-8) {
  return rec(f, a, b, eps, S(a, b));
} // hash-cpp-all = 5ce1b1578b88e19cb895f828aad45477
```

Determinant.h

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

```
double det(vector<vector<double>> &a) {
  int n = a.size(); double res = 1;
  for(int i = 0; i < n; ++i) {
    int b = i;
    for(int j = i+1; j < n; ++j) if (fabs(a[j][i]) > fabs(a
      ↪[b][i])) b = j;
    if (i != b) swap(a[i], a[b]), res *= -1;
    res *= a[i][i];
    if (res == 0) return 0;
    for(int j = i+1; j < n; ++j) {
      double v = a[j][i] / a[i][i];
      if (v != 0) for(int k = i+1; k < n; ++k) a[j][k] -= v
        ↪ * a[i][k];
    }
  }
  return res;
} // hash-cpp-all = 5906bc97b263956b316da1cff94cee0b
```

IntDeterminant.h

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

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

```
        ans *= -1;
      }
    }
    ans = ans * a[i][i] % mod;
    if (!ans) return 0;
  }
  return (ans + mod) % mod;
} // hash-cpp-all = 6ddd70c56d5503da62fc2a3b03ab8df3
```

Elimination.h

Description: Gauss-Jordan algorithm. Transform a matrix into its row echelon form. Returns a vector of pivots (for each variable) or -1 if free variable.

28 lines

```
vector<int> ToRowEchelon(vector<vector<double>> &M) {
    int cons = M.size(), vars = M[0].size() - 1;
    vector<int> pivot(vars, -1);
    int cur = 0;
    for (int var = 0; var < vars; ++var) {
        if (cur >= cons) continue;
        for (int con = cur + 1; con < cons; ++con)
            if(M[con][var] > M[cur][var])
                swap(M[con], M[cur]);
      if (abs(M[cur][var]) > kEps) {
            pivot[var] = cur;
            double aux = M[cur][var];
            for (int i = 0; i <= vars; ++i)
                M[cur][i] /= aux;
            for (int con = 0; con < cons; ++con) {
                if (con != cur) {
                    double mul = M[con][var];
                    for (int i = 0; i <= vars; ++i) {
                        M[con][i] -= mul * M[cur][i];
                    }
                    assert(M[con][var] < kEps);
                }
            }
            ++cur;
        }
    }
    return pivot;
} // hash-cpp-all = 9c481f76a17e885ed2f748ccfe41f37f
```

Math-Simplex.cpp

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

Time: $\mathcal{O}(NM * \#pivots)$, where a pivot may be e.g. an edge relaxation. $\mathcal{O}(2^n)$ in the general case. WARNING- segfaults on empty (size 0) max cx st Ax<=b, x>=0 do 2 phases; 1st check feasibility; 2nd check boundedness and ans

39 lines

```
vector<double> simplex(vector<vector<double>> A, vector<
    ↪double> b, vector<double> c) {
    int n = A.size(), m = A[0].size() + 1, r = n, s = m-1;
    vector<vector<double>> D = vector<vector<double>>(n+2,
        ↪vector<double>(m+1));
    vector<int> ix = vector<int>(n + m);
    for (int i = 0; i < n + m; ++i) ix[i] = i;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m-1; ++j) D[i][j] = -A[i][j];
        D[i][m - 1] = 1;
        D[i][m] = b[i];
        if (D[r][m] > D[i][m]) r = i;
    }
    for (int j = 0; j < m-1; ++j) D[n][j] = c[j];
    D[n + 1][m - 1] = -1; int z = 0;
    for (double d;;) {
        if (r < n) {
            swap(ix[s], ix[r + m]);
            D[r][s] = 1.0/D[r][s];
            for (int j = 0; j <= m; ++j) if (j != s) D[r][j
                ↪] *= -D[r][s];
            for (int i = 0; i <= n+1; ++i) if (i != r) {
                for (int j = 0; j <= m; ++j) if (j != s) D[
                    ↪i][j] += D[r][j] * D[i][s];
                D[i][s] *= D[r][s];
            }
        }
        r = -1; s = -1;
        for (int j = 0; j < m; ++j) if (s < 0 || ix[s] > ix
            ↪[j])
            if (D[n+1][j] > eps || D[n+1][j] > -eps && D[n
                ↪][j] > eps) s = j;
        if (s < 0) break;
        for (int i = 0; i < n; ++i) if (D[i][s] < -eps) {
            if (r < 0 || (d = D[r][m]/D[r][s]-D[i][m]/D[i][
                ↪s]) < -eps
                || d < eps && ix[r+m] > ix[i+m]) r = i;
        }
        if (r < 0) return vector<double>(); // unbounded
    }
    if (D[n+1][m] < -eps) return vector<double>(); //
        ↪infeasible
    vector<double> x(m-1);
    for (int i = m; i < n+m; ++i) if (ix[i] < m-1) x[ix[i]]
        ↪ = D[i-m][m];
    double result = D[n][m];
    return x; // ans: D[n][m]
} // hash-cpp-all = c3703ca3a20dfce2d4b7966e6dc8f74a
```

SolveLinear.h

Description: Solves $A * x = b$. If there are multiple solutions, an arbitrary one is returned. Returns rank, or -1 if no solutions. Data in A and b is lost.

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

36 lines

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

int solveLinear(vector<vd> &A, vd &b, vd &x) {
  int n = A.size(), m = x.size(), rank = 0, br, bc;
  if (n) assert(A[0].size() == m);
  vector<int> col(m); iota(col.begin(), col.end(), 0);
  for(int i = 0; i < n; ++i) {
    double v, bv = 0;
    for(int r = i; r < n; ++r) for(int c = i; c < m; ++c)
      if ((v = fabs(A[r][c])) > bv)
        br = r, bc = c, bv = v;
    if (bv <= eps) {
      for(int j = i; j < n; ++j) if (fabs(b[j]) > eps)
        ↪return -1;
      break;
    }
    swap(A[i], A[br]);
    swap(b[i], b[br]);
    swap(col[i], col[bc]);
    for(int j = 0; j < n; ++j) swap(A[j][i], A[j][bc]);
    bv = 1/A[i][i];
    for(int j = i+1; j < n; ++j) {
      double fac = A[j][i] * bv;
      b[j] -= fac * b[i];
      for(int k = i+1; k < m; ++k) A[j][k] -= fac*A[i][k];
    }
    rank++;
  }
  x.assign(m, 0);
  for (int i = rank; i--;) {
    b[i] /= A[i][i];
    x[col[i]] = b[i];
    for(int j = 0; j < i; ++j) b[j] -= A[j][i] * b[i];
  }
  return rank; // (multiple solutions if rank < m)
} // hash-cpp-all = 2654db9ae0ca64c0f3e32879d85e35d5
```

SolveLinear2.h

Description: To get all uniquely determined values of x back from SolveLinear, make the following changes:

"SolveLinear.h" 8 lines

```
for(int j = 0; j < n; ++j) if (j != i) // instead of for(
    ↪int j = i+1; j < n; ++j)
// ... then at the end:
x.assign(m, undefined);
for(int i = 0; i < rank; ++i) {
  for(int j = rank; j < m; ++j) if (fabs(A[i][j]) > eps)
    ↪goto fail;
  x[col[i]] = b[i] / A[i][i];
fail:; }
// hash-cpp-all = c8e85a5f8fc2c9ae6fc5672997b15cda
```

SolveLinearBinary.h

Description: Solves $Ax = b$ over $\mathbb{F}_2$. If there are multiple solutions, one is returned arbitrarily. Returns rank, or -1 if no solutions. Destroys A and b.

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

34 lines

```
typedef bitset<1000> bs;

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

  x = bs();
  for (int i = rank; i--;) {
    if (!b[i]) continue;
    x[col[i]] = 1;
    for(int j = 0; j < i; ++j) b[j] ^= A[j][i];
  }
  return rank; // (multiple solutions if rank < m)
} // hash-cpp-all = 71d8713aa9eab9f9d77a9e46d9caed1f
```

MatrixInverse.h

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

32 lines

```
int matInv(vector<vector<double>>& A) {
  int n = A.size(); vector<int> col(n);
  vector<vector<double>> tmp(n, vector<double>(n));
  for(int i = 0; i < n; ++i) tmp[i][i] = 1, col[i] = i;
  for(int i = 0; i < n; ++i) { // hash-cpp-1
    int r = i, c = i;
    for(int j = i; j < n; ++j) for(int k = i; k < n; ++k)
      if (fabs(A[j][k]) > fabs(A[r][c]))
        r = j, c = k;
    if (fabs(A[r][c]) < 1e-12) return i;
    A[i].swap(A[r]); tmp[i].swap(tmp[r]);
    for(int j = 0; j < n; ++j)
      swap(A[j][i], A[j][c]), swap(tmp[j][i], tmp[j][c]);
    swap(col[i], col[c]);
    double v = A[i][i];
    for(int j = i+1; j < n; ++j) {
      double f = A[j][i] / v;
      A[j][i] = 0;
      for(int k = i+1; k < n; ++k) A[j][k] -= f*A[i][k];
      for(int k = 0; k < n; ++k) tmp[j][k] -= f*tmp[i][k];
    }
    for(int j = i+1; j < n; ++j) A[i][j] /= v;
    for(int j = 0; j < n; ++j) tmp[i][j] /= v;
    A[i][i] = 1;
  } // hash-cpp-1 = b5c37a0147222a30250f8eb364b7dd25
  for (int i = n-1; i > 0; --i) for(int j = 0; j < i; ++j)
    {{ // hash-cpp-2
    double v = A[j][i];
    for(int k = 0; k < n; ++k) tmp[j][k] -= v*tmp[i][k];
  }
  for(int i = 0; i < n; ++i) for(int j = 0; j < n; ++j) A[
    col[i]][col[j]] = tmp[i][j];
  return n;
} // hash-cpp-2 = cb1e282dd60fc93e07018380693a681b
```

MatrixInverse-mod.h

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

"../number-theory/ModPow.h" 36 lines

```
int matInv(vector<vector<lint>>& A) {
  int n = A.size(); vi col(n);
  vector<vector<lint>> tmp(n, vector<lint>(n));
  for(int i = 0; i < n; ++i) tmp[i][i] = 1, col[i] = i;

  for(int i = 0; i < n; ++i) { // hash-cpp-1
    int r = i, c = i;
    for(int j = i; j < n; ++j) for(int k = i; k < n; ++k)
      if (A[j][k]) {
      r = j; c = k; goto found;
    }
    return i;
found:
    A[i].swap(A[r]); tmp[i].swap(tmp[r]);
    for(int j = 0; j < n; ++j) swap(A[j][i], A[j][c]), swap
      (tmp[j][i], tmp[j][c]);
    swap(col[i], col[c]);
    lint v = modpow(A[i][i], mod - 2);
    for(int i = i+1; i < n; ++i) {
      lint f = A[j][i] * v % mod;
      A[j][i] = 0;
      for(int k = i+1; k < n; ++k) A[j][k] = (A[j][k] - f*A
        [i][k]) % mod;
      for(int k = 0; k < n; ++k) tmp[j][k] = (tmp[j][k] - f
        *tmp[i][k]) % mod;
    }
    for(int j = i+1; j < n; ++j) A[i][j] = A[i][j] * v %
      mod;
    for(int j = 0; j < n; ++j) tmp[i][j] = tmp[i][j] * v %
      mod;
    A[i][i] = 1;
  } // hash-cpp-1 = 13e962046f8aa0d20f52e52c01324996

  for (int i = n-1; i > 0; --i) for(int j = 0; j < i; ++j)
    {{ // hash-cpp-2
    lint v = A[j][i];
    for(int k = 0; k < n; ++k) tmp[j][k] = (tmp[j][k] - v*
      tmp[i][k]) % mod;
  }

  for(int i = 0; i < n; ++i) for(int j = 0; j < n; ++j)
    A[col[i]][col[j]] = tmp[i][j] % mod + (tmp[i][j] < 0 ?
      mod : 0);
  return n;
} // hash-cpp-2 = ffc918f768d7f5d56294f45d9006a137
```

Tridiagonal.h

Description: $x = \text{tridiagonal}(d, p, q, b)$ solves the equation system

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

This is useful for solving problems on the type

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

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

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

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

26 lines

```
typedef double T;
vector<T> tridiagonal(vector<T> diag, const vector<T> &
  super,
  const vector<T> &sub, vector<T> b) {
  int n = b.size(); vector<int> tr(n);
  for(int i = 0; i < n-1; ++i) {
    if (abs(diag[i]) < 1e-9 * abs(super[i])) { // diag[i]
      == 0
      b[i+1] -= b[i] * diag[i+1] / super[i];
      if (i+2 < n) b[i+2] -= b[i] * sub[i+1] / super[i];
      diag[i+1] = sub[i]; tr[++i] = 1;
    } else {
      diag[i+1] -= super[i]*sub[i]/diag[i];
      b[i+1] -= b[i]*sub[i]/diag[i];
    }
  }
  for (int i = n; i--;) {
    if (tr[i]) {
      swap(b[i], b[i-1]);
      diag[i-1] = diag[i];
      b[i] /= super[i-1];
    } else {
      b[i] /= diag[i];
      if (i) b[i-1] -= b[i]*super[i-1];
    }
  }
  return b;
} // hash-cpp-all = d0855fb63594fa47d372bf1a8c3078f9
```

NewtonMethod.h

Description: Root find method

12 lines

```
double f(double x) { return (x*x) - 4; }
double df(double x) { return 2*x; }
double root(double x0) {
    const double eps = 1e-15;
    double x = x0;
    while (1) {
        double nx = x - (f(x)/df(x));
        if (abs(x - nx) < eps) break;
        x = nx;
    }
    return x;
} // hash-cpp-all = a8a5ba0f7c2829b1bbca94088c1811a3
```

NewtonSQRT.h

Description: Square root find method

21 lines

```
double sqrt_newton(double n) {
    const double eps = 1E-15;
    double x = 1;
    while (1) {
        double nx = (x + n / x) / 2;
        if (abs(x - nx) < eps) break;
        x = nx;
    }
    return x;
}
int isqrt_newton(int n) {
    int x = 1;
    bool decreased = false;
    while (1) {
        int nx = (x + n / x) >> 1;
        if (x == nx || nx > x && decreased) break;
        decreased = nx < x;
        x = nx;
    }
    return x;
} // hash-cpp-all = 25ff801e8601f802f249e0b08ecdf4bf
```

Polyominoes.h

Description: Generate all free polyominoes with n squares. Takes less than a sec if $n < 10$, around 2s if $n = 10$ and around 6s if $n = 11$.

34 lines

```
using pii = pair<int,int>;
vector<int> diri = {0, 1, 0, -1};
vector<int> dirj = {1, 0, -1, 0};
vector<vector<pii>> poly[LIM];
void generate(int n){
```

```
  poly[1] = {{{0 , 0}}};
  for(int i = 2 ; i <= n; i++) {
    set<vector<pii>> cur_om;
    for(auto &om : poly[i-1]) {
      pii mini = om[0];
      for(auto &p : om)
        for(int d = 0; d < 4; d++) {
          int x = p.st + diri[d], y = p.nd + dirj[d];
          if(!binary_search(om.begin(), om.end(), pii(x,y))
             ↪) {
            pii m = min(mini, {x, y});
            pii new_cell(x - m.st, y - m.nd);
            bool new_in = false;
            vector<pii> norm;
            for(pii &pn : om) {
              pii cur(pn.st - m.st, pn.nd - m.nd);
              if(cur > new_cell && !new_in) {
                new_in = true;
                norm.push_back(new_cell);
              }
              norm.push_back(cur);
            }
            if(!new_in) norm.push_back(new_cell);
            if(!cur_om.count(norm)) cur_om.insert(norm);
          }
        }
    }
    poly[i].assign(cur_om.begin(),cur_om.end());
  }
} // hash-cpp-all = d63122fd7f9fc7b8159ff93814ef654b
```

4.1 Fourier transforms

FastFourierTransform.h

Description: fft(a) computes $\hat{f}(k) = \sum_x a[x] \exp(2\pi i \cdot kx/N)$ for all k. Useful for convolution: `conv(a, b) = c`, where $c[x] = \sum a[i]b[x-i]$. For convolution of complex numbers or more than two vectors: FFT, multiply pointwise, divide by n, reverse(start+1, end), FFT back. Rounding is safe if $(\sum a_i^2 + \sum b_i^2) \log_2 N < 9 \cdot 10^{14}$ (in practice 10^{16}; higher for random inputs). Otherwise, use long doubles/NTT/FFTMod.
Time: $\mathcal{O}(N \log N)$ with $N = |A| + |B|$ ($\sim 1s$ for $N = 2^{22}$)

35 lines

```
using doublex = complex<long double>;
struct FFT {
    vector<doublex> fft(vector<doublex> y, bool invert =
       ↪false) {
        const int N = y.size(); assert(N == (N&-N));
        vector<lint> rev(N);
        for (int i = 1; i < N; ++i) {
            rev[i] = (rev[i>>1]>>1) | (i&1 ? N>>1 : 0);
            if (rev[i] < i) swap(y[i], y[rev[i]]);
        }
        vector<doublex> rootni(N/2);
        for (lint n = 2; n <= N; n *= 2) {
            const doublex rootn = polar(1.0, (invert ? +1.0
               ↪ : -1.0) * 2.0*acos(-1.0)/n);
            rootni[0] = 1.0;
            for (lint i = 1; i < n/2; ++i) rootni[i] =
               ↪rootni[i-1] * rootn;
            for (lint left = 0; left != N; left += n) {
                const lint mid = left + n/2;
                for (lint i = 0; i < n/2; ++i) {
                    const doublex temp = rootni[i] * y[mid
                       ↪+ i];
                    y[mid + i] = y[left + i] - temp; y[left
                       ↪ + i] += temp;
                }
            }
        } if (invert) for (auto &v : y) v /= (doublex)N;
        return move(y);
    }
    uint nextpow2(uint v) { return v ? 1 << __lg(2*v-1) :
       ↪1; }
    vector<doublex> convolution(vector<doublex> a, vector<
       ↪doublex> b) {
        const lint n = max((int)a.size()+(int)b.size()-1,
           ↪0), n2 = nextpow2(n);
        a.resize(n2); b.resize(n2);
        vector<doublex> fa = fft(move(a)), fb = fft(move(b)
           ↪), &fc = fa;
        for (lint i = 0; i < n2; ++i) fc[i] = fc[i] * fb[i
           ↪];
        vector<doublex> c = fft(move(fc), true);
        c.resize(n);
        return move(c);
    }
}; // hash-cpp-all = b9be027786dfbeb9ccea1f049b9fdd07
```

FastFourierTransformMod.h

Description: Higher precision FFT, can be used for convolutions modulo arbitrary integers as long as $N \log_2 N \cdot \text{mod} < 8.6 \cdot 10^{14}$ (in practice 10^{16} or higher). Inputs must be in $[0, \text{mod})$.
Time: $\mathcal{O}(N \log N)$, where $N = |A| + |B|$ (twice as slow as NTT or FFT)

85 lines

```
typedef unsigned int uint;
typedef long double ldouble;

template<typename T, typename U, typename B> struct
   ↪ModularFFT {
    inline T ifmod(U v, T mod) { return v >= (U)mod ? v -
       ↪mod : v; }
    T pow(T x, U y, T p) {
        T ret = 1, x2p = x;
        while (y) {
            if (y % 2) ret = (B)ret * x2p % p;
            y /= 2; x2p = (B)x2p * x2p % p;
        }
        return ret;
    }
    vector<T> fft(vector<T> y, T mod, T gen, bool invert =
       ↪false) {
        int N = y.size(); assert(N == (N&-N));
        if (N == 0) return move(y);
        vector<int> rev(N);
        for (int i = 1; i < N; ++i) {
            rev[i] = (rev[i>>1]>>1) | (i&1 ? N>>1 : 0);
            if (rev[i] < i) swap(y[i], y[rev[i]]);
        }
        assert((mod-1)%N == 0);
        T rootN = pow(gen, (mod-1)/N, mod);
        if (invert) rootN = pow(rootN, mod-2, mod);
        vector<T> rootni(N/2);
        for (int n = 2; n <= N; n *= 2) {
            T rootn = pow(rootN, N/n, mod);
            rootni[0] = 1;
            for (int i = 1; i < n/2; ++i) rootni[i] = (B)
               ↪rootni[i-1] * rootn % mod;
            for (int left = 0; left != N; left += n) {
                int mid = left + n/2;
                for (int i = 0; i < n/2; ++i) {
                    T temp = (B)rootni[i] * y[mid+i] % mod;
                    y[mid+i] = ifmod((U)y[left+i] + mod -
                       ↪temp, mod);
                    y[left+i] = ifmod((U)y[left+i] + temp,
                       ↪mod);
                }
            }
        }
        if (invert) {
            T invN = pow(N, mod-2, mod);
            for (T &v : y) v = (B)v * invN % mod;
        }
        return move(y);
    }
    vector<T> convolution(vector<T> a, vector<T> b, T mod,
       ↪T gen) {
        int N = a.size()+b.size()-1, N2 = nextpow2(N);
        a.resize(N2); b.resize(N2);
        vector<T> fa = fft(move(a), mod, gen), fb = fft(
           ↪move(b), mod, gen), &fc = fa;
        for (int i = 0; i < N2; ++i) fc[i] = (B)fc[i] * fb[
           ↪i] % mod;
        vector<T> c = fft(move(fc), mod, gen, true);
        c.resize(N); return move(c);
    }
    vector<T> self_convolution(vector<T> a, T mod, T gen) {
        int N = 2*a.size()-1, N2 = nextpow2(N);
        a.resize(N2);
        vector<T> fc = fft(move(a), mod, gen);
        for (int i = 0; i < N2; ++i) fc[i] = (B)fc[i] * fc[
           ↪i] % mod;
        vector<T> c = fft(move(fc), mod, gen, true);
        c.resize(N); return move(c);
    }
    uint nextpow2(uint v) { return v ? 1 << __lg(2*v-1) :
       ↪1; }
};

const int mod = 998244353, mod_gen = 3;

vector<int> convolute(const vector<int> &a, const vector<
   ↪int> &b) {
    if (a.empty() || b.empty()) return {};
    ModularFFT<int, uint, lint> modular_fft;
    return modular_fft.convolution(a, b, mod, mod_gen);
}

vector<int> convolute_all(const vector<vector<int>> &polys,
   ↪ int begin,
                          int end) {
    if (end - begin == 0) return {1};
    else if (end - begin == 1) return polys[begin];
    else {
        int mid = begin + (end - begin) / 2;
        return convolute(convolute_all(polys, begin, mid),
                         convolute_all(polys, mid, end));
    }
}

vector<int> convolute_all(const vector<vector<int>> &polys)
   ↪ {
  return convolute_all(polys, 0, (int)polys.size());
} // hash-cpp-all = dbcea8e7b4d75d1274c7284d384a5a64
```

NumberTheoreticTransform.h

Description: 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. Inputs must be in $[0, \text{mod})$.
Time: $\mathcal{O}(N \log N)$

"../number-theory/modpow.h" 32 lines

```
const lint mod = (119 << 23) + 1, root = 62; // = 998244353
// For p < 2^30 there is also e.g. 5 << 25, 7 << 26, 479 <<
    ↪ 21
// and 483 << 21 (same root). The last two are > 10^9.

typedef vector<lint> vl;
void ntt(vl& a, vl& rt, vl& rev, int n) {
  for(int i = 0; i < n; ++i) if (i < rev[i]) swap(a[i], a[
     ↪rev[i]]);
  for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k) for(int j = 0; j < k
       ↪; ++j) {
      lint z = rt[j + k] * a[i + j + k] % mod, &ai = a[i
         ↪+ j];
      a[i + j + k] = (z > ai ? ai - z + mod : ai - z);
      ai += (ai + z >= mod ? z - mod : z);
  }
}

vl conv(const vl& a, const vl& b) {
  if (a.empty() || b.empty())
    return {};
  int s = a.size()+b.size()-1, B = 32 - __builtin_clz(s), n
     ↪ = 1 << B;
  vl L(a), R(b), out(n), rt(n, 1), rev(n);
  L.resize(n), R.resize(n);
  for(int i = 0; i < n; ++i) rev[i] = (rev[i / 2] | (i & 1)
     ↪ << B) / 2;
  lint curL = mod / 2, inv = modpow(n, mod - 2);
  for (int k = 2; k < n; k *= 2) {
    lint z[] = {1, modpow(root, curL /= 2)};
    for(int i = k; i < 2*k; ++i) rt[i] = rt[i / 2] * z[i &
       ↪1] % mod;
  }
  ntt(L, rt, rev, n); ntt(R, rt, rev, n);
  for(int i = 0; i <n; ++i) out[-i & (n-1)] = L[i] * R[i] %
     ↪ mod * inv % mod;
  ntt(out, rt, rev, n);
  return {out.begin(), out.begin() + s};
} // hash-cpp-all = 1f6be88c85faaf9505586299f0b01d29
```

FastSubsetTransform.h

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

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

16 lines

```
void FST(vector<int> &a, bool inv) { // hash-cpp-1
  for (int n = a.size(), 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 = a[j], &v = a[j + step]; tie(u, v) =
        inv ? pii(v - u, u) : pii(v, u + v); // AND
        inv ? pii(v, u - v) : pii(u + v, u); // OR
        pii(u + v, u - v);                   // XOR
    }
  }
  if (inv) for(auto &x : a) x /= a.size(); // XOR only
} // hash-cpp-1 = a4980de468052607447174d1308c276b
vector<int> conv(vector<int> a, vector<int> b) { // hash-
   ↪cpp-2
  FST(a, 0); FST(b, 0);
  for(int i = 0; i < a.size(); ++i) a[i] *= b[i];
  FST(a, 1); return a;
} // hash-cpp-2 = 733c60843e71a1333215a8d28f020966
```

4.1.1 Duality

max $c^T x$ sjt to $Ax \leq b$. Dual problem is min $b^T x$ sjt to $A^T x \geq c$. By strong duality, min max value coincides.

4.1.2 Generating functions

A list of generating functions for useful sequences:

$(1, 1, 1, 1, 1, 1, \ldots)$	$\frac{1}{1-z}$
$(1, -1, 1, -1, 1, -1, \ldots)$	$\frac{1}{1+z}$
$(1, 0, 1, 0, 1, 0, \ldots)$	$\frac{1}{1-z^2}$
$(1, 0, \ldots, 0, 1, 0, 1, 0, \ldots, 0, 1, 0, \ldots)$	$\frac{1}{1-z^2}$
$(1, 2, 3, 4, 5, 6, \ldots)$	$\frac{1}{(1-z)^2}$
$(1, \binom{m+1}{m}, \binom{m+2}{m}, \binom{m+3}{m}, \ldots)$	$\frac{1}{(1-z)^{m+1}}$
$(1, c, \binom{c+1}{2}, \binom{c+2}{3}, \ldots)$	$\frac{1}{(1-z)^c}$
$(1, c, c^2, c^3, \ldots)$	$\frac{1}{1-cz}$
$(0, 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots)$	$\ln \frac{1}{1-z}$

A neat manipulation trick is:

$$\frac{1}{1-z} G(z) = \sum_n \sum_{k \leq n} g_k z^n$$

4.1.3 Polyominoes

How many free (rotation, reflection), one-sided (rotation) and fixed n-ominoes are there?

n	3	4	5	6	7	8	9	10
free	2	5	12	35	108	369	1.285	4.655
one-sided	2	7	18	60	196	704	2.500	9.189
fixed	6	19	63	216	760	2.725	9.910	36.446

Number theory (5)

5.1 Modular arithmetic

ModularArithmetic.h

Description: Operators for modular arithmetic. You need to set mod to some number first and then you can use the structure.

57 lines

```
template <int MOD_>  struct modnum {
private:
  using lint = long long;
  lint v;
  static int modinv(int a, int m) {
    a %= m;
    assert(a);
    return a == 1 ? 1 : int(m - lint(modinv(m, a)) * lint(m
       ↪) / a);
  }
public:
  static constexpr int MOD = MOD_;
  modnum() : v(0) {}
  modnum(lint v_) : v(int(v_ % MOD)) { if (v < 0) v += MOD;
     ↪ }
  explicit operator int() const { return v; }
  friend std::ostream &operator<<(std::ostream& out, const
     ↪modnum& n) { return out << int(n); }
  friend std::istream &operator>>(std::istream& in, modnum&
     ↪ n) { lint v_; in >> v_; n = modnum(v_); return in;
     ↪}
  friend bool operator==(const modnum& a, const modnum& b)
     ↪{ return a.v == b.v; }
  friend bool operator!=(const modnum& a, const modnum& b)
     ↪{ return a.v != b.v; }
  modnum inv() const {
    modnum res;
    res.v = modinv(v, MOD);
    return res;
  }
  modnum neg() const {
    modnum res;
    res.v = v ? MOD-v : 0;
    return res;
  }
  modnum operator-() const { return neg(); }
  modnum operator+() const { return modnum(*this); }
  modnum& operator+=(const modnum& o) {
    v += o.v;
    if (v >= MOD) v -= MOD;
    return *this;
  }
  modnum& operator-=(const modnum& o) {
    v -= o.v;
    if (v < 0) v += MOD;
    return *this;
  }
  modnum& operator*=(const modnum& o) {
    v = int(lint(v) * lint(o.v) % MOD);
    return *this;
  }
  modnum& operator/=(const modnum& o) { return *this *= o.
     ↪inv(); }
  friend modnum operator+(const modnum& a, const modnum& b)
     ↪ { return modnum(a) += b; }
  friend modnum operator-(const modnum& a, const modnum& b)
     ↪ { return modnum(a) -= b; }
  friend modnum operator*(const modnum& a, const modnum& b)
     ↪ { return modnum(a) *= b; }
  friend modnum operator/(const modnum& a, const modnum& b)
     ↪ { return modnum(a) /= b; }
};
template <typename T> T pow(T a, lint b) {
  assert(b >= 0);
  T r = 1; while (b) { if (b & 1) r *= a; b >>= 1; a *= a;
     ↪} return r;
}
using num = modnum<int(1e9)+7>;

// hash-cpp-all = 56537454341667fa4f164b1506b9ed93
```

PairNumTemplate.h

Description: Support pairs operations using modnum template. Pretty good for string hashing.

43 lines

```
template <typename T, typename U> struct pairnum {
  T t; U u;
  pairnum() : t(0), u(0) {}
  pairnum(long long v) : t(v), u(v) {}
```

```
  pairnum(const T& t_, const U& u_) : t(t_), u(u_) {}
  friend std::ostream& operator << (std::ostream& out,
    ↪const pairnum& n) { return out << '(' << n.t << ','
    ↪<< ' ' << n.u << ')'; }
  friend std::istream& operator >> (std::istream& in,
    ↪pairnum& n) { long long v; in >> v; n = pairnum(v);
    ↪return in; }
  friend bool operator == (const pairnum& a, const pairnum&
    ↪ b) { return a.t == b.t && a.u == b.u; }
  friend bool operator != (const pairnum& a, const pairnum&
    ↪ b) { return a.t != b.t || a.u != b.u; }
  pairnum inv() const {
    return pairnum(t.inv(), u.inv());
  }
  pairnum neg() const {
    return pairnum(t.neg(), u.neg());
  }
  pairnum operator- () const {
    return pairnum(-t, -u);
  }
  pairnum operator+ () const {
    return pairnum(+t, +u);
  }
  pairnum& operator += (const pairnum& o) {
    t += o.t; u += o.u;
    return *this;
  }
  pairnum& operator -= (const pairnum& o) {
    t -= o.t; u -= o.u;
    return *this;
  }
  pairnum& operator *= (const pairnum& o) {
    t *= o.t; u *= o.u;
    return *this;
  }
  pairnum& operator /= (const pairnum& o) {
    t /= o.t; u /= o.u;
    return *this;
  }
  friend pairnum operator + (const pairnum& a, const
    ↪pairnum& b) { return pairnum(a) += b; }
  friend pairnum operator - (const pairnum& a, const
    ↪pairnum& b) { return pairnum(a) -= b; }
  friend pairnum operator * (const pairnum& a, const
    ↪pairnum& b) { return pairnum(a) *= b; }
  friend pairnum operator / (const pairnum& a, const
    ↪pairnum& b) { return pairnum(a) /= b; }
};
// hash-cpp-all = 229a89dc1bd3c18584636921c098ebdc
```

ModInv.h

Description: Find x such that $ax \equiv 1 (\text{mod } m)$. The inverse only exist if a and m are coprimes.

16 lines

```
template<typename T>
T modinv(T a, T m) {
  assert(m > 0);
  if (m == 1) return 0;
  a %= m;
  if (a < 0) a += m;
  assert(a != 0);
  if (a == 1) return 1;
  return m - modinv(m, a) * m/a;
}
// Iff mod is prime
lint modinv(lint a) { return modpow(a % mod, mod-2); }
// const lint mod = 1000000007, LIM = 200000;
lint* inv = new lint[LIM] - 1; inv[1] = 1;
for(int i = 2; i < LIM; ++i) inv[i] = mod - (mod/i) * inv[
  ↪mod%i] % mod;
// hash-cpp-all = 00205250d7354b57c72ec90fcf5947ff
```

Modpow.h

12 lines

```
lint modpow(lint a, lint e){
    if(e == 0) return 1;
    if(e & 1) return (a*modpow(a,e-1)) % mod;
    lint c = modpow(a, e>>1);
    return (c*c) % mod;
}
lint modpow(lint b, lint e) {
  lint ret = 1;
  for (int i = 1; i <= e; i *= 2, b = b * b % mod)
    if (i & e) ret = ret * b % mod;
  return ret;
} // hash-cpp-all = ba3b4f88d538ef63e2d071661d1e3106
```

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.

17 lines

```
typedef unsigned long long ull;
ull sumsq(ull to) { return to / 2 * ((to-1) | 1); }
ull divsum(ull to, ull c, ull k, ull m) {
  ull res = k / m * sumsq(to) + c / m * to;
  k %= m; c %= m;
  if (k) {
    ull to2 = (to * k + c) / m;
    res += to * to2;
    res -= divsum(to2, m-1 - c, m, k) + to2;
  }
  return res;
}
lint modsum(ull to, lint c, lint k, lint m) {
  c = ((c % m) + m) % m;
  k = ((k % m) + m) % m;
  return to * c + k * sumsq(to) - m * divsum(to, c, k, m);
} // hash-cpp-all = decfb822377b09e4ce538b62e72ff625
```

ModMul.cpp

Description: Modular multiplication operation

10 lines

```
lint modMul(lint a, lint b){
    lint ret = 0;
    a %= mod;
    while (b){
        if (b & 1) ret = (ret + a) % mod;
        a = (2 * a) % mod;
        b >>= 1;
    }
    return ret;
} // hash-cpp-all = f741d07bbdfa19949a4d645f2c519ecd
```

ModMulLL.h

Description: Calculate $a \cdot b \bmod c$ (or $a^b \bmod c$) for $0 \le a, b < c < 2^{63}$.
Time: $\mathcal{O}(1)$ for mod_mul, $\mathcal{O}(\log b)$ for mod_pow

12 lines

```
typedef unsigned long long ull;
typedef long double ld;
ull mod_mul(ull a, ull b, ull M) {
  lint ret = a * b - M * ull(ld(a) * ld(b) / ld(M));
  return ret + M * (ret < 0) - M * (ret >= (lint)M);
}
ull mod_pow(ull b, ull e, ull mod) {
  ull ans = 1;
  for (; e; b = mod_mul(b, b, mod), e >>= 1)
    if (e & 1) ans = mod_mul(ans, b, mod);
  return ans;
} // hash-cpp-all = 6ecbeac391f4533c348906f0d41e9ede
```

ModSqrt.h

Description: Tonelli-Shanks algorithm for modular square roots. Finds x s.t. $x^2 = a \pmod p$ ($-x$ gives the other solution).
Time: $\mathcal{O}\left(\log^2 p\right)$ worst case, $\mathcal{O}(\log p)$ for most p

"ModPow.h" 23 lines

```
lint sqrt(lint a, lint p) {
  a %= p; if (a < 0) a += p;
  if (a == 0) return 0;
  assert(modpow(a, (p-1)/2, p) == 1); // else no solution
  if (p % 4 == 3) return modpow(a, (p+1)/4, p);
  // a^(n+3)/8 or 2^(n+3)/8 * 2^(n-1)/4 works if p % 8 == 5
  lint s = p - 1, n = 2;
  int r = 0, m;
  while (s % 2 == 0) ++r, s /= 2;
  while (modpow(n, (p - 1) / 2, p) != p - 1) ++n;
  lint x = modpow(a, (s + 1) / 2, p);
  lint b = modpow(a, s, p), g = modpow(n, s, p);
  for (;; r = m) {
    lint t = b;
    for (m = 0; m < r && t != 1; ++m)
      t = t * t % p;
    if (m == 0) return x;
    lint gs = modpow(g, 1LL << (r - m - 1), p);
    g = gs * gs % p;
    x = x * gs % p;
    b = b * g % p;
  }
} // hash-cpp-all = 09107ec523e930fa8ba3787d3729bde9
```

MulOrder.h

Description: Find the smallest integer k such that $a^k \pmod m = 1$. $0 < k < m$.

8 lines

```
int mulOrder(int x, int y){
    if (__gcd(x, y) != 1) return 0;
    lint p = phi(y);
    auto k = factorize(x);
    for (auto &t : k)
        while(p % t.first == 0 && modpow(x, p/t.first, p)
          ↪== 1) p /= t.first;
    return P;
} // hash-cpp-all = 363fd1325a5d3d6c75a391490df50230
```

Quadratic.h

Description: Solve $x^2 \equiv n \mod p (0 \le a < p)$ where p is prime in $O(\log p)$. If $p > n$, factorize p and solve each of $x^2 \equiv n \mod p_i \forall i$.

20 lines

```
void mul(lint &a1, lint &b1, lint a2, lint b2, lint w, lint
  ↪ p) {
  lint t1 = (a1*a2 + b1*b2 % p*w), t2 = (a1*b2 + a2*b1);
  a1 = t1 % p, b1 = t2 % p;
}
int Pow(lint a, lint w, lint b, lint p) {
  lint res1=1, res2=0, c1=a, c2=1;
  for (;b;b>>=1) { if (b&1) mul(res1,res2,c1,c2,w,p); mul(
    ↪c1,c2,c1,c2,w,p); }
  return res1;
}
```

```
int quadratic(lint n, int p) {
  lint a, r = 0; n %= p;
  if (p == 2) return -1;
  if (n == 0) return 0;
  if (modpow(n, p/2, p) != 1) return -1;
  do a = rng() % (p-1)+1; while((modpow(a*a-n, p/2, p)-1) %
    ↪ p == 0);
  r = Pow(a, (a*a-n) % p, (p+1)/2, p);
  if (r < 0) r += p;
  assert((r*r-n) % p == 0);
  return r;
} // hash-cpp-all = 2d1b004f3363428e35bc25c000484ae9
```

5.2 Primality

Sieve.h

Description: Prime sieve for generating all primes up to a certain limit. isprime[i] is true iff i is a primes. Also useful if you need to compute any multiplicative function (in this case Moebius..). 15 lines

```
vector<int> lp, primes, mu;
void run_sieve(int n) {
  lp.resize(n); primes.resize(n); mu.resize(n);
  mu[1] = 1;
  int i, j, tot, t1;
  for (i = 1; i <= n; ++i) lp[i] = i;
  for (i = 2, tot = 0; i <= n; ++i){
    if (lp[i] == i) primes[++tot] = i;
    for (j = 1; j <= tot && (t1 = primes[j] * i) <= n; ++j)
      ↪ {
      lp[t1] = primes[j];
      mu[t1] *= mu[i];
      if (i % primes[j] == 0) break;
    }
  }
} // hash-cpp-all = 1a595267a9e22d1e857153b2af267523
```

Mobius.h

Description: If g and f are arithmetic functions . Return 0 if divisible by any perfect square, 1 if has an even quantity of prime numbers and -1 if has an odd quantity of primes.
Time: $\mathcal{O}(sqrt(n))$ 10 lines

```
template<typename T> T mobius(T n) {
    T p = 0, aux = n;
    for (int i = 2; i*i <= n; ++i)
        if (n % i == 0) {
            n /= i;
            p += 1;
            if (n % i == 0) return 0;
        }
    return (p&1 ? 1 : -1);
} // hash-cpp-all = c2cf445d5148aab42f5f697c3d61f4bb
```

MillerRabin.h

Description: Deterministic Miller-Rabin primality test. Guaranteed to work for numbers up to 2^{64}; for larger numbers, extend A randomly.
Time: 7 times the complexity of $a^b \mod c$.

"ModMulLL.h" 12 lines

```
bool isPrime(ull n) {
  if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
  vector<ull> A = {2, 325, 9375, 28178, 450775, 9780504,
    ↪1795265022};
  ull s = __builtin_ctzll(n-1), d = n >> s;
  for(ull a : A) {   // ^ count trailing zeroes
    ull p = mod_pow(a%n, d, n), i = s;
    while (p != 1 && p != n - 1 && a % n && i--)
      p = mod_mul(p, p, n);
    if (p != n-1 && i != s) return 0;
  }
  return 1;
} // hash-cpp-all = 8458d13990d1b39ac5d53f1b1bbbea39
```

Factorize.h

Description: Get all factors of n. 17 lines

```
vector<pair<int, int>> factorize(int value) {
    vector<pair<int, int>> result;
    for (int p = 2; p*p <= value; ++p)
        if (value % p == 0) {
            int exp = 0;
            while (value % p == 0) {
                value /= p;
                ++exp;
            }
            result.emplace_back(p, exp);
        }
    if (value != 1) {
        result.emplace_back(value, 1);
        value = 1;
    }
    return result;
} // hash-cpp-all = 46ea351907e7fba012d0082844c7c198
```

PollardRho.h

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

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

```
ull pollard(ull n) {
  auto f = [n](ull x) { return mod_mul(x, x, n) + 1; };
  ull x = 0, y = 0, t = 0, prd = 2, i = 1, q;
  while (t++ % 40 || __gcd(prd, n) == 1) {
    if (x == y) x = ++i, y = f(x);
    if ((q = mod_mul(prd, max(x,y) - min(x,y), n))) prd = q
      ↪;
    x = f(x), y = f(f(y));
  }
  return __gcd(prd, n);
}
vector<ull> factor(ull n) {
  if (n == 1) return {};
  if (isPrime(n)) return {n};
  ull x = pollard(n);
  auto l = factor(x), r = factor(n/x);
  l.insert(l.end(), r.begin(), r.end());
  return l;
} // hash-cpp-all = aeb78d808e824202d52e0e5e78ce624b
```

5.3 Divisibility

ExtendedEuclidean.h

Description: Finds two integers x and y, such that $ax+by = \gcd(a,b)$. If you just need gcd, use the built in __gcd instead. If a and b are coprime, then x is the inverse of $a \pmod b$. 10 lines

```
template<typename T>
T egcd(T a, T b, T &x, T &y) {
    if (a == 0) {
        x = 0, y = 1;
        return b;
    }
    T p = b/a, g = egcd(b - p * a, a, y, x);
    x -= y * p;
    return g;
} // hash-cpp-all = 7bbefb658dff0adc3293dd80efb22a86
```

DiophantineEquation.h

Description: Check if a the Diophantine Equation $ax + by = c$ has solution. 34 lines

```
template<typename T>
bool diophantine(T a, T b, T c, T &x, T &y, T &g) { // hash
  ↪-cpp-1
    if (a == 0 && b == 0) {
        if (c == 0) {
            x = y = g = 0;
            return true;
        }
        return false;
    }
    if (a == 0) {
        if (c % b == 0) {
            x = 0; y = c / b; g = abs(b);
            return true;
        }
        return false;
    }
    if (b == 0) {
        if (c % a == 0) {
            x = c / a; y = 0; g = abs(a);
            return true;
        }
        return false;
    }
    g = egcd<lint>(a, b, x, y);
    if (c % g != 0) return false;
    T dx = c / a;
    c -= dx * a;
    T dy = c / b;
    c -= dy * b;
    x = dx + (T) ((__int128) x * (c / g) % b);
    y = dy + (T) ((__int128) y * (c / g) % a);
    g = abs(g);
    return true; // |x|, |y| <= max(|a|, |b|, |c|)
}
```

Divisors.h

Description: Get all divisors of n. 10 lines

```
vector<int> divisors(int n) {
    vector<int> result, aux;
    for (int i = 1; i*i <= n; ++i)
        if (n % i == 0) {
            result.push_back(i);
            if (i*i != n) aux.push_back(n/i);
        }
    for (int i = aux.size()-1; i+1; --i) result.push_back(
      ↪aux[i]);
    return result;
} // hash-cpp-all = bcec3ee4e6771883ca0790e0b62eb42f
```

Pell.h

Description: Find the smallest integer root of $x^2 - ny^2 = 1$ when n is not a square number, with the solution set $x_{k+1} = x_0x_k + ny_0y_k, y_{k+1} = x_0y_k + y_0x_k$. 16 lines

```
pair<int,int> Pell(int n) {
  int p0 = 0, p1 = 1, q0 = 1, q1 = 0;
  int a0 = (int)sqrt(n), a1 = a0, a2 = a0;
  if(a0 * a0 == n) return {-1, -1};
```

```
  int g1 = 0, h1 = 1;
  while (1) {
    int g2 = -g1 + a1 * h1;
    int h2 = (n - g2 * g2)/h1;
    a2 = (g2 + a0)/h2;
    int p2 = a1 * p1 + p0;
    int q2 = a1 * q1 + q0;
    if (p2*p2 - n*q2*q2 == 1) return {p2, q2};
    a1 = a2; g1 = g2; h1 = h2; p0 = p1;
    p1 = p2; q0 = q1; q1 = q2;
  }
} // hash-cpp-all = 21e64f801a6641a3f2537e6f2f604860
```

PrimeFactors.h

Description: Find all prime factors of n.

"Sieve.h" 13 lines

```
vector<int> primeFac(int n){
    vector<int> factors;
    int idx = 0, prime_factors = primes[idx];
    while (prime_factors * prime_factors <= n){
        while (n % prime_factors == 0) {
            n /= prime_factors;
            factors.push_back(prime_factors);
        }
        prime_factors = primes[++idx];
    }
    if (n != 1) factors.push_back(n);
    return factors;
} // hash-cpp-all = 36e8e616befeeb19a350d9ad4e50cb0e
```

NumDiv.h

Description: Count the number of divisors of n.

"Sieve.h" 15 lines

```
int NumDiv(lint n){
    int idx = 0, prime_factors = primes[idx];
    lint ans = 1;
    while (prime_factors * prime_factors <= n) {
        int power = 0;
        while (n % prime_factors == 0) {
            n /= prime_factors;
            ++power;
        }
        ans *= 1ll * (power + 1);
        prime_factors = primes[++idx];
    }
    if (n != 1) ans *= 2;
    return ans;
} // hash-cpp-all = b91d21d80ce02a37a2911bcc84a23458
```

SumDiv.h

Description: Sum of all divisors of n.

"Sieve.h", "Modpow.h" 15 lines

```
lint sumDiv(int n){
    int idx = 0, prime_factors = primes[idx];
    lint ans = 1;
    while (prime_factors * prime_factors <= n){
        int power = 0;
        while (n % prime_factors == 0) {
            n /= prime_factors;
            power++;
        }
        ans *= (Pow(prime_factors, power+1)-1)/(
            ↪prime_factors-1);
        prime_factors = primes[++idx];
    }
    if (n != 1) ans *= (n*n - 1)/(n-1);
    return ans;
} // hash-cpp-all = d5dee31ad4cda543fe1f214b4cc3953d
```

Bezout.h

Description: Let $d := mdc(a, b)$. Then, there exist a pair x and y such that $ax + by = d$.

5 lines

```
pair<int, int> find_bezout(int x, int y) {
    if (y == 0) return bezout(1, 0);
    pair<int, int> g = find_bezout(y, x % y);
    return {g.second, g.first - (x/y) * g.second};
} // hash-cpp-all = d5ea908f84c746952727ecfe20a4f6f4
```

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 $\Rightarrow a^{p-1} \equiv 1 \pmod{p}$ $\forall a$.

22 lines

```
const int n = int(1e5) * 5;
vector<int> phi(n);
void calculatePhi() {
  for(int i = 0; i < n; ++i) phi[i] = i&1 ? i : i/2;
  for(int i = 3; i < n; i += 2) if(phi[i] == i)
    for(int j = i; j < n; j += i) phi[j] -= phi[j] / i;
}
template<typename T> T phi(T n){
    T aux, result;
    aux = result = n;
    for (T i = 2; i*i <= n; ++i)
        if (aux % i == 0) {
            while (aux % i == 0) aux /= i;
            result /= i;
            result *= (i-1);
        }
    if (aux > 1) {
      result /= aux;
      result *= (aux-1);
    }
    return result;
} // hash-cpp-all = 6984ee4343bd4e28b496afc6ebcec408
```

DiscreteLogarithm.h

Description: Returns the smallest $x \geq 0$ s.t. $a^x = b \pmod{m}$. a and m must be coprime.

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

10 lines

```
lint modLog(lint a, lint b, lint m) {
    assert(__gcd(a, m) == 1);
    lint n = (lint) sqrt(m) + 1, e = 1, x = 1, res =
        ↪LLONG_MAX;
    unordered_map<lint, lint> f;
    for(int i = 0; i < n; ++i) e = e * a % m;
    for(int i = 0; i < n; ++i) x = x * e % m, f.emplace(x,
        ↪i + 1);
    for(int i = 0; i < n; ++i)
        if (f.count(b = b * a % m)) res = min(res, f[b] * n
            ↪ - i - 1);
    return res;
} // hash-cpp-all = 4e6790ea248af84e0e24fd996ab7b22f
```

Legendre.h

Description: Given an integer n and a prime number p, find the largest x such that p^x divides $n!$.

8 lines

```
int legendre(int n, int p){
    int ret = 0, prod = p;
    while (prod <= n) {
        ret += n/prod;
        prod *= p;
    }
    return ret;
} // hash-cpp-all = 81613f762a8ec7c41ca9f6db5e02878a
```

GroupOrder.h

Description: Calculate the order of a in Z_n. A group Z_n is cyclic if, and only if n = $1, 2, 4, p^k$ or $2p^k$, being p an odd prime number.

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

"Divisors.h" 6 lines

```
template<typename T> T order(T a, T n) {
    vector<T> d = divisors(phi(n));
    for (int i : v)
        if (modpow(a, i, n) == 1) return i;
    return -1;
} // hash-cpp-all = 82034ce2bc7ee33cb83eaa0602926616
```

PrimitiveRoots.h

Description: That is, g is a primitive root mod n if for every number x coprime to n there is an integer z s.t. $x \equiv g^z \pmod{n}$

17 lines

```
template<typename T>
T primitive_roots(T p) {
    T n = p - 1;
    vector<T> factors;
    for (int i = 2; i*i <= n; ++i) if (n % i == 0) {
        factors.push_back(i);
        while (n % i == 0) n /= i;
    }
    if (n > 1) factors.push_back(n);
    for (int i = 2; i <= p; ++i) {
        bool works = true;
        for (int j = 0; j < factors.size() && works; ++j)
            works &= modpow(i, (p-1)/factors[i], p) != 1;
        if (works) return i;
    }
    return -1;
} // hash-cpp-all = 0133e9368db22d6251359e2b964e566b
```

5.4 Chinese remainder theorem

ChineseRemainder.h

Description: Chinese Remainder Theorem.

crt(a, m, b, n) computes x such that $x \equiv a \pmod{m}$, $x \equiv b \pmod{n}$. If $|a| < m$ and $|b| < n$, x will obey $0 \leq x < \text{lcm}(m, n)$. Assumes $mn < 2^{62}$.

Time: $\mathcal{O}\left(\log(n)\right) - \mathcal{O}\left(n \log(LCM(m))\right)$

21 lines

```
template<typename T>
T crt(T a, T m, T b, T n, T &x, T &y) { // hash-cpp-1
  if (n > m) swap(a, b), swap(m, n);
  T g = egcd(m, n, x, y);
  assert((a - b) % g == 0); // else no solution
  x = (b - a) % n * x % n / g * m + a;
  return x < 0 ? x + m*n/g : x;
} // hash-cpp-1 = 7913facb67d55ef46cdf5f2ba5862ed5
template<typename T> // Solve system up to n congruences
T crt_system(vector<T> &a, vector<T> &m, int n) {
  for (int i = 0; i < n; ++i)
```

```
    a[i] = (a[i] % m[i] + m[i]) % m[i];
  T ret = a.front(), lcm = m.front();
  for (int i = 1; i < n; ++i) {
    T x, y;
    ret = crt(ret, lcm, a[i], m[i], x, y);
    T d = egcd(lcm, m[i], x = 0, y = 0);
    lcm = lcm * m[i] / d;
  }
  return ret;
}
```

5.5 Fractions

Fractions.h

Description: Template that helps deal with fractions. 31 lines

```
struct frac { // hash-cpp-1
    lint n, d;
    frac() { n = 0, d = 1; }
    frac(lint _n, lint _d) {
        n = _n, d = _d;
        lint g = __gcd(n,d); n /= g, d /= g;
        if (d < 0) n *= -1, d *= -1;
    }
    frac(lint _n) : frac(_n,1) {}
    friend frac abs(frac F) { return frac(abs(F.n),F.d); }
    friend bool operator<(const frac& l, const frac& r) {
      ↪return l.n*r.d < r.n*l.d; }
    friend bool operator==(const frac& l, const frac& r) {
      ↪return l.n == r.n && l.d == r.d; }
    friend bool operator!=(const frac& l, const frac& r) {
      ↪return !(l == r); }
    friend frac operator+(const frac& l, const frac& r) {
      ↪return frac(l.n*r.d+r.n*l.d,l.d*r.d); }
    friend frac operator-(const frac& l, const frac& r) {
      ↪return frac(l.n*r.d-r.n*l.d,l.d*r.d); }
    friend frac operator*(const frac& l, const frac& r) {
      ↪return frac(l.n*r.n,l.d*r.d); }
    friend frac operator*(const frac& l, int r) { return l*
      ↪frac(r,1); }
    friend frac operator*(int r, const frac& l) { return l*
      ↪r; }
    friend frac operator/(const frac& l, const frac& r) {
      ↪return l*frac(r.d,r.n); }
    friend frac operator/(const frac& l, const int& r) {
      ↪return l/frac(r,1); }
    friend frac operator/(const int& l, const frac& r) {
      ↪return frac(l,1)/r; }
    friend frac& operator+=(frac& l, const frac& r) {
      ↪return l = l+r; }
    friend frac& operator-=(frac& l, const frac& r) {
      ↪return l = l-r; }
    template<class T> friend frac& operator*=(frac& l,
      ↪const T& r) { return l = l*r; }
    template<class T> friend frac& operator/=(frac& l,
      ↪const T& r) { return l = l/r; }
    friend ostream& operator<<(ostream& strm, const frac& a
      ↪) {
        strm << a.n;
        if (a.d != 1) strm << "/" << a.d;
        return strm;
    }
};
```

ContinuedFractions.h

Description: Given N and a real number $x \geq 0$, finds the closest rational approximation p/q with $p, q \leq N$. It will obey $|p/q - x| \leq 1/qN$. For consecutive convergents, $p_{k+1}q_k - q_{k+1}p_k = (-1)^k$. ($p_k/q_k$ alternates between $> x$ and $< x$.) If x is rational, y eventually becomes ∞; if x is the root of a degree 2 polynomial the a's eventually become cyclic.
Time: $\mathcal{O}(\log N)$ 21 lines

```
typedef double d; // for N ~ 1e7; long double for N ~ 1e9
pair<lint, lint> approximate(d x, lint N) { // hash-cpp-1
  lint LP = 0, LQ = 1, P = 1, Q = 0, inf = LLONG_MAX; d y =
    ↪ x;
  for (;;) {
    lint lim = min(P ? (N-LP) / P : inf, Q ? (N-LQ) / Q :
      ↪inf),
      a = (lint)floor(y), b = min(a, lim),
      NP = b*P + LP, NQ = b*Q + LQ;
    if (a > b) {
      // If b > a/2, we have a semi-convergent that gives
        ↪us a
      // better approximation; if b = a/2, we *may* have
        ↪one.
      // Return {P, Q} here for a more canonical
        ↪approximation.
      return (abs(x - (d)NP / (d)NQ) < abs(x - (d)P / (d)Q)
        ↪) ?
      {NP, NQ} : {P, Q};
    }
    if (abs(y = 1/(y - (d)a)) > 3*N) {
      return {NP, NQ};
    }
    LP = P; P = NP;
    LQ = Q; Q = NQ;
  }
} // hash-cpp-1 = e3f27076ec30785b7826aabd1eb5ac59
```

FracBinarySearch.h

Description: Given f and N, finds the smallest fraction $p/q \in [0, 1]$ such that $f(p/q)$ is true, and $p, q \leq N$. You may want to throw an exception from f if it finds an exact solution, in which case N can be removed.
Usage: `fracBS([](Frac f) { return f.p>=3*f.q; }, 10); // {1,3}`
Time: $\mathcal{O}(\log(N))$ 23 lines

```
struct Frac { lint p, q; };
template<class F>
Frac fracBS(F f, lint N) { // hash-cpp-1
  bool dir = 1, A = 1, B = 1;
  Frac left{0, 1}, right{1, 1}; // Set right to 1/0 to
    ↪search (0, N]
  assert(!f(left)); assert(f(right));
  while (A || B) {
    lint adv = 0, step = 1; // move right if dir, else left
    for (int si = 0; step; (step *= 2) >>= si) {
      adv += step;
      Frac mid{left.p * adv + right.p, left.q * adv + right
        ↪.q};
      if (abs(mid.p) > N || mid.q > N || dir == !f(mid)) {
        adv -= step; si = 2;
      }
    }
    right.p += left.p * adv;
    right.q += left.q * adv;
    dir = !dir;
    swap(left, right);
    A = B; B = !!adv;
  }
  return dir ? right : left;
} // hash-cpp-1 = 66f3c71eb28df4393cd2a2abbea9345e
```

5.5.1 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

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

5.5.2 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.

5.5.3 Primitive Roots

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

5.5.4 Chicken McNugget theorem

Sejam x e y dois inteiros coprimos, o maior inteiro que não pode ser escrito como $ax + by$ é $\frac{(x-1)(y-1)}{2}$

5.6 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 $\mathbb{Z}_2 \times \mathbb{Z}_{2^{a-2}}$.

5.6.1 Sum of primes

For any multiplicative f:

$$S(n, p) = S(n, p-1) - f(p) \cdot (S(n/p, p-1) - S(p-1, p-1))$$

5.7 Mobius Function

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

Mobius Inversion: If $g(n) = \sum_{d|n} f(d)$, then $f(n) = \sum_{d|n} \mu(d) g(n/d)$.

$$\sum_{d|n} \mu(d) = \begin{cases} 1 & \text{if } n = 1 \\ 0 & \text{if } n > 1 \end{cases}$$

5.7.1 Wilson's theorem

Seja $n > 1$. Então $n|(n-1)! + 1$ sse n é primo.

5.7.2 Wolstenholme's theorem

Seja $p > 3$ um número primo. Então o numerador do número $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{p-1}$ é divisível por p^2.

5.7.3 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 $n < 1e19$.

Let $s(x) = \sum_{i=1}^{x} \phi(i)$. Then

$$s(n) = \frac{n(n+1)}{2} - \sum_{i=2}^{n} s\left(\left\lfloor \frac{n}{i} \right\rfloor\right)$$

5.7.4 Prime counting function ($\pi(x)$)

The prime counting function is asymptotic to $\frac{x}{\log x}$, by the prime number theorem.

x	10	10^2	10^3	10^4	10^5	10^6	10^7	10^8
$\pi(x)$	4	25	168	1.229	9.592	78.498	664.579	5.761.455

Combinatorial (6)

6.1 Permutations

6.1.1 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		

Factorial.h

Description: Pre-compute all the factorial numbers until n. 11 lines

```
void init(int n) {
    fact.resize(n + 1);
    fact[0] = 1;
    for (int i = 1; i <= n; ++i)
        fact[i] = (lint)i * fact[i-1] % mod;
    ifact.resize(n + 1);
    ifact[n] = modinv(fact[n], mod);
    for (int i = n-1; i >= 0; --i)
        ifact[i] = (lint)(i+1) * ifact[i+1] % mod;
}
// hash-cpp-all = 4635b5a3176ad24eb7ad8c63e12007b2
```

numPerm.h

Description: Number of permutations 6 lines

```
lint num_perm(int n, int r) {
    if (r < 0 || n < r) return 0;
    lint ret = 1;
    for (int i = n; i > n-r; --i) ret *= i;
    return ret;
} // hash-cpp-all = 9063aaab522de1bd1cbb483b1e4d6a39
```

6.1.2 Binomials

PascalTriangle.h

Description: Pre-compute all the Binomial Coefficients until n.
Time: $\mathcal{O}(N^2)$ 9 lines

```
void init() {
  c[0][0] = 1;
  for (int i = 0; i < n; ++i) {
      c[i][0] = c[i][i] = 1;
      for (int j = 1; j < i; ++j)
          c[i][j] = c[i-1][j-1] + c[i-1][j];
  }
}
// hash-cpp-all = 8ccd6947f990e14d2a4eaf3f588f1c05
```

nCr.h

Description: Pre-compute all the factorial numbers until lim.
Time: $\mathcal{O}(N + \log(mod))$ 12 lines

```
num ncr(int n, int k) {
  num res = 1;
  for (int i = 1; i <= k; ++i) {
    res *= (n - i + 1);
    res /= i;
  }
  return res;
}
num ncr(int n, int k){
  if(k < 0 || k > n) return 0;
  return fact[n] * ifact[k] * ifact[n-k];
} // hash-cpp-all = 37b9ceba2a47c82c591f93858ef8fba4
```

Multinomial.h

Description: Computes $\binom{k_1 + \cdots + k_n}{k_1, k_2, \ldots, k_n} = \frac{(\sum k_i)!}{k_1! k_2! \ldots k_n!}$. 7 lines

```
lint multinomial(vector<int>& v) {
  lint c = 1, m = v.empty() ? 1 : v[0];
  for (int i = 1 < v.size(); ++i)
      for (int j = 0; j < v[i]; ++j)
          c = c * ++m / (j+1);
  return c;
} // hash-cpp-all = 864cdb12b60507bb64330bca4f60b112
```

6.1.3 Involutions

Uma involução é uma permutação com ciclo de tamanho máximo 2, e é a sua própria inversa.

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

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

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

6.1.4 Cycles

Suponha que $g_S(n)$ é o número de n-permutações quais o tamanho do ciclo pertence ao conjunto S. Então

$$\sum_{n=0}^{\infty} g_S(n) \frac{x^n}{n!} = \exp\left(\sum_{n \in S} \frac{x^n}{n}\right)$$

6.1.5 Inclusion-Exclusion Principle

Sejam $A_1, A_2, ..., A_n$ conjuntos. Então o número de elementos da união $A_1 \cup A_2 \cup ... \cup A_n$ é

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{\substack{I \subseteq \{1,2,\ldots,n\} \\ I \neq \emptyset}} (-1)^{|I|+1} \left|\bigcap_{i \in I} A_i\right|$$

6.1.6 The twelvefold way (from Stanley)

How many functions $f\colon N \to X$ are there?

N	X	Any f	Injective	Surjective
dist.	dist.	x^n	$\frac{x!}{(x-n)!}$	$x!\left\{{n \atop x}\right\}$
indist.	dist.	$\binom{x+n-1}{n}$	$\binom{x}{n}$	$\binom{n-1}{n-x}$
dist.	indist.	$\left\{{n \atop 1}\right\} + \ldots + \left\{{n \atop x}\right\}$	$[n \leq x]$	$\left\{{n \atop k}\right\}$
indist.	indist.	$p_1(n) + \ldots p_x(n)$	$[n \leq x]$	$p_x(n)$

Where $\binom{a}{b} = \frac{1}{b!}(a)_b$, $p_x(n)$ is the number of ways to partition the integer n using x summand and $\left\{{n \atop x}\right\}$ is the number of ways to partition a set of n elements into x subsets (aka Stirling number of the second kind).

6.1.7 Burnside

Seja $A\colon GX \to X$ uma ação. Defina:

- $w \coloneqq$ número de órbitas em X.
- $S_x \coloneqq \{g \in G \mid g \cdot x = x\}$
- $F_g \coloneqq \{x \in X \mid g \cdot x = x\}$

Então $w = \frac{1}{|G|} \sum_{x \in X} |S_x| = \frac{1}{|G|} \sum_{g \in G} |F_g|$.

6.1.8 Derangements

Permutações de um conjunto tais que nenhum dos elementos aparecem em sua posição original.

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

6.2 Partitions and subsets

6.2.1 Partition function

Número de formas de escrever n como a soma de inteiros positivos, independente da ordem deles.

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

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

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

6.2.2 Lucas's theorem

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 $\binom{n}{m} \equiv \prod_{i=0}^{k} \binom{n_i}{m_i} \pmod{p}$.

6.3 General purpose numbers

6.3.1 Bernoulli numbers

EGF of Bernoulli numbers is $B(t) = \frac{t}{e^t - 1}$ (FFT-able).
$B[0, \ldots] = [1, -\frac{1}{2}, \frac{1}{6}, 0, -\frac{1}{30}, 0,$
frac142, $\ldots]$

Sums of powers:

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

Fórmula de Euler-Maclaurin para somas infinitas:

$$\sum_{i=m}^{\infty} f(i) = \int_m^{\infty} f(x)dx - \sum_{k=1}^{\infty} \frac{B_k}{k!} f^{(k-1)}(m)$$

$$\approx \int_m^{\infty} f(x)dx + \frac{f(m)}{2} - \frac{f'(m)}{12} + \frac{f'''(m)}{720} + O(f^{(5)}(m))$$

6.3.2 Stirling numbers of the first kind

Número de permutações em n itens com k ciclos.

$c(n,k) = c(n-1,k-1) + (n-1)c(n-1,k), c(0,0) = 1$
$\sum_{k=0}^{n} c(n,k)x^k = x(x+1)\ldots(x+n-1)$
$c(8,k) = 8, 0, 5040, 13068, 13132, 6769, 1960, 322, 28, 1$
$c(n,2) =$
$0, 0, 1, 3, 11, 50, 274, 1764, 13068, 109584, \ldots$

6.3.3 Eulerian numbers

Número de permutações $\pi \in S_n$ na qual exatamente k elementos são maiores que os anteriores. 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. $\pi(j) > j$.

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

$$E(n,0) = E(n,n-1) = 1$$

$$E(n,k) = \sum_{j=0}^{k} (-1)^j \binom{n+1}{j} (k+1-j)^n$$

6.3.4 Stirling numbers of the second kind

Partições de n elementos distintos em exatamente k grupos.

$$S(n,k) = S(n-1,k-1) + kS(n-1,k)$$

$$S(n,1) = S(n,n) = 1$$

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

6.3.5 Bell numbers

Número total de partições de n elementos distintos.
$B(n) = 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, \ldots.$

$$\mathcal{B}_{n+1} = \sum_{k=0}^{n} \binom{n}{k} \mathcal{B}_k$$

Também é possível calcular usando Stirling numbers of the second kind,

$$B_n = \sum_{k=0}^{n} S(n,k)$$

Já para p primo,

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

6.3.6 Labeled unrooted trees

em n vertices: n^{n-2}
em k árvores existentes de tamanho n_i:
$n_1 n_2 \cdots n_k n^{k-2}$
de grau d_i: $(n-2)!/((d_1-1)! \cdots (d_n-1)!)$
florestas com exatamente k árvores enraizadas:

$$\binom{n}{k} k \cdot n^{n-k-1}$$

6.3.7 Catalan numbers

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

$$C_0 = 1,\ C_{n+1} = \frac{2(2n+1)}{n+2} C_n,\ C_{n+1} = \sum C_i C_{n-i}$$

$C_n = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \ldots$

- sub-diagonal monotone paths in a $n \times n$ grid.
- strings with n pairs of parenthesis, correctly nested.
- binary trees with with $n+1$ leaves (0 or 2 children) or $2n+1$ elements.
- ordered trees with $n+1$ vertices.
- # ways a convex polygon with $n+2$ sides can be cut into triangles by connecting vertices with straight lines.
- permutations of $[n]$ with no 3-term increasing subsequence.

6.3.8 Super Catalan numbers

The number of monotonic lattice paths of a nxn 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

6.3.9 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

6.3.10 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

6.3.11 Schroder 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

6.3.12 Triangles

Given rods of length 1, ..., 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 Game Theory

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

6.4.1 Nim

Let $X = \bigoplus_{i=1}^{n} x_i$, then $(x_i)_{i=1}^{n}$ is a winning position iff $X \neq 0$. Find a move by picking k such that $x_k > x_k \oplus X$.

6.4.2 Misère Nim

Regular Nim, except that the last player to move *loses*. Play regular Nim until there is only one pile of size larger than 1, reduce it to 0 or 1 such that there is an odd number of piles. The second player wins $(a_1, \ldots, a_n)$ if 1) there is a pile $a_i > 1$ and $\oplus_{i=1}^{n} a_i = 0$ or 2) all $a_i \le 1$ and $\oplus_{i=1}^{n} a_i = 1$.

6.4.3 Staircase Nim

Stones are moved down a staircase and only removed from the last pile. $(x_i)_{i=1}^{n}$ is an L-position if $(x_{2i-1})_{i=1}^{n/2}$ is (i.e. only look at odd-numbered piles).

6.4.4 Moore's Nim$_k$

The player may remove from at most k piles (Nim = Nim$_1$). Expand the piles in base 2, do a carry-less addition in base $k+1$ (i.e. the number of ones in each column should be divisible by $k+1$).

6.4.5 Dim$^+$

The number of removed stones must be a divisor of the pile size. The Sprague-Grundy function is $k+1$ where 2^k is the largest power of 2 dividing the pile size.

6.4.6 Aliquot Game

Same as above, except the divisor should be proper (hence 1 is also a terminal state, but watch out for size 0 piles). Now the Sprague-Grundy function is just k.

6.4.7 Nim (at most half)

Write $n + 1 = 2^m y$ with m maximal, then the Sprague-Grundy function of n is $(y-1)/2$.

6.4.8 Lasker's Nim

Players may alternatively split a pile into two new non-empty piles. $g(4k+1) = 4k+1$, $g(4k+2) = 4k+2$, $g(4k+3) = 4k+4$, $g(4k+4) = 4k+3$ ($k \ge 0$).

6.4.9 Hackenbush on Trees

A tree with stalks $(x_i)_{i=1}^{n}$ may be replaced with a single stalk with length $\bigoplus_{i=1}^{n} x_i$.

Nim-Product.cpp

Description: Product of nimbers is associative, commutative, and distributive over addition (xor). Forms finite field of size 2^{2^k}. Application: Given 1D coin turning games G_1, G_2 $G_1 \times G_2$ is the 2D coin turning game defined as follows. If turning coins at $x_1, x_2, \ldots, x_m$ is legal in G_1 and $y_1, y_2, \ldots, y_n$ is legal in G_2, then turning coins at all positions (x_i, y_j) is legal assuming that the coin at (x_m, y_n) goes from heads to tails. Then the grundy function $g(x, y)$ of $G_1 \times G_2$ is $g_1(x) \times g_2(y)$.
Time: 64^2 xors per multiplication, memorize to speed up.
27 lines

```
using ull = uint64_t;
ull _nimProd2[64][64];
ull nimProd2(int i, int j) {
    if (_nimProd2[i][j]) return _nimProd2[i][j];
    if ((i & j) == 0) return _nimProd2[i][j] = 1ull << (i|j
       ↪);
    int a = (i&j) & -(i&j);
    return _nimProd2[i][j] = nimProd2(i ^ a, j) ^ nimProd2
       ↪((i ^ a) | (a-1), (j ^ a) | (i & (a-1)));
}
void allNimProd2() {
    for (int i = 0; i < 64; i++) {
        for (int j = 0; j < 64; j++) {
            if ((i & j) == 0) _nimProd2[i][j] = 1ull << (i|
               ↪j);
            else {
                int a = (i&j) & -(i&j);
                _nimProd2[i][j] = _nimProd2[i ^ a][j] ^
                   ↪_nimProd2[(i ^ a) | (a-1)][(j ^ a) | (
                   ↪i & (a-1))];
            }
        }
    }
}
ull nimProd(ull x, ull y) {
    ull res = 0;
    for (int i = 0; (x >> i) && i < 64; ++i)
        if ((x >> i) & 1)
            for (int j = 0; (y >> j) && j < 64; ++j)
                if ((y >> j) & 1) res ^= nimProd2(i, j);
    return res;
} // hash-cpp-all = 8d57cb8b36f37481a6cadacca39b731d
```

Partitions.cpp

Description: Fills array with partition function $p(n)$ $\forall 0 \le ileqn$
17 lines

```
array<int, 122> part; // 121 is max partition that will fit
   ↪ into int
void partition(int n) {
    part[0] = 1;
    for (int i = 1; i <= n; ++i) {
        part[i] = 0;
        for (int k = 1, x; k <= i; ++k) {
            x = i - k * (3*k-1)/2;
            if (x < 0) break;
            if (k&1) part[i] += part[x];
            else part[i] -= part[x];
            x = i - k * (3*k+1)/2;
            if (x < 0) break;
            if (k&1) part[i] += part[x];
            else part[i] -= part[x];
        }
```

```
    }
} // hash-cpp-all = 3b7c36794ae71cf38eae6fa9a135ace2
```

Lucas.h

Description: Lucas theorem
Time: $\mathcal{O}\left(log_p(n) mod_i nverse()\right)$

<ModTemplate.h>, <nCr.h> 10 lines

```
num lucas(lint n, lint m) {
  num c = 1;
  while (n || m) {
    lint x = n % MOD, y = m % MOD;
    if (x < y) return 0;
    c = c * ncr(x, y);
    n /= MOD; m /= MOD;
  }
  return c;
} // hash-cpp-all = fb6ac919315ce14ff27894d42b0fd271
```

Graph (7)

7.1 Fundamentals

BellmanFord.h

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

19 lines

```
const lint inf = LLONG_MAX;
struct edge_t { int a, b, w, s() { return a < b ? a : -a;
  ↪}};
struct node_t { lint dist = inf; int prev = -1; };
void bellmanFord(vector<node_t>& nodes, vector<edge_t>& eds
  ↪, int s) {
  nodes[s].dist = 0;
  sort(eds.begin(), eds.end(), [](edge_t a, edge_t b) {
    ↪return a.s() < b.s(); });
  int lim = nodes.size() / 2 + 2; // /3+100 with shuffled
    ↪vertices
  for(int i = 0; i < lim; ++i) for(auto &ed : eds) {
    node_t cur = nodes[ed.a], &dest = nodes[ed.b];
    if (abs(cur.dist) == inf) continue;
    lint d = cur.dist + ed.w;
    if (d < dest.dist) {
      dest.prev = ed.a;
      dest.dist = (i < lim-1 ? d : -inf);
    }
  }
  for(int i = 0; i < lim; ++i) for(auto &e : eds)
    if (nodes[e.a].dist == -inf) nodes[e.b].dist = -inf;
} // hash-cpp-all = 5d9aa7fd195b46e0152d63f8b21fadc0
```

FloydWarshall.h

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

16 lines

```
const lint inf = 1LL << 62;
void floydWarshall(vector<vector<lint>>& m) {
  int n = m.size();
  for (int i = 0; i < n; ++i) m[i][i] = min(m[i][i], {});
  for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < n; ++j)
        if (m[i][k] != inf && m[k][j] != inf) {
          auto newDist = max(m[i][k] + m[k][j], -inf);
          m[i][j] = min(m[i][j], newDist);
        }
  for (int k = 0; k < n; ++k) if (m[k][k] < 0)
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < n; ++j)
        if (m[i][k] != inf && m[k][j] != inf) m[i][j] =
          ↪ -inf;
} // hash-cpp-all = 578e31a61dfb8557ef1e1f4c611b2815
```

TopoSort.h

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

12 lines

```
vector<int> topo_sort(const vector<vector<int>> &g) {
  vector<int> indeg(g.size()), ret;
  for(auto &li : g) for(auto &x : li) indeg[x]++;
  queue<int> q; // use priority queue for lexic. smallest
    ↪ans.
  for(int i = 0; i < g.size(); ++i) if (indeg[i] == 0) q.
    ↪push(-i);
  while (!q.empty()) {
    int i = -q.front(); // top() for priority queue
    ret.push_back(i); q.pop();
    for(auto &x : g[i]) if (--indeg[x] == 0) q.push(-x);
  }
  return ret;
} // hash-cpp-all = d2ba1ef7b98de4bab3212a9a20c7220d
```

CutVertices.h

28 lines

```
vector<int> cut, seen, low, par;
vector<vector<int>> edges;
int Time = 0;
void dfs(int v, int p) {
    int cnt = 0;
    par[v] = p;
    low[v] = seen[v] = Time++;
    for (int u : edges[v]) {
        if (seen[u] == -1) {
            par[u] = v;
            dfs(u, v);
            low[v] = min(low[v], low[u]);
            if (low[u] >= seen[v]) cnt++;
            //if (low[u] > seen[v]) u-v bridge
        }
        else if (u != par[v]) low[v] = min(low[v], seen[u])
          ↪;
    }
    if (cnt > 1 || (seen[v] != 0 && cnt > 0)) cut[v] = 1;
}
void solve(int n) {
    cut.resize(n, 0); seen.resize(n, -1);
    low.resize(n, 0); par.resize(n, 0);
    for (int i = 0; i < n; ++i)
        if (seen[i] == -1) {
            Time = 0;
            dfs(i, i);
        }
} // hash-cpp-all = b21d97812df2cb2a3408ce071f662b21
```

Bridges.h

Description: Find bridges in an undirected graph G. Do not forget to set the first level as 1. (level[0] = 1)

18 lines

```
vector<vector<int>> edges;
vector<int> level, dp;
int bridge = 0;
void dfs(int v, int p) {
    dp[v] = 0;
    for (int u : edges[v]) {
        if (level[u] == 0) {
            level[u] = level[v] + 1;
            dfs(u, v);
            dp[v] += dp[u];
        }
        else if (level[u] < level[v]) dp[v]++;
        else if (level[u] > level[v]) dp[v]--;
    }
    dp[v]--;
    if (level[v] > 1 && dp[v] == 0) // Edge_vp is a bridge
        bridge++;
} // hash-cpp-all = 990615e56d90abaddbb7130047b6dd79
```

Dijkstra.cpp

Description: Calculates the shortest path between start node and every other node in the graph

19 lines

```
void dijkstra(vector<vector<pii>> &graph, vector<int> &dist
  ↪, int start){
  vector<bool> vis(n, 0);
  for(int i = 0; i < n; i++) dist[i] = INF;
  priority_queue <pii, vector<pii>, greater<pii>> q;
  q.push({dist[start] = 0,start});
  while(!q.empty()) {
    int u=q.top().nd;
    q.pop();
    vis[u]=1;
    for(pii p: graph[u]){
      int e=p.st, v=p.nd;
      if (vis[v]) continue;
      int new_dist=dist[u]+e;
      if(new_dist<dist[v]){
        q.push({dist[v] = new_dist,v});
      }
    }
  }
} // hash-cpp-all = dca271572a4b037e16e5d9002cc482c3
```

Prim.h

Description: Find the minimum spanning tree. Better for dense graphs.
Time: $\mathcal{O}\left(E \log V\right)$

18 lines

```
priority_queue<pair<int, int>> pq;
void process(int v) {
    seen[v] = true;
    for (auto u : edges[v])
        if (!seen[u.first])
            pq.push({-u.second, -u.first});
}
int mst() {
    int mst_cost = 0; process(0);
    while (!pq.empty()) {
        auto v = pq.top(); pq.pop();
        int u = -v.second, w = -v.first;
        if (!seen[u]) mst_cost += w;
        process(u);
    }
```

```
    return mst_cost;
}
// hash-cpp-all = 69c0c1a3a468dcd6310de0dd51b9cf09
```

Kruskal.h

Description: Find the minimum spanning tree. Better for sparse graphs.
Time: $\mathcal{O}(E \log E)$

12 lines

```
template<typename T>
T kruskal(int n, vector<pair<T, pair<int,int>>> &edges) {
    sort(edges.begin(), edges.end());
    T cost = 0;
    UF dsu(n);
    for (auto &e : edges)
        if (dsu.find(e.second.first) != dsu.find(e.second.
            ↪second)) {
            dsu.unite(e.second.first, e.second.second);
            cost += e.first;
        }
    return cost;
} // hash-cpp-all = 3fba6712e1bea52f808aec6f571c6f35
```

SPFA.h

Description: Shortest Path Faster Algorithm.
Time: $\mathcal{O}(E)$

22 lines

```
int d[100100], f[100100];
vector<pair<int,int>> edges[100100];
void spfa(int s = 0) {
    vector<int> q = {s};
    memset(d, 127, sizeof(d));
    memset(f, 0, sizeof(f));
    f[s] = 1, d[s] = 0;
    for (int i = 0; i < q.size(); ++i) {
        int now = q[i];
        f[now] = 0;
        for(auto u : edges[now]) {
            int cost = u.second;
            if (d[u.first] > d[now] + cost) {
                d[u.first] = d[now] + cost;
                if (!f[u.first]) {
                    f[u.first] = 1;
                    q.push_back(u.first);
                }
            }
        }
    }
} // hash-cpp-all = 579f7957494e7f9d510e121d3c2226ef
```

EulerWalk.h

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

16 lines

```
using pii = pair<int,int>;
vector<int> eulerWalk(vector<vector<pii>>& gr, int nedges,
    ↪int src=0) {
    int n = gr.size();
    vector<int> D(n), its(n), eu(nedges), ret, s = {src};
    D[src]++; // to allow Euler paths, not just cycles
    while (!s.empty()) {
        int x = s.back(), y, e, &it = its[x], end = gr[x].
            ↪size();
        if (it == end){ ret.push_back(x); s.pop_back();
            ↪continue; }
        tie(y, e) = gr[x][it++];
        if (!eu[e]) {
            D[x]--, D[y]++;
            eu[e] = 1; s.push_back(y);
        }}
    for(auto &x : D) if (x < 0 || ret.size() != nedges+1)
        ↪return {};
    return {ret.rbegin(), ret.rend()};
} // hash-cpp-all = 400c6e63c2e9553cfc3b4909f8898483
```

7.2 Network flow

PushRelabel.h

Description: Push-relabel using the highest label selection rule and the gap heuristic. Quite fast in practice. To obtain the actual flow, look at positive values only. id can be used to restore each edge and its amount of flow used.
Time: $\mathcal{O}\left(V^2\sqrt{E}\right)$ Better for dense graphs - Slower than Dinic (in practice)

42 lines

```
template<typename Flow = lint> struct PushRelabel {
  struct edge_t { int dest, back; Flow f, c; };
  vector<vector<edge_t>> g;
  vector<Flow> ec;
  vector<edge_t*> cur;
  vector<vector<int>> hs; vector<int> H;
  PushRelabel(int n) : g(n), ec(n), cur(n), hs(2*n), H(n)
      ↪{}
  void addEdge(int s, int t, Flow cap, Flow rcap = 0) {
    if (s == t) return;
    g[s].push_back({t, (int)g[t].size(), 0, cap});
    g[t].push_back({s, (int)g[s].size()-1, 0, rcap});
  }
  void addFlow(edge_t& e, Flow f) {
    edge_t &back = g[e.dest][e.back];
    if (!ec[e.dest] && f) hs[H[e.dest]].push_back(e.dest);
    e.f += f; e.c -= f; ec[e.dest] += f;
    back.f -= f; back.c += f; ec[back.dest] -= f;
  }
  Flow maxflow(int s, int t) {
    int v = g.size(); H[s] = v; ec[t] = 1;
    vector<int> co(2*v); co[0] = v-1;
    for(int i = 0; i < v; ++i) cur[i] = g[i].data();
    for(auto& e : g[s]) addFlow(e, e.c);
    for (int hi = 0;;) {
      while (hs[hi].empty()) if (!hi--) return -ec[s];
      int u = hs[hi].back(); hs[hi].pop_back();
      while (ec[u] > 0)  // discharge u
        if (cur[u] == g[u].data() + g[u].size()) {
          H[u] = 1e9;
          for(auto &e : g[u]) if (e.c && H[u] > H[e.dest
              ↪]+1)
            H[u] = H[e.dest]+1, cur[u] = &e;
          if (++co[H[u]], !--co[hi] && hi < v)
            for(int i = 0; i < v; ++i) if (hi < H[i] && H[i
                ↪] < v)
              --co[H[i]], H[i] = v + 1;
          hi = H[u];
        } else if (cur[u]->c && H[u] == H[cur[u]->dest]+1)
          addFlow(*cur[u], min(ec[u], cur[u]->c));
        else ++cur[u];
    }
  }
  bool leftOfMinCut(int a) { return H[a] >= g.size(); }
}; // hash-cpp-all = 58d957708bdc815dfb6a929a5eabe78c
```

Dinitz.h

Description: Flow algorithm with complexity $O(VE \log U)$ where $U = \max|\text{cap}|$. $O(\min(E^{1/2}, V^{2/3})E)$ if $U = 1$; $O(\sqrt{V}E)$ for bipartite matching. To obtain each partition A and B of the cut look at lvl, for $v \subset A$, $lvl[v] > 0$, for $u \subset B$, $lvl[u] = 0$.

45 lines

```
template<typename T = lint> struct Dinitz { // hash-cpp-1
  struct edge_t { int to, rev; T c, f; };
  vector<vector<edge_t>> adj;
  vector<int> lvl, ptr, q;
  Dinitz(int n) : lvl(n), ptr(n), q(n), adj(n) {}
  inline void addEdge(int a, int b, T c, T rcap = 0) {
    adj[a].push_back({b, (int)adj[b].size(), c, 0});
    adj[b].push_back({a, (int)adj[a].size() - 1, rcap, 0});
  } // hash-cpp-1 = 84721ec2e4cb4ddd4ecd05de725ec92d
  T dfs(int v, int t, T f) { // hash-cpp-2
    if (v == t || !f) return f;
    for (int &i = ptr[v]; i < adj[v].size(); ++i) {
      edge_t &e = adj[v][i];
      if (lvl[e.to] == lvl[v] + 1)
        if (T p = dfs(e.to, t, min(f, e.c - e.f))) {
          e.f += p, adj[e.to][e.rev].f -= p;
          return p;
        }
    }
    return 0;
  } // hash-cpp-2 = 990b8517435ec0305e94f9cc76e99974
  T maxflow(int s, int t) { // hash-cpp-3
    T flow = 0; q[0] = s;
    for (int L = 0; L < 31; ++L) do { // 'int L=30' maybe
        ↪faster for random data
      lvl = ptr = vector<int>(q.size());
      int qi = 0, qe = lvl[s] = 1;
      while (qi < qe && !lvl[t]) {
        int v = q[qi++];
        for (edge_t &e : adj[v])
          if (!lvl[e.to] && (e.c - e.f) >> (30 - L))
            q[qe++] = e.to, lvl[e.to] = lvl[v] + 1;
      }
      while (T p = dfs(s, t, numeric_limits<T>::max()/4))
          ↪flow += p;
    } while (lvl[t]);
    return flow;
  } // hash-cpp-3 = db2141ffd86aebc46b790c57ee36b535
  bool leftOfMinCut(int v) { return lvl[v] != 0; }
  pair<T, vector<pair<int,int>>> minCut(int s, int t) { //
      ↪hash-cpp-4
    T cost = maxflow(s,t);
    vector<pair<int,int>> cut;
    for (int i = 0; i < adj.size(); i++) for(edge_t &e :
        ↪adj[i])
      if (lvl[i] && !lvl[e.to]) cut.push_back({i, e.to});
    return {cost, cut};
  } // hash-cpp-4 = fe5b29f2811efc3bc088924354375666
};
```

HLPP.h

Description: Highest label preflow push algorithm. Use it only if you really need the fastest maxflow algo. One limitation of the HLPP implementation is that you can't recover the weights for the full flow - use Dinic's for this.
Time: $\mathcal{O}\left(V^2\sqrt{E}\right)$. Faster than Dinic with scaling(in practice).

"../../content/various/LinkedList.h" 90 lines

```cpp
template <typename T, bool UseGlobal = true, bool UseGap =
    ↪true>
struct HLPP {
    struct edge_t { int to, rev; T cap; };
    const T INF = numeric_limits<T>::max();
    int n, highest_active, highest;
    vector<vector<edge_t>> adj;
    vector<int> height, count, que;
    vector<T> excess;
    LinkedList list;
    DoublyLinkedList dlist;
    HLPP(int n) : n(n), adj(n), que(n), list(n), dlist(n)
        ↪{}
    inline void addEdge(int from, int to, T cap, T rcap =
        ↪0) {
        adj[from].push_back({to, (int)adj[to].size(), cap})
            ↪;
        adj[to].push_back({from, (int)adj[from].size() - 1,
            ↪ rcap});
    }
    void globalRelabel(int t) {
        if (!UseGlobal) return;
        height.assign(n, n); height[t] = 0;
        count.assign(n, 0);
        int qh = 0, qt = 0;
        for (que[qt++] = t; qh < qt;) {
            int u = que[qh++], h = height[u] + 1;
            for (edge_t &e : adj[u]) if (height[e.to] == n
                ↪&& adj[e.to][e.rev].cap > 0) {
                    count[height[e.to] = h]++;
                    que[qt++] = e.to;
                }
        }
        list.clear(); dlist.clear();
        for (int u = 0; u < n; ++u) if (height[u] < n) {
                dlist.insert(height[u], u);
                if (excess[u] > 0) list.push(height[u], u);
            }
        highest = highest_active = height[que[qt-1]];
    }
    void push(int u, edge_t &e) {
        int v = e.to;
        T df = min(excess[u], T(e.cap));
        e.cap -= df, adj[v][e.rev].cap += df;
        excess[u] -= df, excess[v] += df;
        if (0 < excess[v] && excess[v] <= df) list.push(
            ↪height[v], v);
    }
    void discharge(int u) {
        int nh = n;
        for (edge_t &e : adj[u]) if (e.cap > 0)
                if (height[u] == height[e.to] + 1) {
                    push(u, e);
                    if (excess[u] == 0) return;
                } else nh = min(nh, height[e.to] + 1);
        int h = height[u];
        if (UseGap && count[h] == 1) {
            dlist.erase_all(h, highest, [&](int u) {
                count[height[u]]--, height[u] = n; });
            highest = h - 1;
        } else {
            count[h]--; dlist.erase(h, u);
            height[u] = nh;
            if (nh == n) return;
            count[nh]++; dlist.insert(nh, u);
            highest = max(highest, highest_active = nh);
            list.push(nh, u);
        }
    }
    T maxflow(int s, int t) {
        if (s == t) return 0;
        highest_active = 0; // highest label (active)
        highest = 0; // highest label (active and inactive)
        height.assign(n, 0); height[s] = n;
        for (int i = 0; i < n; ++i) if (i != s) dlist.
            ↪insert(height[i], i);
        count.assign(n, 0); count[0] = n - 1;
        excess.assign(n, 0); excess[s] = INF; excess[t] = -
            ↪INF;
        for (edge_t &e : adj[s]) push(s, e);
        globalRelabel(t);
        for (int u = -1, rest = n; highest_active >= 0; ) {
            if ((u = list.front(highest_active)) < 0) { --
                ↪highest_active; continue; }
            list.pop(highest_active);
            discharge(u);
            if (--rest == 0) rest = n, globalRelabel(t);
        }
        return excess[t] + INF;
    }
    bool leftOfMinCut(int a) { return height[a] >= adj.size
        ↪(); }
    pair<int, vector<pair<int,int>>> minCut(int s, int t) {
        T maxflow = maxflow(s, t);
        vector<pair<int,int>> cut; // if 0-indexed
        for (int i = 0; i < n; ++i) for (edge_t &e : adj[i
            ↪])
            if (leftOfMinCut(i) && !leftOfMinCut(e.to))
                cut.push_back({i, e.to});
        return {maxflow, cut};
    }
}; // hash-cpp-all = 24393106ee6c6e02b4dbd1b66a2f7fdc
```

EdmondsKarp.h

Description: Flow algorithm with guaranteed complexity $O(VE^2)$. To get edge flow values, compare capacities before and after, and take the positive values only.
Usage: unordered_map<int, T> graph;
graph[a][b] += c; //adds edge from a to b with capacity c, use "+=" NOT "="

33 lines

```cpp
template<class T> T edmondsKarp(vector<unordered_map<int, T
    ↪>> &graph, int source, int sink) {
  assert(source != sink);
  T flow = 0;
  vector<int> par(graph.size()), q = par;
  for (;;) {
    fill(par.begin(),par.end(), -1);
    par[source] = 0;
    int ptr = 1;
    q[0] = source;
    for (int i = 0; i < ptr; ++i) {
      int x = q[i];
      for (pair<int,int> e : graph[x]) {
        if (par[e.first] == -1 && e.second > 0) {
          par[e.first] = x;
          q[ptr++] = e.first;
          if (e.first == sink) goto out;
        }
      }
    }
    return flow;
out:
    T inc = numeric_limits<T>::max();
    for (int y = sink; y != source; y = par[y])
      inc = min(inc, graph[par[y]][y]);
    flow += inc;
    for (int y = sink; y != source; y = par[y]) {
      int p = par[y];
      if ((graph[p][y] -= inc) <= 0) graph[p].erase(y);
      graph[y][p] += inc;
    }
  }
};
// hash-cpp-all = 61d8900b275a8485d1f54c130eee76fa
```

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

```cpp
// hash-cpp-all = d41d8cd98f00b204e9800998ecf8427e
```

MinCostMaxFlow.h

Description: Min-cost max-flow. cap[i][j] != cap[j][i] is allowed; double edges are not.
Time: Approximately $\mathcal{O}\left(E^2\right)$ faster than Kactl's on practice

<bits/extc++.h> // don't forget! 71 lines

```cpp
template <typename flow_t = int, typename cost_t = long
    ↪long>
struct MCMF_SSPA { // hash-cpp-1
  int N;
  vector<vector<int>> adj;
  struct edge_t { int dest; flow_t cap; cost_t cost; };
  vector<edge_t> edges;
  vector<char> seen;
  vector<cost_t> pi;
  vector<int> prv;
  explicit MCMF_SSPA(int N_) : N(N_), adj(N), pi(N, 0), prv
      ↪(N) {}
  void addEdge(int from, int to, flow_t cap, cost_t cost) {
    assert(cap >= 0);
    int e = int(edges.size());
    edges.emplace_back(edge_t{to, cap, cost});
    edges.emplace_back(edge_t{from, 0, -cost});
    adj[from].push_back(e);
    adj[to].push_back(e+1);
  }
  const cost_t INF_COST = numeric_limits<cost_t>::max() /
      ↪4;
  const flow_t INF_FLOW = numeric_limits<flow_t>::max() /
      ↪4;
  vector<cost_t> dist;
  __gnu_pbds::priority_queue<pair<cost_t, int>> q;
  vector<typename decltype(q)::point_iterator> its;
// hash-cpp-1 = 8aca97b902d3c8e2ff81879aff6726b7
  void path(int s) { // hash-cpp-2
    dist.assign(N, INF_COST);
    dist[s] = 0;
    its.assign(N, q.end());
    its[s] = q.push({0, s});
    while (!q.empty()) {
      int i = q.top().second; q.pop();
      cost_t d = dist[i];
      for (int e : adj[i]) {
        if (edges[e].cap) {
          int j = edges[e].dest;
          cost_t nd = d + edges[e].cost;
          if (nd < dist[j]) {
            dist[j] = nd;
            prv[j] = e;
```

```
          if (its[j] == q.end()) its[j] = q.push({-(dist[
            ↪j] - pi[j]), j});
          else q.modify(its[j], {-(dist[j] - pi[j]), j});
        }
      }
    }
    swap(pi, dist);
  } // hash-cpp-2 = e0e5e63209e5bf3bf43cf2446879454e
  pair<flow_t, cost_t> maxflow(int s, int t) { // hash-cpp
    ↪-3
    assert(s != t);
    flow_t totFlow = 0; cost_t totCost = 0;
    while (path(s), pi[t] < INF_COST) {
      flow_t curFlow = numeric_limits<flow_t>::max();
      for (int cur = t; cur != s; ) {
        int e = prv[cur];
        int nxt = edges[e^1].dest;
        curFlow = min(curFlow, edges[e].cap);
        cur = nxt;
      }
      totFlow += curFlow;
      totCost += pi[t] * curFlow;
      for (int cur = t; cur != s; ) {
        int e = prv[cur];
        int nxt = edges[e^1].dest;
        edges[e].cap -= curFlow;
        edges[e^1].cap += curFlow;
        cur = nxt;
      }
    }
    return {totFlow, totCost};
  } // hash-cpp-3 = f023f1f510c6212c3225362b96a23efc
};
```

StoerWagner.h

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

30 lines

```
pair<int, vector<int>> GetMinCut(vector<vector<int>> &
  ↪weights) {
  int N = weights.size();
  vector<int> used(N), cut, best_cut;
  int best_weight = -1;
  for (int phase = N-1; phase >= 0; phase--) { // hash-cpp
    ↪-1
    vector<int> w = weights[0], 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) 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 (best_weight == -1 || w[k] < best_weight) {
          best_cut = cut;
          best_weight = w[k];
        }
      } else {
          for (int j = 0; j < N; ++j)
            w[j] += weights[k][j];
        added[k] = true;
      }
    }
  } // hash-cpp-1 = 134b05ab04bdf6f5735abb5acd44401c
  return {best_weight, best_cut};
}
```

GomoryHu.h

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

"PushRelabel.h" 13 lines

```
typedef array<lint, 3> Edge;
vector<Edge> gomoryHu(int N, vector<Edge> ed) {
    vector<Edge> tree;
    vector<int> par(N);
    for(int i = 1; i < N; ++i) {
        PushRelabel D(N); // Dinitz/HLPP also works
        for(auto &t : ed) D.addEdge(t[0], t[1], t[2], t[2])
          ↪;
        tree.push_back({i, par[i], D.calc(i, par[i])});
        for(int j = i+1; j < N; ++j)
            if (par[j] == par[i] && D.leftOfMinCut(j)) par[
              ↪j] = i;
    }
    return tree;
} // hash-cpp-all = ce9ee3ffe1cbfc1f8bc95ae4391a7b0c
```

7.3 Matching

HopcroftKarp.h

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

35 lines

```
using vi = vector<int>;
bool dfs(int a, int L, const vector<vi> &g, vi &btoa, vi &A
  ↪, vi &B) { // hash-cpp-1
    if (A[a] != L) return 0;
    A[a] = -1;
    for(auto &b : g[a]) if (B[b] == L + 1) {
        B[b] = 0;
        if (btoa[b] == -1 || dfs(btoa[b], L+1, g, btoa, A,
          ↪B))
            return btoa[b] = a, 1;
    }
    return 0;
} // hash-cpp-1 = 2beb8de565a17955202792704cbbe58b
int hopcroftKarp(const vector<vi> &g, vi &btoa) { // hash-
  ↪cpp-2
    int res = 0;
    vector<int> A(g.size()), B(btoa.size()), cur, next;
    for (;;) {
        fill(A.begin(), A.end(), 0), fill(B.begin(), B.end
          ↪(), 0);
        cur.clear();
        for(auto &a : btoa) if (a != -1) A[a] = -1;
        for (int a = 0; a < g.size(); ++a) if (A[a] == 0)
          ↪cur.push_back(a);
        for (int lay = 1;; ++lay) {
            bool islast = 0; next.clear();
            for(auto &a : cur) for(auto &b : g[a]) {
                if (btoa[b] == -1) B[b] = lay, islast = 1;
                else if (btoa[b] != a && !B[b])
                    B[b] = lay, next.push_back(btoa[b]);
            }
            if (islast) break;
            if (next.empty()) return res;
            for(auto &a : next) A[a] = lay;
            cur.swap(next);
        }
        for(int a = 0; a < g.size(); ++a)
            res += dfs(a, 0, g, btoa, A, B);
    }
} // hash-cpp-2 = 5cb1049d4e0be6f78f7071cea290845d
```

MaxBipartiteMatching.h

Description: Fast Kuhn! Simple maximum cardinality bipartite matching algorithm. Fast and reliable maximum cardinality matching solver, better than DFSMatching and sometimes even faster than hopcroftKarp (Crazy heuristic huh). This implementation has got an $O(n^2)$ worst case on a sparse graph. Shuffling the edges and vertices ordering might fix it. Good Luck. $R[i]$ will be the match for vertex i on the right side, or -1 if it's not matched. $L[i]$ will be the match for vertex i on the left side.
Time: $\mathcal{O}(VE)$ worst case with shuffling I guess

38 lines

```
struct BipartiteMatcher {
    vector<vector<int>> edges;
    vector<int> L, R, seen;
    BipartiteMatcher(int n, int m) : edges(n), L(n, -1), R(
      ↪m, -1), seen(n) {}
    void addEdge(int a, int b) { edges[a].push_back(b); }
    void improve() {
        mt19937 rng(chrono::steady_clock::now().
          ↪time_since_epoch().count());
        for (int i = 0; i < edges.size(); ++i)
            shuffle(edges[i].begin(), edges[i].end(), rng);
    }
    bool find(int v) {
        if (seen[v]) return false;
        seen[v] = true;
        for (int u : edges[v])
            if (R[u] == -1) {
                L[v] = u, R[u] = v;
                return true;
            }
        for (int u : edges[v])
            if (find(R[u])) {
                L[v] = u, R[u] = v;
                return true;
            }
        return false;
    }
    int maxMatching() {
        int ok = true;
        while (ok--) {
            fill(seen.begin(), seen.end(), 0);
            for (int i = 0; i < (int)L.size(); ++i)
                if (L[i] == -1) ok |= find(i);
        }
        int ret = 0;
        for (int i = 0; i < L.size(); ++i)
            ret += (L[i] != -1);
        return ret;
    }
}; // hash-cpp-all = 624aaeb76e6f9b872987424e9f7fd578
```

WeightedMatching.h

Description: Given array of (possibly negative) costs to complete N jobs w/ M workers ($N \leq M$), finds min cost to complete all jobs s.t. each worker is assigned to at most one job. Takes cost[N][M], where cost[i][j] = cost for i-th job to be completed by j-th worker and returns (min cost, match), where match[i] = worker assigned to i-th job. Negate costs for max cost.

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

31 lines

```
pair<int, vector<int>> hungarian(const vector<vector<int>>
    ↪&a) {
  if (a.empty()) return {0, {}};
  int n = a.size() + 1, m = a[0].size() + 1;
  vector<int> u(n), v(m), p(m), ans(n - 1);
  for(int i = 1; i < n; ++i) {
    p[0] = i;
    int j0 = 0; // add "dummy" worker 0
    vector<int> dist(m, INT_MAX), pre(m, -1);
    vector<bool> done(m + 1);
    do { // dijkstra
      done[j0] = true;
      int i0 = p[j0], j1, delta = INT_MAX;
      for(int j = 1; j < m; ++j) if (!done[j]) {
        auto cur = a[i0-1][j-1] - u[i0] - v[j];
        if (cur < dist[j]) dist[j] = cur, pre[j] = j0;
        if (dist[j] < delta) delta = dist[j], j1 = j;
      }
      for(int j = 0; j < m; ++j) {
        if (done[j]) u[p[j]] += delta, v[j] -= delta;
        else dist[j] -= delta;
      }
      j0 = j1;
    } while (p[j0]);
    while (j0) { // update alternating path
      int j1 = pre[j0];
      p[j0] = p[j1], j0 = j1;
    }
  }
  for(int j = 1; j < m; ++j) if (p[j]) ans[p[j]-1] = j-1;
  return {-v[0], ans}; // min cost
} // hash-cpp-all = 7a23922b36ffbdfed5d138657d5cdc34
```

GeneralMatching.h

Description: Maximum Matching for general graphs (undirected and non bipartite) using Edmond's Blossom Algorithm.

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

68 lines

```
struct blossom_t {
    int t, n; // 1-based indexing!!
    vector<vector<int>> edges;
    vector<int> seen, parent, og, match, aux, Q;
    blossom_t(int _n) : n(_n), edges(n+1), seen(n+1),
        parent(n+1), og(n+1), match(n+1), aux(n+10), t(0)
            ↪{}
    void addEdge(int u, int v) {
        edges[u].push_back(v);
        edges[v].push_back(u);
    }
    void augment(int u, int v) {
        int pv = v, nv; // flip states of edges on u-v path
        do {
            pv = parent[v]; nv = match[pv];
            match[v] = pv; match[pv] = v;
            v = nv;
        } while(u != pv);
    }
    int lca(int v, int w) { // find LCA in O(dist)
        ++t;
        while (1) {
            if (v) {
                if (aux[v] == t) return v; aux[v] = t;
                v = og[parent[match[v]]];
            }
            swap(v, w);
        }
    }
    void blossom(int v, int w, int a) {
        while (og[v] != a) {
            parent[v] = w; w = match[v]; // go other way
                ↪around cycle
            if(seen[w] == 1) Q.push_back(w), seen[w] = 0;
            og[v] = og[w] = a;      // merge into supernode
            v = parent[w];
        }
    }
    bool bfs(int u) {
        for (int i = 1; i <= n; ++i) seen[i] = -1, og[i] =
            ↪i;
        Q = vector<int>(); Q.push_back(u); seen[u] = 0;
        for(int i = 0; i < Q.size(); ++i) {
            int v = Q[i];
            for(auto &x : edges[v]) {
                if (seen[x] == -1) {
                    parent[x] = v; seen[x] = 1;
                    if (!match[x]) return augment(u, x),
                        ↪true;
                    Q.push_back(match[x]); seen[match[x]] =
                        ↪ 0;
                } else if (seen[x] == 0 && og[v] != og[x])
                    ↪{
                    int a = lca(og[v], og[x]);
                    blossom(x, v, a); blossom(v, x, a);
                }
            }
        }
        return false;
    }
    int solve() {
        int ans = 0; // find random matching (not necessary
            ↪,
        vector<int> V(n-1); iota(V.begin(), V.end(), 1);//
            ↪constant improvement)
        shuffle(V.begin(), V.end(), mt19937(0x94949));
        for(auto &x : V) if(!match[x])
            for(auto &y : edges[x]) if (!match[y]) {
                match[x] = y, match[y] = x;
                ++ans; break;
            }
        for (int i = 1; i <= n; ++i)
            if (!match[i] && bfs(i)) ++ans;
        return ans;
    }
}; // hash-cpp-all = 0b82eef228c4effc781724d4c3d8c468
```

MaximumIndependentSet.h

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

1 lines

```
// hash-cpp-all = d41d8cd98f00b204e9800998ecf8427e
```

MinimumVertexCover.h

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

"MaxBipartiteMatching.h" 19 lines

```
vector<int> cover(BipartiteMatcher& B, int n, int m) {
    int res = B.maxMatching();
    vector<bool> lfound(n, true), seen(m);
    for(int &it : B.R) if (it != -1) lfound[it] = false;
    vector<int> q, cover;
    for(int i = 0; i < n; ++i) if (lfound[i]) q.push_back(i
        ↪);
    for(int i = 0; i < q.size(); ++i) {
        int v = q[i];
        lfound[v] = true;
        for (int e : B.edges[v]) if (!seen[e] && B.R[e] !=
            ↪-1) {
            seen[e] = true;
            q.push_back(B.R[e]);
        }
    }
    for(int i = 0; i < n; ++i) if (!lfound[i]) cover.
        ↪push_back(i);
    for(int i = 0; i < m; ++i) if (seen[i]) cover.push_back
        ↪(n+i);
    assert(cover.size() == res);
    return cover;
} // hash-cpp-all = fbfea38b27386ccb5563ec189e292e28
```

MinimumEdgeCover.h

Description: Finds a minimum edge cover in a bipartite graph. The size is the same as the number of vertices minus the size of a maximum matching. The mark vector represents who the vertices of set B has an edge to.

Usage: `vector<int> mark(n+m, -1);`
`auto cover = minEdgeCover(g, mark);`

"MaxBipartiteMatching.h" 12 lines

```
vector<pair<int,int>> minEdgeCover(BipartiteMatcher &g,
    ↪vector<int> &mark) {
    int maxMatching = g.maxMatching();
    vector<pair<int,int>> cover;
    for (int i = 0; i < g.L.size(); ++i) {
        if (g.L[i] >= 0) cover.push_back({i, g.L[i]});
        else if (g.edges[i].size()) cover.push_back({i, g.
            ↪edges[i][0]});
    }
    for (int i = 0; i < g.R.size(); ++i)
        if (g.R[i] == -1 && mark[i] >= 0)
            cover.push_back({mark[i], i});
    return cover;
} // hash-cpp-all = 6cb7da971b3ff6fde447445c1aba7b6b
```

MinimumPathCover.h

Description: Finds a minimum vertex-disjoint path cover in a dag. The size is the same as the number of vertices minus the size of a maximum matching.

"MaxBipartiteMatching.h" 15 lines

```
vector<vector<int>> minPathCover(BipartiteMatcher &g) {
    int how_many = g.edges.size() - g.maxMatching();
    vector<vector<int>> paths;
    for (int i = 0; i < g.edges.size(); ++i)
        if (g.R[i] == -1) {
            vector<int> path = {i};
            int cur = i;
            while (g.L[cur] >= 0) {
                cur = g.L[cur];
```

```
            path.push_back(cur);
        }
        paths.push_back(path);
    }
    return paths;
} // hash-cpp-all = dbe138fcae3059d083e98c399dafc45b
```

7.4 DFS algorithms

CentroidDecomposition.cpp

Description: Divide and Conquer on Trees.

34 lines

```
struct centroid_t {
    vector<bool> seen;
    vector<int> sz, level, par_tree, closest;
    vector<vector<int>> edges, dist, parent;
    centroid_t(vector<vector<int>> &e, int n) : edges(e),
       ↪seen(n),
        sz(n), level(n), par_tree(n), closest(n, INT_MAX/2)
           ↪,
        dist(n, vector<int>(20)), parent(n, vector<int>(20)
           ↪) {
            build(0, -1); }
    void dfs(int v, int p, int parc, int lvl) {
        sz[v] = 1;
        parent[v][lvl] = parc;
        dist[v][lvl] = 1 + dist[p][lvl];
        for (int u : edges[v]) {
            if (u == p || seen[u]) continue;
            dfs(u, v, parc, lvl);
            sz[v] += sz[u];
        }
    }
    int get_centroid(int v, int p, int tsz) {
        for (int u : edges[v])
            if (!seen[u] && u != p && sz[u] > tsz/2)
                return get_centroid(u, v, tsz);
        return v;
    }
    void build(int v, int p, int lvl = 0) {
        dfs(v, -1, p, lvl);
        int x = get_centroid(v, v, sz[v]);
        seen[x] = 1;
        par_tree[x] = p;
        level[x] = 1 + lvl;
        for (int u : edges[x])
            if (!seen[u]) build(u, x, 1 + lvl);
    }
}; // hash-cpp-all = 114b96449cf7cb6efe28752355d4ea5b
```

Tarjan.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: `cnt_of[i] holds the component index of a node (a component only has edges to components with lower index). ncnt will contain the number of components.`

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

28 lines

```
struct tarjan_t {
    int n, ncnt = 0, time = 0;
    vector<vector<int>> edges;
    vector<int> preorder_of, cnt_of, order, stack_t;
    tarjan_t(int n): n(n), edges(n), preorder_of(n), cnt_of
       ↪(n,-1) {}
    int dfs(int u) {
        int reach = preorder_of[u] = ++time, v;
        stack_t.push_back(u);
        for (int v : edges[u]) if (cnt_of[v] == -1)
            reach = min(reach, preorder_of[v]?:dfs(v));
        if (reach == preorder_of[u]) {
            do {
                v = stack_t.back();
                stack_t.pop_back();
                order.push_back(v);
                cnt_of[v] = ncnt;
            } while (v != u);
            ++ncnt;
        }
        return preorder_of[u] = reach;
    }
    void scc() {
        time = ncnt = 0;
        for (int i = 0; i < (int)edges.size(); ++i)
            if (cnt_of[i] == -1) dfs(i);
    }
};
// hash-cpp-all = 4c2a4b2d70bec88795d61d404585c14a
```

BiconnectedComponents.h

Description: Finds all biconnected components in an undirected graph, and runs a callback for the edges in each. In a biconnected component there are at least two distinct paths between any two nodes. Note that a node can be in several components. An edge which is not in a component is a bridge, i.e., not part of any cycle.

Usage: `int eid = 0; ed.resize(N);`
`for each edge (a,b) {`
`ed[a].emplace_back(b, eid);`
`ed[b].emplace_back(a, eid++);`
`}`

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

36 lines

```
typedef vector<vector<pair<int,int>>> vii;
vector<int> num, st;
vii ed;
int Time;
int dfs(int at, int par,vector<vector<int>> &comps) {
  int me = num[at] = ++Time, e, y, top = me;
  for (auto &pa : ed[at]) if (pa.second != par) {
    tie(y, e) = pa;
    if (num[y]) {
      top = min(top, num[y]);
      if (num[y] < me) st.push_back(e);
    } else {
      int si = st.size();
      int up = dfs(y, e, comps);
      top = min(top, up);
      if (up == me) {
        st.push_back(e);
        comps.push_back(vector<int>());
        for(int i = st.size()-1;i >= si; --i)
          comps[comps.size()-1].push_back(st[i]);
        st.resize(si);
        cont_comp++;
      }
      else if (up < me){ st.push_back(e);}
      else { cont_comp++; comps.push_back({e});/* e is a
         ↪bridge */ }
    }
  }
  return top;
}
vector<vector<int>> solve() { // returns components
  vector<vector<int>> comps; // and its edges ids
  num.assign(ed.size(), 0);
  for (int i = 0; i < ed.size(); ++i)
    if (!num[i]) dfs(i, -1, comps);
  return comps;
} // hash-cpp-all = 0dcf4650df5778978094af058e78fbdf
```

2sat.h

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

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

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

49 lines

```
struct TwoSat {
  int N;
  vector<vector<int>> gr;
  vector<int> values; // 0 = false, 1 = true
  TwoSat(int n = 0) : N(n), gr(2*n) {}
  int add_var() { // (optional)
    gr.emplace_back();
    gr.emplace_back();
    return N++;
  }
  void either(int f, int j) { // hash-cpp-1
    f = max(2*f, -1-2*f);
    j = max(2*j, -1-2*j);
    gr[f].push_back(j^1);
    gr[j].push_back(f^1);
  } // hash-cpp-1 = 516db0b7f02565e6773ff37b6319169f
  void set_value(int x) { either(x, x); }
  void at_most_one(const vector<int>& li) { // (optional)
     ↪// hash-cpp-2
    if (li.size() <= 1) return;
    int cur = ~li[0];
    for (int i = 2; i < li.size(); ++i) {
      int next = add_var();
      either(cur, ~li[i]);
      either(cur, next);
      either(~li[i], next);
      cur = ~next;
    }
    either(cur, ~li[1]);
  } // hash-cpp-2 = d1cd651b7bb790d3aba3c4895427d962
  vector<int> val, comp, z; int time = 0;
  int dfs(int i) { // hash-cpp-3
    int low = val[i] = ++time, x; z.push_back(i);
    trav(e, gr[i]) if (!comp[e])
      low = min(low, val[e] ?: dfs(e));
    if (low == val[i]) do {
      x = z.back(); z.pop_back();
      comp[x] = low;
      if (values[x>>1] == -1)
        values[x>>1] = x&1;
    } while (x != i);
    return val[i] = low;
  } // hash-cpp-3 = af17664bf233022868bbb5c5ec1140fe
  bool solve() { // hash-cpp-4
    values.assign(N, -1);val.assign(2*N, 0); comp = val;
    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]) return 0;
    return 1;
  } // hash-cpp-4 = 49f5aec465cba73979ba291353751689
};
```

Cycles.h

Description: Cycle Detection (Detects a cycle in a directed or undirected graph.)
Time: $\mathcal{O}(V)$

25 lines

```
bool detectCycle(vector<vector<int>> &edges, bool
   ↪undirected) {
    vector<int> seen(n, 0), parent(n), stack_t;
    for (int i = 0; i < edges.size(); ++i) {
        if (seen[i] == 2) continue;
        stack_t.push_back(i);
        while(!stack_t.empty()) {
            int u = stack_t.back();
            stack_t.pop_back();
            if (seen[u] == 1) seen[u] = 2;
            else {
                stack_t.push_back(u);
                seen[u] = 1;
                for (int w : edges[u]) {
                    if (seen[w] == 0) {
                        parent[w] = u;
                        stack_t.push_back(w);
                    }
                    else if (seen[w] == 1 && (!undirected
                       ↪|| w != parent[u]))
                        return true;
                }
            }
        }
    }
}
// hash-cpp-all = 7ff93a874ccce87f8fcc944ce4adc144
```

7.5 Heuristics

MaximalCliques.h

Description: Runs a callback for all maximal cliques in a graph (given as a symmetric bitset matrix; self-edges not allowed). Possible optimization: on the top-most recursion level, ignore 'cands', and go through nodes in order of increasing degree, where degrees go down as nodes are removed.
Time: $\mathcal{O}\left(3^{n/3}\right)$, much faster for sparse graphs

12 lines

```
typedef bitset<128> B;
template<class F>
void cliques(vector<B> &eds, F f, B P = ~B(), B X={}, B R
   ↪={}) { // hash-cpp-1
  if (!P.any()) { if (!X.any()) f(R); return; }
  auto q = (P | X)._Find_first();
  auto cands = P & ~eds[q];
  for(int i = 0; i < eds.size(); ++i) if (cands[i]) {
    R[i] = 1;
    cliques(eds, f, P & eds[i], X & eds[i], R);
    R[i] = P[i] = 0; X[i] = 1;
  }
} // hash-cpp-1 = 1dc1acd20ad3a69c17c07ce840d575ca
```

MaximumClique.h

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

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

49 lines

```
typedef vector<bitset<200>> vb;
struct Maxclique {
  double limit = 0.025, pk = 0;
  struct Vertex { int i, d = 0; };
  typedef vector<Vertex> vv;
  vb e;
  vv V;
  vector<vector<int>> C;
  vector<int> qmax, q, S, old;
  void init(vv& r) {
    for(auto& v : r) v.d = 0;
    for(auto& v : r) for(auto& j : r) v.d += e[v.i][j.i];
    sort(r.begin(), r.end(), [](auto a, auto b) { return a.
       ↪d > b.d; });
    int mxD = r[0].d;
    for(int i = 0; i < r.size(); ++i) r[i].d = min(i, mxD)
       ↪+ 1;
  }
  void expand(vv& R, int lev = 1) {
    S[lev] += S[lev - 1] - old[lev];
    old[lev] = S[lev - 1];
    while (R.size()) {
      if (q.size() + R.back().d <= qmax.size()) return;
      q.push_back(R.back().i);
      vv T;
      for(auto& v : R) if (e[R.back().i][v.i]) T.push_back
         ↪({v.i});
      if (T.size()) {
        if (S[lev]++ / ++pk < limit) init(T);
        int j = 0, mxk = 1, mnk = max(qmax.size() - q.size
           ↪() + 1, 1);
        C[1].clear(), C[2].clear();
        for(auto& v : T) {
          int k = 1;
          auto f = [&](int i) { return e[v.i][i]; };
          while (any_of(C[k].begin(), C[k].end(), f)) k++;
          if (k > mxk) mxk = k, C[mxk + 1].clear();
          if (k < mnk) T[j++].i = v.i;
          C[k].push_back(v.i);
        }
        if (j > 0) T[j - 1].d = 0;
        for(int k = mnk; k <= mxk; ++k) for(auto& i : C[k])
          T[j].i = i, T[j++].d = k;
        expand(T, lev + 1);
      } else if (q.size() > qmax.size()) qmax = q;
      q.pop_back(), R.pop_back();
    }
  }
  vector<int> maxClique() { init(V), expand(V); return qmax
     ↪; }
  Maxclique(vb conn) : e(conn), C(sz(e)+1), S(C.size()),
     ↪old(S) {
    for(int i = 0; i < e.size(); ++i) V.push_back({i});
  }
}; // hash-cpp-all = 0fb921df39bfda2151477954b30fd256
```

Cycle-Counting.cpp

Description: Counts 3 and 4 cycles

55 lines

```
#define P 1000000007
#define N 110000
int n, m;
vector<int> go[N], lk[N];
int w[N], deg[N], pos[N], id[N];
int circle3(){ // hash-cpp-1
  int ans=0;
  for (int i = 1; i <= n; i++)
    w[i]=0;

  for (int x = 1; x <= n; x++) {
    for(int y:lk[x])w[y]=1;

    for(int y:lk[x])for(int z:lk[y])if(w[z]){
      ans=(ans+go[x].size()+go[y].size()+go[z].size()-6)%P;
    }

    for(int y:lk[x])w[y]=0;
  }
  return ans;
} // hash-cpp-1 = 719dcec935e20551fd984c12c3bfa3ba
int circle4(){ // hash-cpp-2
  for (int i = 1; i <= n; i++)
    w[i]=0;
  int ans=0;
  for (int x = 1; x <= n; x++) {
    for(int y:go[x])for(int z:lk[y])if(pos[z]>pos[x]){
      ans=(ans+w[z])%P;
      w[z]++;
    }
    for(int y:go[x])for(int z:lk[y])w[z]=0;
  }
  return ans;
} // hash-cpp-2 = 39b3aaf47e9fdc4dfff3fdfdf22d3a8e
inline bool cmp(const int &x,const int &y){
  return deg[x]<deg[y];
}
void init() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++)
    deg[i] = 0, go[i].clear(), lk[i].clear();;
  while (m--) {
    int a,b;
    scanf("%d%d",&a,&b);
    deg[a]++;deg[b]++;
    go[a].push_back(b);go[b].push_back(a);
  }
  for (int i = 1; i <= n; i++)
    id[i] = i;
  sort(id+1,id+1+n,cmp);
  for (int i = 1; i <= n; i++) pos[id[i]]=i;
  for (int x = 1; x <= n; x++)
    for(int y:go[x])
      if(pos[y]>pos[x])lk[x].push_back(y);
}
```

7.6 Trees

Tree.h

Description: Structure that handles tree's, can find its diameter points, diameter length, center vertices, etc. Consider using two BFS's if constraints are too tight.

41 lines

```
struct tree_t {
    int n;
    vector<vector<int>> edges;
    vector<int> parent, dist;
    pair<int, int> center, diameter;
    tree_t(vector<vector<int>> g) : n(g.size()), parent(n),
       ↪ dist(n) {
        edges = g;
        diameter = {1, 1};
    }
    void dfs(int v, int p) {
```

```
        for (int u : edges[v]) {
            if (u == p) continue;
            parent[u] = v;
            dist[u] = dist[v] + 1;
            dfs(u, v);
        }
    }
    pair<int,int> find_diameter() { // diameter start->
       ↪finish point
        parent[0] = -1; dist[0] = 0;
        dfs(0, 0);
        for (int i = 0; i < n; ++i)
            if (dist[i] > dist[diameter.first]) diameter.
               ↪first = i;
        parent[diameter.first] = -1;
        dist[diameter.first] = 0;
        dfs(diameter.first, diameter.first);
        for (int i = 0; i < n; ++i)
            if (dist[i] > dist[diameter.second]) diameter.
               ↪second = i;
        return diameter;
    }
    int get_diameter() { // length of diameter
        diameter = find_diameter();
        return dist[diameter.second];
    }
    pair<int,int> find_center() {
        diameter = find_diameter();
        int k = diameter.second, length = dist[diameter.
           ↪second];
        for (int i = 0; i < length/2; ++i) k = parent[k];
        if (length%2) return center = {k, parent[k]}; //
           ↪two centers
        else return center = {k, -1}; // k is the only
           ↪center of the tree
    }
}; // hash-cpp-all = efc11e16a1306de29644c4ce6907baba
```

LCA.cpp

Description: Solve lowest common ancestor queries using binary jumps. Can also find the distance between two nodes.
Time: $\mathcal{O}(N+Q)$

53 lines

```
struct lca_t {
    int logn{0}, preorderpos{0};
    vector<int> invpreorder, height;
    vector<vector<int>> jump_binary, edges;
    lca_t(int n, vector<vector<int>>& adj) :
    edges(adj), height(n), invpreorder(n) {
        while((1<<(logn+1)) <= n) ++logn;
        jump_binary.assign(n+1, vector<int>(logn+1, 0));
        dfs(0, -1, 0);
    }
    void dfs(int v, int p, int h) {
        invpreorder[v] = preorderpos++;
        height[v] = h;
        jump_binary[v][0] = p < 0 ? v : p;
        for (int l = 1; l <= logn; ++l)
            jump_binary[v][l] = jump_binary[jump_binary[v][
               ↪l-1]][l-1];
        for (int u : edges[v]) {
            if (u == p) continue;
            dfs(u, v, h + 1);
        }
    }
    int climb(int v, int dist) {
        for (int l = 0; l <= logn; ++l)
            if (dist & (1 << l)) v = jump_binary[v][l];
        return v;
    }
    int query(int a, int b) {
        if (height[a] < height[b]) swap(a, b);
        a = climb(a, height[a] - height[b]);
        if (a == b) return a;
        for (int l = logn; l >= 0; --l)
            if (jump_binary[a][l] != jump_binary[b][l])
                a = jump_binary[a][l], b = jump_binary[b][l
                   ↪];
        return jump_binary[a][0];
    }
    T dist(int a, int b) {
        return height[a] + height[b] - 2 * height[query(a,b
           ↪)];
    }
    bool is_parent(int p, int v) {
        if (height[p] > height[v]) return false;
        return p == climb(v, height[v] - height[p]);
    }
    bool on_path(int x, int a, int b) {
        int v = query(a, b);
        return is_parent(v, x) && (is_parent(x, a) ||
           ↪is_parent(x, b));
    }
    int get_kth_on_path(int a, int b, int k) {
        int v = query(a, b);
        int x = height[a] - height[v], y = height[b] -
           ↪height[v];
        if (k < x) return climb(a, k);
        else return climb(b, x + y - k);
    }
}; // hash-cpp-all = 8032d895d0e237e6b1b9bd2770247dd2
```

LCA.h

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

"../data-structures/RMQ.h" 21 lines

```
struct lca_t {
    int T = 0;
    vector<int> time, path, walk, depth;
    rmq_t<int> rmq;
    lca_t(vector<vector<int>> &edges) : time(edges.size()),
    depth(edges.size()), rmq((dfs(edges,0,-1), walk)) {}
    void dfs(vector<vector<int>> &edges, int v, int p) {
        time[v] = T++;
        for(int u : edges[v]) {
            if (u == p) continue;
            depth[u] = depth[v] + 1;
            path.push_back(v), walk.push_back(time[v]);
            dfs(edges, u, v);
        }
    }
    int lca(int a, int b) {
        if (a == b) return a;
        tie(a, b) = minmax(time[a], time[b]);
        return path[rmq.query(a, b-1).first];
    }
}; // hash-cpp-all = ff2c92ceb8c59b4dba74f82329188ba9
```

CompressTree.h

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

"LCA.h" 20 lines

```
vector<pair<int,int>> compressTree(lca_t &lca, const vector
   ↪<int>& subset) {
  static vector<int> rev; rev.resize(lca.time.size());
  vector<int> li = subset, &T = lca.time;
  auto cmp = [&](int a, int b) { return T[a] < T[b]; };
  sort(li.begin(), li.end(), cmp);
  int m = li.size()-1;
  for (int i = 0; i < m; ++i) {
    int a = li[i], b = li[i+1];
    li.push_back(lca.lca(a, b));
  }
  sort(li.begin(), li.end(), cmp);
  li.erase(unique(li.begin(), li.end()), li.end());
  for (int i = 0; i < li.size(); ++i) rev[li[i]] = i;
  vector<pair<int,int>> ret = {{0, li[0]}};
  for (int i = 0; i < li.size()-1; ++i) {
    int a = li[i], b = li[i+1];
    ret.emplace_back(rev[lca.lca(a, b)], b);
  }
  return ret;
} // hash-cpp-all = 1543dc6b0d9830771f607897d8e46906
```

Heavylight.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. Code supports commutative segtree modifications/-queries on paths, edges and subtrees. Takes as input the full adjacency list with pairs of (vertex, value). USE_EDGES being true means that values are stored in the edges and are initialized with the adjacency list, otherwise values are stored in the nodes and are initialized to the T defaults value.
Time: $\mathcal{O}\left((\log N)^2\right)$

"../data-structures/SegTree.h" 72 lines

```
typedef vector<vector<pair<int,int>>> adj;
template<typename T, bool USE_EDGES>
struct HLD {
  int t{0}, n;
  vector<int> in, par, chain, sz, dep;
  vector<T> val;
  tree_t<T> seg;
  HLD(adj &g, int r = 0) : n(g.size()), par(n,-1),
  chain(n,-1), dep(n), in(n), sz(n),  val(n){
    par[r] = chain[r] = r;
    dfs_sz(g, r), dfs_hld(g, r);
    seg = {val};
  }
  void f(T &a, T b){ a = max(a,b); }
  T query(int a, int b) { return seg.query(a, b+1); }
  void update(int a, T value) { seg.update(a, value); }
  void update(int a, int b, T value) { }
  void dfs_sz(adj &g, int u) {
    sz[u]++;
    for(auto &e : g[u]) {
      int v = e.first;
      if(par[v] == -1) {
        par[v] = u; dep[v] = dep[u] + 1;
        dfs_sz(g, v);
        sz[u] += sz[v];
        if (par[u] == g[u][0].first || sz[v] > sz[g[u][0].
           ↪first])
          swap(g[u][0], e);
```

```
    }
  }
}
void dfs_hld(adj &g, int u) {
  in[u] = t++;
  for (auto &e : g[u]) {
    int v = e.first;
    if (chain[v] == -1) {
      if (e == g[u][0]) chain[v] = chain[u];
      else chain[v] = v;
      dfs_hld(g, v);
      if (USE_EDGES) val[in[v]] = e.second;
    }
  }
}
void path(int u, int v, function<void(int,int)> func){
  if (u == v){ return func(in[u],in[u]);}
  for(int e, p; chain[u] != chain[v]; u = p){
    if (dep[chain[u]] < dep[chain[v]]) swap(u,v);
    e = 1, p = chain[u];
    if(u == p) e = 0, p = par[u];
    func(in[chain[u]] + e, in[u]);
  }
  if (in[u] > in[v]) swap(u, v);
  func(in[u] + USE_EDGES, in[v]);
}
void update_path(int u, int v, T value){
  path(u, v, [&](int a,int b){ update(a, b, value); });
}
T query_path(int u, int v) {
  T ans = -(1<<29);
  path(u, v, [&](int a,int b){ f(ans, query(a, b)); });
  return ans;
}
void update_edge(int u, int v, T value) {
  if (dep[u] < dep[v]) u = v;
  update(in[u], value);
}
T query_subtree(int v) {
  return query(in[v] + USE_EDGES, in[v] + sz[v] - 1);
}
void update_subtree(int v, T value) {
  update(in[v] + USE_EDGES, in[v] + sz[v] - 1, value);
}
}; // hash-cpp-all = 2054f4a32912310ed0148842e12e48c6
```

Tree-Isomorphism.h

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

51 lines

```
map<vector<int>, int> delta;
struct tree_t {
  int n;
  pair<int,int> centroid;
  vector<vector<int>> edges;
  vector<int> sz;
  tree_t(vector<vector<int>>& graph) :
    edges(graph), sz(edges.size()) {}
  int dfs_sz(int v, int p) {
    sz[v] = 1;
    for (int u : edges[v]) {
      if (u == p) continue;
      sz[v] += dfs_sz(u, v);
    }
    return sz[v];
  }
  int dfs(int tsz, int v, int p) {
    for (int u : edges[v]) {
      if (u == p) continue;
      if (2*sz[u] <= tsz) continue;
      return dfs(tsz, u, v);
    }
    return centroid.first = v;
  }
  pair<int,int> find_centroid(int v) {
    int tsz = dfs_sz(v, -1);
    centroid.second = dfs(tsz, v, -1);
    for (int u : edges[centroid.first]) {
      if (2*sz[u] == tsz)
        centroid.second = u;
    }
    return centroid;
  }
  int hash_it(int v, int p) {
    vector<int> offset;
    for (int u : edges[v]) {
      if (u == p) continue;
      offset.push_back(hash_it(u, v));
    }
    sort(offset.begin(), offset.end());
    if (!delta.count(offset))
      delta[offset] = int(delta.size());
    return delta[offset];
  }
  lint get_hash(int v = 0) {
    pair<int,int> cent = find_centroid(v);
    lint x = hash_it(cent.first, -1), y = hash_it(cent.
      ↪second, -1);
    if (x > y) swap(x, y);
    return (x << 30) + y;
  }
}; // hash-cpp-all = 92e59fd174d98fae157272b14c6b43ee
```

LineTree.h

Description: Performs a preprocessing to enable querying the maximum/minimum edge weight on any path in a tree in constant time.

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

<RMQ.h> 73 lines

```
struct UF {
    vector<int> parent, size, left, right;
    UF(int n) : parent(n), size(n, 1), left(n), right(n) {
        for (int i = 0; i < n; i++)
            parent[i] = left[i] = right[i] = i;
    }
    int find(int x) {
        return x == parent[x] ? x : parent[x] = find(parent
          ↪[x]);
    }
    pair<int, int> unite(int x, int y) {
        x = find(x);
        y = find(y);
        assert(x != y);
        if (size[x] < size[y]) swap(x, y);
        parent[y] = x;
        size[x] += size[y];
        pair<int,int> result = {right[x], left[y]};
        right[x] = right[y];
        return result;
    }
};
template<typename T> struct linetree_t {
  struct edge_t {
    int u, v; T w;
    edge_t() {}
    edge_t(int a, int b, T c) : u(a), v(b), w(c) {}
    bool operator<(const edge_t &other) const {
      return w < other.w;
    }
  };
  int n;
  const T limit = numeric_limits<T>::min();
  vector<int> index, line;
  vector<edge_t> edges; vector<T> line_w;
  unique_ptr<rmq_t<T>> rmq;
  linetree_t(int _n) : n(_n), index(n) {}
  void addEdge(int from, int to, T weight) {
    edges.emplace_back(from, to, weight);
  }
  void make_tree() {
    sort(edges.begin(), edges.end());
    UF dsu(n);
    vector<int> next_v(n, -1), has_prev(n);
    vector<T> next_w(n, limit);
    for (edge_t& e : edges) {
      pair<int, int> united = dsu.unite(e.u, e.v);
      next_v[united.first] = united.second;
      has_prev[united.second] = 1;
      next_w[united.first] = e.w;
    }
    int start = -1;
    for (int i = 0; i < n; ++i)
      if (!has_prev[i]) {
        start = i;
        break;
      }
    while(start >= 0) {
      line.push_back(start);
      if (next_v[start] >= 0)
        line_w.push_back(next_w[start]);
      start = next_v[start];
    }
    for (int i = 0; i < n; ++i)
      index[line[i]] = i;
    rmq.reset(new RMQ<T>(line_w));
  }
  T query(int a, int b) {
    if (a == b) return limit;
    a = index[a], b = index[b];
    if (a > b) swap(a, b);
    return rmq->query(a-1, b-1).first;
  }
}; // hash-cpp-all = 2054f4a32912310ed0148842e12e48c6b6c12c8dd25d1af05d2db
```

7.6.1 Sqrt Decomposition

HLD generally suffices. If not, here are some common strategies:

- Rebuild the tree after every $\sqrt{N}$ queries.
- Consider vertices with $>$ or $<\sqrt{N}$ degree separately.
- For subtree updates, note that there are $O(\sqrt{N})$ distinct sizes among child subtrees of any vertex.

Block Tree: Use a DFS to split edges into contiguous groups of size $\sqrt{N}$ to $2\sqrt{N}$.

Mo's Algorithm for Tree Paths: Maintain an array of vertices where each one appears twice, once when a DFS enters the vertex (`st`) and one when the DFS exists (`en`). For a tree path $u \leftrightarrow v$ such that `st[u]<st[v]`,

- If u is an ancestor of v, query `[st[u],st[v]]`.
- Otherwise, query `[en[u],st[v]]` and consider $lca(u, v)$ separately.

7.7 Functional Graphs

Lumberjack.h

Description: Called lumberjack technique, solve functional graphs problems for digraphs, it's also pretty good for dp on trees. Consists in go cutting the leaves until there is no leaves, only cycles. For that we keep a processing queue of the leaves, note that during this processing time we go through all the childrens of v before reaching a vertex v, therefore we can compute some infos about the children, like subtree of a given vertex

Usage: `Lumberjack<10010> g; g.init(N); (Be careful with the size of cyles when declared locally!)`

69 lines

```
template<int T> struct Lumberjack {
  int n, numcycle;
  vector<int> subtree, order, par, cycle;
  vector<int> parincycles, idxcycle, sz, st;
  vector<int> depth, indeg, cycles[T];
  vector<bool> seen, incycle, leaf;
  void init(vector<int>& par, vector<int>& indeg){
    init(par.size());
    par = par; indeg = indeg;
  }
  void init(int N) {
    n = N;
    subtree.assign(n, 0);
    seen.assign(n, false);
    sz = st = subtree;
    parincycles = order = par = cycle = sz;
    idxcycle = depth = indeg = sz;
    incycle = leaf = seen;
  }
  void find_cycle(int u){
    int idx= ++numcycle, cur = 0, p = u;
    st[idx] = u;
    sz[idx] = 0;
    cycles[idx].clear();
    while (!seen[u]) {
      seen[u] = incycle[u] = 1;
      parincycles[u] = u;
      cycle[u] = idx;
      idxcycle[u] = cur;
      cycles[idx].push_back(u);
      ++sz[idx];
      depth[u] = 0;
      ++subtree[u];
      u = par[u];
      ++cur;
    }
  }
  void bfs() {
    queue<int> q;
    for (int i = 0; i < n; ++i)
      if (!indeg[i]){
        seen[i] = leaf[i] = true;
        q.push(i);
      }
    while(!q.empty()){
      int v = q.front(); q.pop();
      order.push_back(v);
      ++subtree[v];
      int curpar = par[v];
      indeg[curpar]--;
      subtree[curpar] += subtree[v];
      if(!indeg[curpar]){
        q.push(curpar);
        seen[curpar] = true;
      }
    }
    numcycle = 0;
    for (int i = 0; i < n; ++i)
      if (!seen[i]) find_cycle(i);
    for(int i = order.size()-1; i >= 0; --i){
      int v = order[i], curpar = par[v];
      parincycles[v] = parincycles[curpar];
      cycle[v] = cycle[curpar];
      incycle[v] = false;
      idxcycle[v] = -1;
      depth[v] = 1 + depth[curpar];
    }
  }
}; // hash-cpp-all = 9d55412a8d50b2a2e80e4e50b87887c3
```

Lumberjack2.h

Description: Called lumberjack technique, solve functional graphs problems for graphs, it's also pretty good for dp on trees. Consists in go cutting the leaves until there is no leaves, only cycles. For that we keep a processing queue of the leaves, note that during this processing time we go through all the childrens of v before reaching a vertex v, therefore we can compute some infos about the children, like subtree of a given vertex

79 lines

```
template<int T> struct Lumberjack {
  int n, numcycle;
  vector<int> subtree, order, par, cycle;
  vector<int> parincycles, idxcycle, sz, st;
  vector<int> depth, deg, cycles[T];
  vector<bool> seen, incycle, leaf;
  vector<vector<int>> graph;
  void init(vector<vector<int>>& graph, vector<int>& deg){
    init(graph.size());
    graph = graph; deg = deg;
  }
  void init(int N) {
    n = N;
    subtree.assign(n, 0);
    seen.assign(n, false);
    sz = st = subtree;
    parincycles = order = par = cycle = sz;
    idxcycle = depth = deg = sz;
    incycle = leaf = seen;
    vector<int> adj; graph.assign(n, adj);
  }
  int find_par(int u) {
      for (int v : graph[u]) if (!seen[v])
        return v;
      return -1;
  }
  void find_cycle(int u){
    int idx= ++numcycle, cur = 0, p = u;
    st[idx] = u;
    sz[idx] = 0;
    cycles[idx].clear();
    while (!seen[u]) {
      seen[u] = incycle[u] = true;
      par[u] = find_par(u);
      if(par[u] == -1) par[u] = p;
      parincycles[u] = u;
      cycle[u] = idx;
      idxcycle[u] = cur;
      cycles[idx].push_back(u);
      ++sz[idx];
      depth[u] = 0;
      ++subtree[u];
      u = par[u];
      ++cur;
    }
  }
  void bfs() {
    queue<int> q;
    for (int i = 0; i < n; ++i)
      if (deg[i] == 1){
        seen[i] = leaf[i] = true;
        q.push(i);
      }
    while(!q.empty()){
      int v = q.front(); q.pop();
      order.push_back(v);
      ++subtree[v];
      int curpar = find_par(v);
      deg[curpar]--;
      subtree[curpar] += subtree[v];
      if(deg[curpar] == 1){
        q.push(curpar);
        seen[curpar] = true;
      }
    }
    numcycle = 0;
    for (int i = 0; i < n; ++i)
      if (!seen[i]) find_cycle(i);
    for(int i = order.size()-1; i >= 0; --i){
      int v = order[i], curpar = par[v];
      parincycles[v] = parincycles[curpar];
      cycle[v] = cycle[curpar];
      incycle[v] = false;
      idxcycle[v] = -1;
      depth[v] = 1 + depth[curpar];
    }
  }
};
// hash-cpp-all = ec29159ed9e96acb668f88ac63e4167f
```

7.8 Other

kthShortestPath.h

Description: Find Kth shortest path from s to t.

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

21 lines

```
int getCost(vector<vector<pair<int,int>>> &G, int s, int t,
    ↪ int k) {
    int n = G.size();
    vector<int> dist(n, INF), count(n, 0);
    priority_queue<pair<int,int>, vector<pair<int,int>>,
        ↪greater<pair<int,int>>> Q;
  Q.push({0, s});
  while (!Q.empty() && (count[t] < k)) {
    pair<int,int> v = Q.top();
    int u = v.second, w = v.first;
    Q.pop();
```

```
    if ((dist[u] == INF) || (w > dist[u])) { // remove
       ↪equal paths
      count[u] += 1;
      dist[u] = w;
    }
    if (count[u] <= k)
        for (int x : G[u]) {
        int v = x.first, w = x.second;
        Q.push({dist[u] + w, v});
        }
  }
    return dist[t];
} // hash-cpp-all = b611794901cec100dd9015bce082d108
```

MatrixTreeMST.h

Description: Returns the number of msts in undirected weighted graph using the Matrix Tree theorem.
Time: $\mathcal{O}\left(N^3\right)$

"DSU.h" 85 lines

```
lint det(vector<vector<lint>> a, int n, int p) {
  lint ans = 1;
  for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) a
     ↪[i][j] %= p;
  for(int i = 1; i < n; ++i) {
    for(int j = i+1; j < n; ++j) {
      while (a[j][i] != 0) { // gcd step
        lint t = a[i][i] / a[j][i];
        if (t) for(int k = i; k < n; ++k) {
          a[i][k] = (a[i][k] - a[j][k] * t) % p;
          a[i][k] %= p;
        }
        swap(a[i], a[j]);
        ans *= -1;
      }
    }
    ans = ans * a[i][i] % p;
    if (!ans) return 0;
  }
  return (ans + p) % p;
}
struct edge_t {
  int u, v, w;
  bool operator<(const edge_t& o) const {
    return w < o.w;
  }
};
const int N = 101;
int edgenum = 0;
vector<edge_t> edge;
vector<bool> seen;
vector<int> g[N];
vector<vector<lint>> p, deg;
void addEdge(int u, int v, int d){
  edge_t E = { u, v, d };
  edge[++edgenum] = E;
}
lint MST_count(int n, lint MOD) {
  sort(edge.begin()+1, edge.begin()+edgenum+1);
  int pre = edge[1].w;
  lint ans = 1;
  UF a(n+1), b(n+1);
  seen = vector<bool>(n+1, false);
  deg = vector<vector<lint>>(n+1, vector<lint>(n+1));
  for (int i = 0; i <= n; i++) g[i].clear();
  for (int t = 1; t <= edgenum+1; ++t) {
    if (edge[t].w != pre || t == edgenum + 1) {
      for (int i = 1, k; i <= n; i++) if (seen[i]) {
        k = b.find(i);
        g[k].push_back(i);
        seen[i] = false;
      }
      for (int i = 1; i <= n; ++i)
        if (g[i].size()) {
          p = vector<vector<lint>>(n+1, vector<lint>(n+1));
          for (int j = 0; j < g[i].size(); j++)
          for (int k = j+1, x, y; k < g[i].size(); ++k) {
            x = g[i][j];
            y = g[i][k];
            p[j][k] = p[k][j] = -deg[x][y];
            p[j][j] += deg[x][y];
            p[k][k] += deg[x][y];
          }
          ans = ans*det(p, g[i].size(), MOD) % MOD;
          for (int j = 0; j < g[i].size(); ++j) a.par[g[i][
             ↪j]] = i;
        }
      deg = vector<vector<lint>>(n+1, vector<lint>(n+1));
      for (int i = 1; i <= n; ++i) {
        b.par[i] = a.find(i);
        g[i].clear();
      }
      if (t == edgenum+1) break;
      pre = edge[t].w;
    }
    int x = a.find(edge[t].u);
    int y = a.find(edge[t].v);
    if (x == y) continue;
    seen[x] = seen[y] = true;
    b.unite(x, y);
    deg[x][y]++; deg[y][x]++;
  }
  if (!edgenum) return 0;
  for (int i = 2; i <= n; i++)
    if (b.find(i) != b.find(1)) return 0;
  return ans;
} // hash-cpp-all = d2ed36c54cf26fb9fdf6b7bf5009816c
```

ManhattanMST.h

Description: Given N points, returns up to 4*N edges, which are guaranteed to contain a minimum spanning tree for the graph with edge weights $w(p,q) = |p.x - q.x| + |p.y - q.y|$. Edges are in the form (distance, src, dst). Use a standard MST algorithm on the result to find the final MST.
Time: $\mathcal{O}\left(NlogN\right)$

<UnionFind.h> 28 lines

```
typedef Point<int> P;
pair<vector<array<int, 3>>, int> manhattanMST(vector<P> ps)
   ↪ {
    vector<int> id(ps.size());
    iota(id.begin(), id.end(), 0);
    vector<array<int, 3>> edges;
    for(int k = 0; k < 4; ++k) {
        sort(id.begin(), id.end(), [&](int i, int j) {
             return (ps[i]-ps[j]).x < (ps[j]-ps[i]).y;});
        map<int, int> sweep;
        for(auto& i : id) {
            for (auto it = sweep.lower_bound(-ps[i].y);
                      it != sweep.end(); sweep.erase(it
                         ↪++)) {
                int j = it->second;
                P d = ps[i] - ps[j];
                if (d.y > d.x) break;
                edges.push_back({d.y + d.x, i, j});
            }
            sweep[-ps[i].y] = i;
        }
        if (k & 1) for(auto& p : ps) p.x = -p.x;
        else for(auto& p : ps) swap(p.x, p.y);
    }
    sort(edges.begin(), edges.end());
    UF uf(ps.size());
    int cost = 0;
    for (auto e: edges) if (uf.unite(e[1], e[2])) cost += e
       ↪[0];
    return {edges, cost};
} // hash-cpp-all = de81704447870021010c8019913b976a
```

Pruefer.cpp

Description: Given a tree, construct its pruefer sequence. The Pruefer code is a way of encoding a labeled tree with n vertices using a sequence of $n2$ integers in the interval $[0, n-1]$. This encoding also acts as a bijection between all spanning trees of a complete graph and the numerical sequences.

37 lines

```
struct pruefer_t {
    vector<vector<int>> adj;
    vector<int> parent;
    pruefer_t(int _n) : adj(n), parent(n) {}
    void dfs (int u) {
        for (int i = 0; i < adj[u].size(); ++i) {
            if (i != parent[u]) {
                parent[i] = v;
                dfs(i);
            }
        }
    }
    vector<int> pruefer() {
        int n = adj.size();
        parent.resize(n);
        parent[n-1] = -1;
        dfs(n-1);
        int one_leaf = -1;
        vector<int> degree(n), ret(n-2);
        for (int i = 0; i < n; ++i) {
            degree[i] = adj[i].size();
            if (degree[i] == 1 && one_leaf == -1) one_leaf
               ↪= 1;
        }
        int leaf = one_leaf;
        for (int i = 0; i < n-2; ++i) {
            int next = parent[leaf];
            ret[i] = next;
            if (--degree[next] == 1 && next < one_leaf)
               ↪leaf = next;
            else {
                ++one_leaf;
                while (degree[one_leaf] != 1) ++one_leaf;
                leaf = one_leaf;
            }
        }
        return ret;
    }
}; // hash-cpp-all = 9617131fb6492a5a9ac2ba9ace41373d
```

ErdosGallai.h

Description: Check if an array of degrees can represent a graph
Time: if sorted $\mathcal{O}\left(n\right)$, otherwise $\mathcal{O}\left(nlog(n)\right)$

15 lines

```
bool EG(vector<int> deg) {
    sort(deg.begin(), deg.end(), greater<int>());
    vector<long long> dp(deg.size());
```

```
    int n = deg.size(), p = n-1;
    for(int i = 0; i < n; i++)
        dp[i] = deg[i] + (i > 0 ? dp[i-1] : 0);
    for(int k = 1; k <= n; k++) {
        while(p >= 0 && deg[p] < k) p--;
        long long sum;
        if (p >= k-1) sum = (p-k+1)*1ll*k + dp[n-1] - dp[p
            ↪];
        else sum = dp[n-1] - dp[k-1];
        if (dp[k-1] > k*(k-1LL) + sum) return 0;
    }
    return dp[n-1] % 2 == 0;
} // hash-cpp-all = 56391b5d1b51835fc41d647ab44f510d
```

MisraGries.h

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

47 lines

```
struct edge {int to, color, rev; };
struct MisraGries {
    int N, K = 0;
    vector<vector<int>> F;
    vector<vector<edge>> graph;
    MisraGries(int n) : N(n), graph(n) {}
    // add an undirected edge, NO DUPLICATES ALLOWED
  void addEdge(int u, int v) {
    graph[u].push_back({v, -1, (int) graph[v].size()});
    graph[v].push_back({u, -1, (int) graph[u].size()-1});
  }
  void color(int v, int i) {
    vector<int> fan = { i };
    vector<bool> used(graph[v].size());
    used[i] = true;
    for (int j = 0; j < (int) graph[v].size(); j++)
      if (!used[j] && graph[v][j].col >= 0 && F[graph[v][
          ↪fan.back()].to][graph[v][j].col] < 0)
        used[j] = true, fan.push_back(j), j = -1;
    int c = 0; while (F[v][c] >= 0) c++;
    int d = 0; while (F[graph[v][fan.back()].to][d] >= 0) d
        ↪++;
    int w = v, a = d, k = 0, ccol;
    while (true) {
      swap(F[w][c], F[w][d]);
      if (F[w][c] >= 0) graph[w][F[w][c]].col = c;
      if (F[w][d] >= 0) graph[w][F[w][d]].col = d;
      if (F[w][a^=c^d] < 0) break;
      w = graph[w][F[w][a]].to;
    }
    do {
      Edge &e = graph[v][fan[k]];
      ccol = F[e.to][d] < 0 ? d : graph[v][fan[k+1]].col;
      if (e.col >= 0) F[e.to][e.col] = -1;
      F[e.to][ccol] = e.rev;
      F[v][ccol] = fan[k];
      e.col = graph[e.to][e.rev].col = ccol;
      k++;
    } while (ccol != d);
  }
  // finds a K-edge-coloringraph
  void color() {
    for(int v = 0; v < N; ++v)
        K = max(K, (int)graph[v].size() + 1);
    F = vector<vector<int>>(N, vector<int>(K, -1));
    for(int v = 0; v < N; ++v) for (int i = graph[v].size()
        ↪; i--; )
      if (graph[v][i].col < 0) color(v, i);
  }
};
node_t *merge(node_t *a, node_t *b) {
    if (!a || !b) return a ?: b;
    a->prop(), b->prop();
    if (a->key.w > b->key.w) swap(a, b);
    swap(a->l, (a->r = merge(b, a->r)));
    return a;
}
void pop(node_t*& a) { a->prop(); a = merge(a->l, a->r); }
lint dmst(int n, int r, vector<edge_t>& g) {
    UF uf(n);
    vector<node_t*> heap(n);
    for(auto &e : g) heap[e.b] = merge(heap[e.b], new
        ↪node_t{e});
    lint res = 0;
    vector<int> seen(n, -1), path(n);
    seen[r] = r;
    for(int s = 0; s < n; ++s) {
        int u = s, qi = 0, w;
        while (seen[u] < 0) {
            path[qi++] = u, seen[u] = s;
            if (!heap[u]) return -1;
            edge_t e = heap[u]->top();
            heap[u]->delta -= e.w, pop(heap[u]);
            res += e.w, u = uf.find(e.a);
            if (seen[u] == s) {
                node_t* cyc = 0;
                do cyc = merge(cyc, heap[w = path[--qi]]);
                while (uf.unite(u, w));
                u = uf.find(u);
                heap[u] = cyc, seen[u] = -1;
            }
        }
    }
    return res;
} // hash-cpp-all = e6517a3e2849e63d1af1fb535148ced3
```

Graph-Negative-Cycle.cpp

Description: negative cycle

26 lines

```
double b[N][N];
double dis[N];
int vis[N], pc[N];
bool dfs(int k) {
  vis[k] += 1; pc[k] = true;
  if (vis[k] > N) return true;
  for (int i = 0; i < N; i++)
    if (dis[k] + b[k][i] < dis[i]) {
      dis[i] = dis[k] + b[k][i];
      if (!pc[i]) {
        if (dfs(i)) return true;
      } else return true;
    }
  pc[k] = false;
  return false;
}
bool chk(double d) {
  for (int i = 0; i < N; i ++)
    for (int j = 0; j < N; j ++)
      b[i][j] = -a[i][j] + d;
  for (int i = 0; i < N; i++)
    vis[i] = false, dis[i] = 0, pc[i] = false;
  for (int i = 0; i < N; i++)
    if (!vis[i] && dfs(i)) return true;
  return false;
} // hash-cpp-all = c0deaa64151d7ae840553946bfebdc69
```

7.9 Theorems

7.9.1 Vizing's Thereom

Dado um grafo G, seja δ o maior grau de um vértice. Então G tem número cromático de aresta δ ou $\delta + 1$.

7.9.2 Euler's theorem

Sendo V, A e F as quantidades de vértices, arestas e faces de um grafo planar conexo, $V - A + F = 2$.

7.9.3 Hall's Marriage theorem

Dado um grafo bipartido com classes V_1 e V_2, para $S \subset V_1$ seja $N(S)$ o conjunto de todos os vértices vizinhos a algum elemento de S. Um emparelhamento de V_1 em V_2 é um conjunto de arestas disjuntas cujas extremidades estão em classes diferentes. Então existe um emparelhamento completo de V_1 em V_2 sse $|N(S)| \geq |S|\ \forall\ S \subset V_1$.

7.9.4 Dilworth's theorem

Em todo conjunto parcialmente ordenado, a quantidade máxima de elementos de uma anticadeia é igual à quatidade mínima de cadeias disjuntas que cobrem o conjunto.

7.9.5 Conjunto Independente de Peso Máximo num Grafo Bipartido

É o mesmo que a cobertura de peso mínimo. Podemos resolver criando uma rede de fluxo com arestas $(S, u, w(u))$ para $u \in L$, $(v, T, w(v))$ para $v \in R$ e (u, v, ∞) para $(u, v) \in E$. O corte mínimo de S a T é a resposta. Vértices adjacentes a uma aresta de corte estão na cobertura de vértices.

7.9.6 Tutte-Berge formula

The theorem states that the size of a maximum matching of a graph $G = (V, E)$ equals $\frac{1}{2}\min_{U \subseteq V}(|U| - \texttt{odd}(G - U) + |V|)$, where odd(H) counts how many of the connected components of the graph H have an odd number of vertices.

7.9.7 Tutte's theorem

Um grafo $G = (V, A)$ tem um emparelhamento perfeito sse para todo subconjunto U de V, o subgrafo induzido por $V \setminus U$ tem no máximo $|U|$ componentes conexas com uma quantidade ímpar de vértices.

7.9.8 Menger's theorem

Para vértices: Um grafo é k-conexo sse todo par de vértices é conectado por pelo menos k caminhos sem vértices intermediários em comum.

Para arestas: Um grafo é dito k-aresta-conexo se a retirada de menos de k arestas do grafo o mantém conexo. Então um grafo é k -aresta-conexo sse para todo par de vértices u e v, existem k caminhos que ligam u a v sem arestas em comum.

7.9.9 Landau

Existe um torneio com graus de saída $d_1 \leq d_2 \leq \ldots \leq d_n$ sse:

- $d_1 + d_2 + \ldots + d_n = \binom{n}{2}$
- $d_1 + d_2 + \ldots + d_k \geq \binom{k}{2} \quad \forall 1 \leq k \leq n.$

Para construir, fazemos 1 apontar para $2, 3, \ldots, d_1 + 1$ e seguimos recursivamente.

7.9.10 Erdős–Gallai theorem

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

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

7.9.11 Turán's theorem

Nenhum grafo com n vértices que é K_{r+1}-livre pode ter mais arestas do que o grafo de $Turán$: Um grafo completo k-partido com conjuntos de tamanho mais próximo possível.

7.9.12 Dirac's theorem

Seja G um grafo com n vértices, cada um com grau pelo menos $n/2$. Então G é hamiltoniano.

7.9.13 Ore's theorem

Seja G um grafo simples de ordem $n \geq 3$ tal que

$$g(u) + g(v) \geq n$$

para todo par u,v de vértices não adjacentes, então G é hamiltoniano.

7.9.14 Eulerian Cycles

A quantidade de ciclos *Eulerianos* num digrafo G é:

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

onde $t_w(G)$ é a quantidade de arborescências (árvore geradora direcionada) enraizada em w:
$t_w(G) = \det(q_{ij})_{i,j \neq w}$, with
$q_{ij} = [i = j]\text{indeg}(i) - \#(i, j) \in E.$

7.9.15 Matroid Intersection theorem

Sejam $M_1 = (E, I_1)$ e $M_2 = (E, I_2)$ matróides. Então $\max_{S \in I_1 \cap I_2} |S| = \min_{U \subseteq E} r_1(U) + r_2(E \setminus U)$.

7.9.16 Number of Spanning Trees

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

7.9.17 König-Egervary theorem

Em todo grafo bipartido G, a quantidade de arestas no emparelhamento máximo é maior ou igual à quantidade de vértices na cobertura mínima. Ou seja, para todo G, $\alpha(G) \geq \beta(G)$. Note que isso prova que $\alpha(G) = \beta(G)$ para grafos bipartidos.

7.9.18 Tutte Matrix

- A graph has a perfect matching iff the $Tutte$ matrix has a non-zero determinant.
- The rank of the $Tutte$ matrix is equal to twice the size of the maximum matching. The maximum cost matching can be found by polynomial interpolation.

Geometry (8)

8.1 Geometric primitives

Point.h

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

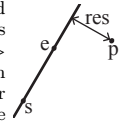
28 lines

```
template <class T> int sgn(T x) { return (x > 0) - (x < 0);
  ↪ }
template<class T>
struct Point {
  typedef Point P;
  T x, y;
  explicit Point(T x=0, T y=0) : x(x), y(y) {}
  bool operator<(P p) const { return tie(x,y) < tie(p.x,p.y
    ↪); }
  bool operator==(P p) const { return tie(x,y)==tie(p.x,p.y
    ↪); }
  P operator+(P p) const { return P(x+p.x, y+p.y); }
  P operator-(P p) const { return P(x-p.x, y-p.y); }
  P operator*(T d) const { return P(x*d, y*d); }
  P operator/(T d) const { return P(x/d, y/d); }
  T dot(P p) const { return x*p.x + y*p.y; }
  T cross(P p) const { return x*p.y - y*p.x; }
  T cross(P a, P b) const { return (a-*this).cross(b-*this)
    ↪; }
  T dist2() const { return x*x + y*y; }
  double dist() const { return sqrt((double)dist2()); }
  // angle to x-axis in interval [-pi, pi]
  double angle() const { return atan2(y, x); }
  P unit() const { return *this/dist(); } // makes dist()=1
  P perp() const { return P(-y, x); } // rotates +90
    ↪degrees
  P normal() const { return perp().unit(); }
  // returns point rotated 'a' radians ccw around the
    ↪origin
  P rotate(double a) const {
    return P(x*cos(a)-y*sin(a),x*sin(a)+y*cos(a)); }
};
using P = Point<double>;
// hash-cpp-all = f90ade9c83109953cbaa31ef81c2acf8
```

Complex.h

LineDistance.h

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

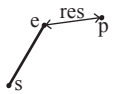


"Point.h"

4 lines

```
template<class P>
double lineDist(const P& a, const P& b, const P& p) {
  return (double)(b-a).cross(p-a)/(b-a).dist();
} // hash-cpp-all = f6bf6b556d99b09f42b86d28d1eaa86d
```

SegmentDistance.h



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

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

"Point.h" 5 lines

```
double segDist(P& s, P& e, P& p) {
  if (s==e) return (p-s).dist();
  auto d = (e-s).dist2(), t = min(d,max(.0,(p-s).dot(e-s)))
    ↪;
  return ((p-s)*d-(e-s)*t).dist()/d;
} // hash-cpp-all = ae751abdc935034be953183ec1e22b31
```

SegmentClosestPoint.h

Description: Returns the closest point to p in the segment from point s to e as well as the distance between them

13 lines

```
pair<P,double> SegmentClosestPoint(P &s, P &e, P &p){
  P ds=p-s, de=p-e;
  if(e==s)
    return {s, ds.dist()};
  P u=(e-s).unit();
  P proj=u*ds.dot(u);
  if(onSegment(s, e, proj+s))
    return {proj+s, (ds-proj).dist()};
  double dist_s=ds.dist(), dist_e=de.dist();
  if(cmp(dist_s, dist_e)==1)
    return {s, dist_s};
  return{e, dist_e};
} // hash-cpp-all = d4b82f64908a45c928d4451948ff0f60
```

SegmentIntersection.h

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

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

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

```
template<class P> vector<P> segInter(P a, P b, P c, P d) {
  auto oa = c.cross(d, a), ob = c.cross(d, b),
       oc = a.cross(b, c), od = a.cross(b, d);
  // Checks if intersection is single non-endpoint point.
  if (sgn(oa) * sgn(ob) < 0 && sgn(oc) * sgn(od) < 0)
    return {(a * ob - b * oa) / (ob - oa)};
  set<P> s;
  if (onSegment(c, d, a)) s.insert(a);
  if (onSegment(c, d, b)) s.insert(b);
  if (onSegment(a, b, c)) s.insert(c);
  if (onSegment(a, b, d)) s.insert(d);
  return {s.begin(), s.end()};
} // hash-cpp-all = f6be1695014f7d839a498a46024031e2
```

SegmentIntersectionQ.h

Description: Like segmentIntersection, but only returns true/false. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or long long.

"Point.h" 16 lines

```
template<class P>
bool segmentIntersectionQ(P s1, P e1, P s2, P e2) {
  if (e1 == s1) {
    if (e2 == s2) return e1 == e2;
    swap(s1,s2); swap(e1,e2);
  }
  P v1 = e1-s1, v2 = e2-s2, d = s2-s1;
  auto a = v1.cross(v2), a1 = d.cross(v1), a2 = d.cross(v2)
    ↪;
  if (a == 0) { // parallel
    auto b1 = s1.dot(v1), c1 = e1.dot(v1),
         b2 = s2.dot(v1), c2 = e2.dot(v1);
    return !a1 && max(b1,min(b2,c2)) <= min(c1,max(b2,c2));
  }
  if (a < 0) { a = -a; a1 = -a1; a2 = -a2; }
  return (0 <= a1 && a1 <= a && 0 <= a2 && a2 <= a);
} // hash-cpp-all = 1ff4ba22bd0aefb04bf48cca4d6a7d8c
```

LineIntersection.h

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

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

"Point.h" 8 lines

```
template<class P>
pair<int, P> lineInter(P s1, P e1, P s2, P e2) {
  auto d = (e1 - s1).cross(e2 - s2);
  if (d == 0) // if parallel
    return {-(s1.cross(e1, s2) == 0), P(0, 0)};
  auto p = s2.cross(e1, e2), q = s2.cross(e2, s1);
  return {1, (s1 * p + e1 * q) / d};
} // hash-cpp-all = a01f815e2e60161e03879264c4826dd0
```

LineProjectionReflection.h

Description: Projects point p onto line ab. Set refl=true to get reflection of point p across line ab insted. The wrong point will be returned if P is an integer point and the desired point doesn't have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow.

"Point.h" 5 lines

```
template<class P>
P lineProj(P a, P b, P p, bool refl=false) {
    P v = b - a;
    return p - v.perp()*(1+refl)*v.cross(p-a)/v.dist2();
} // hash-cpp-all = b5562d9ee2f720df36d24b4a7d427ea5
```

SideOf.h

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

Usage: `bool left = sideOf(p1,p2,q)==1;`

"Point.h" 9 lines

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

```
template<class P>
int sideOf(const P& s, const P& e, const P& p, double eps)
  ↪{
  auto a = (e-s).cross(p-s);
  double l = (e-s).dist()*eps;
  return (a > l) - (a < -l);
} // hash-cpp-all = 3af81cc4f24d9d9fb109d930f3b9764c
```

OnSegment.h

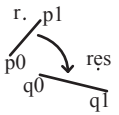
Description: Returns true iff p lies on the line segment from s to e. Use (segDist(s,e,p)<=epsilon) instead when using Point<double>.

"Point.h" 4 lines

```
template<class P> bool onSegment(P s, P e, P p) {
  return p.cross(s, e) == 0 && (s - p).dot(e - p) <= 0;
}
// hash-cpp-all = c597e8749250f940e4b0139f0dc3e8b9
```

LinearTransformation.h

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



"Point.h" 6 lines

```
typedef Point<double> P;
P linearTransformation(const P& p0, const P& p1,
    const P& q0, const P& q1, const P& r) {
  P dp = p1-p0, dq = q1-q0, num(dp.cross(dq), dp.dot(dq));
  return q0 + P((r-p0).cross(num), (r-p0).dot(num))/dp.
    ↪dist2();
} // hash-cpp-all = 03a3061b3ef024b4e29ea06169932b21
```

Angle.h

Description: A class for ordering angles (as represented by int points and a number of rotations around the origin). Useful for rotational sweeping. Sometimes also represents points or vectors.

Usage: `vector<Angle> v = {w[0], w[0].t360() ...}; // sorted int j = 0; rep(i,0,n) { while (v[j] < v[i].t180()) ++j; } // sweeps j such that (j-i) represents the number of positively oriented triangles with vertices at 0 and i`

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 b) const { return {x-b.x, y-b.y, t
    ↪}; }
  int half() const {
    assert(x || y);
    return y < 0 || (y == 0 && x < 0);
  }
  Angle t90() const { return {-y, x, t + (half() && x >= 0)
    ↪}; }
  Angle t180() const { return {-x, -y, t + half()}; }
  Angle t360() const { return {x, y, t + 1}; }
};
bool operator<(Angle a, Angle b) {
  // add a.dist2() and b.dist2() to also compare distances
  return make_tuple(a.t, a.half(), a.y * (ll)b.x) <
         make_tuple(b.t, b.half(), a.x * (ll)b.y);
}
// Given two points, this calculates the smallest angle
  ↪between
// them, i.e., the angle that covers the defined line
  ↪segment.
pair<Angle, Angle> segmentAngles(Angle a, Angle b) {
```

```
  if (b < a) swap(a, b);
  return (b < a.t180() ?
          make_pair(a, b) : make_pair(b, a.t360()));
}
Angle operator+(Angle a, Angle b) { // point a + vector b
  Angle r(a.x + b.x, a.y + b.y, a.t);
  if (a.t180() < r) r.t--;
  return r.t180() < a ? r.t360() : r;
}
Angle angleDiff(Angle a, Angle b) { // angle b - angle a
  int tu = b.t - a.t; a.t = b.t;
  return {a.x*b.x + a.y*b.y, a.x*b.y - a.y*b.x, tu - (b < a
     ↪)};
} // hash-cpp-all = 0f0602c74320456a8a8627737c1d3c64
```

AngleCmp.h

Description: Useful utilities for dealing with angles of rays from origin. OK for integers, only uses cross product. Doesn't support (0,0). 22 lines

```
template <class P>
bool sameDir(P s, P t) {
  return s.cross(t) == 0 && s.dot(t) > 0;
}
// checks 180 <= s..t < 360?
template <class P>
bool isReflex(P s, P t) {
  auto c = s.cross(t);
  return c ? (c < 0) : (s.dot(t) < 0);
}
// operator < (s,t) for angles in [base,base+2pi)
template <class P>
bool angleCmp(P base, P s, P t) {
  int r = isReflex(base, s) - isReflex(base, t);
  return r ? (r < 0) : (0 < s.cross(t));
}
// is x in [s,t] taken ccw? 1/0/-1 for in/border/out
template <class P>
int angleBetween(P s, P t, P x) {
  if (sameDir(x, s) || sameDir(x, t)) return 0;
  return angleCmp(s, x, t) ? 1 : -1;
} // hash-cpp-all = 6edd25f30f9c69989bbd2115b4fdceda
```

LinearSolver.h

8.2 Circles

CircleIntersection.h

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

"Point.h" 10 lines

```
bool circleInter(P a,P b,double r1,double r2,pair<P, P>*
   ↪out) {
  if (a == b) { assert(r1 != r2); return false; }
  P vec = b - a;
  double d2 = vec.dist2(), sum = r1+r2, dif = r1-r2,
         p = (d2 + r1*r1 - r2*r2)/(d2*2), h2 = r1*r1 - p*p*
            ↪d2;
  if (sum*sum < d2 || dif*dif > d2) return false;
  P mid = a + vec*p, per = vec.perp() * sqrt(fmax(0, h2) /
     ↪d2);
  *out = {mid + per, mid - per};
  return true;
} // hash-cpp-all = c64785998cac9428b9966a1c56300216
```

CircleTangents.h

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

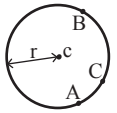
"Point.h" 13 lines

```
template<class P>
vector<pair<P, P>> tangents(P c1, double r1, P c2, double
   ↪r2) {
  P d = c2 - c1;
  double dr = r1 - r2, d2 = d.dist2(), h2 = d2 - dr * dr;
  if (d2 == 0 || h2 < 0)  return {};
  vector<pair<P, P>> out;
  for (double sign : {-1, 1}) {
    P v = (d * dr + d.perp() * sqrt(h2) * sign) / d2;
    out.push_back({c1 + v * r1, c2 + v * r2});
  }
  if (h2 == 0) out.pop_back();
  return out;
} // hash-cpp-all = b0153d0ef1b8a6b1fa4d91480c4126e8
```

Circumcircle.h

Description:

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



"Point.h" 8 lines

```
double ccRadius(const P& A, const P& B, const P& C) {
  return (B-A).dist()*(C-B).dist()*(A-C).dist()/
      abs((B-A).cross(C-A))/2;
}
P ccCenter(const P& A, const P& B, const P& C) {
  P b = C-A, c = B-A;
  return A + (b*c.dist2()-c*b.dist2()).perp()/b.cross(c)/2;
} // hash-cpp-all = c98102c51cd46aeac93a76816c8f6211
```

MinimumEnclosingCircle.h

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

"circumcircle.h" 19 lines

```
pair<P, double> mec(vector<P> ps) {
  shuffle(ps.begin(),ps.end(), mt19937(time(0)));
  P o = ps[0];
  double r = 0, EPS = 1 + 1e-8;
  for(int i = 0; i < ps.size(); ++i)
      if ((o - ps[i]).dist() > r * EPS) {
        o = ps[i], r = 0;
        for(int j = 0; j < i; ++j) if ((o - ps[j]).dist() >
           ↪ r * EPS) {
          o = (ps[i] + ps[j]) / 2;
          r = (o - ps[i]).dist();
          for(int k = 0; k < j; ++k)
              if ((o - ps[k]).dist() > r * EPS) {
                o = ccCenter(ps[i], ps[j], ps[k]);
                r = (o - ps[i]).dist();
              }
        }
      }
  return {o, r};
} // hash-cpp-all = 8ab87fe7c0e622c4171e24dcad6bee01
```

CircleUnion.h

Description: Computes the circles union total area

87 lines

```
struct CircleUnion {
    static const int maxn = 1e5 + 5;
    const double PI = acos((double)-1.0);
    int n;
    double x[maxn], y[maxn], r[maxn];
    int covered[maxn];
    vector<pair<double, double>> seg, cover;
    double arc, pol;
    inline int sign(double x) {return x < -EPS ? -1 : x >
       ↪EPS;}
    inline int sign(double x, double y) {return sign(x - y)
       ↪;}
    inline double sqr(const double x) {return x * x;}
    inline double dist(double x1, double y1, double x2,
       ↪double y2) {return sqrt(sqr(x1 - x2) + sqr(y1 - y2
       ↪));}
    inline double angle(double A, double B, double C) {
        double val = (sqr(A) + sqr(B) - sqr(C)) / (2 * A *
           ↪B);
        if (val < -1) val = -1;
        if (val > +1) val = +1;
        return acos(val);
    }
    CircleUnion() {
        n = 0;
        seg.clear(), cover.clear();
        arc = pol = 0;
    }
    void init() {
        n = 0;
        seg.clear(), cover.clear();
        arc = pol = 0;
    }
    void add(double xx, double yy, double rr) {
        x[n] = xx, y[n] = yy, r[n] = rr, covered[n] = 0, n
           ↪++;
    }
    void getarea(int i, double lef, double rig) {
        arc += 0.5 * r[i] * r[i] * (rig - lef - sin(rig -
           ↪lef));
        double x1 = x[i] + r[i] * cos(lef), y1 = y[i] + r[i
           ↪] * sin(lef);
        double x2 = x[i] + r[i] * cos(rig), y2 = y[i] + r[i
           ↪] * sin(rig);
        pol += x1 * y2 - x2 * y1;
    }
    double calc() {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < i; j++)
                if (!sign(x[i] - x[j]) && !sign(y[i] - y[j
                   ↪]) && !sign(r[i] - r[j])) {
                    r[i] = 0.0;
                    break;
                }
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (i != j && sign(r[j] - r[i]) >= 0 &&
                   ↪sign(dist(x[i], y[i], x[j], y[j]) - (r
                   ↪[j] - r[i])) <= 0) {
                    covered[i] = 1;
                    break;
                }
        for (int i = 0; i < n; i++) {
            if (sign(r[i]) && !covered[i]) {
                seg.clear();
```

```
                for (int j = 0; j < n; j++)
                    if (i != j) {
                        double d = dist(x[i], y[i], x[j], y
                            ↪[j]);
                        if (sign(d - (r[j] + r[i])) >= 0 ||
                            ↪ sign(d - abs(r[j] - r[i])) <=
                            ↪ 0)
                            continue;
                        double alpha = atan2(y[j] - y[i], x
                            ↪[j] - x[i]);
                        double beta = angle(r[i], d, r[j]);
                        pair<double, double> tmp(alpha -
                            ↪beta, alpha + beta);
                        if (sign(tmp.first) <= 0 && sign(
                            ↪tmp.second) <= 0)
                            seg.push_back(pair<double,
                                ↪double>(2 * PI + tmp.first
                                ↪, 2 * PI + tmp.second));
                        else if (sign(tmp.first) < 0) {
                            seg.push_back(pair<double,
                                ↪double>(2 * PI + tmp.first
                                ↪, 2 * PI));
                            seg.push_back(pair<double,
                                ↪double>(0, tmp.second));
                        }
                        else seg.push_back(tmp);
                    }
                sort(seg.begin(), seg.end());
                double rig = 0;
                for (vector<pair<double, double> >::
                    ↪iterator iter = seg.begin(); iter !=
                    ↪seg.end(); iter++) {
                    if (sign(rig - iter->first) >= 0)
                        rig = max(rig, iter->second);
                    else {
                        getarea(i, rig, iter->first);
                        rig = iter->second;
                    }
                }
                if (!sign(rig)) arc += r[i] * r[i] * PI;
                else getarea(i, rig, 2 * PI);
            }
        }
        return pol / 2.0 + arc;
    }
} ccu;
// hash-cpp-all = fd65da13521a93e5f0e2c1632e9f56e3
```

CircleLine.h

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

"Point.h", "lineDistance.h", "LineProjectionReflection.h" — 8 lines

```
template<class P>
vector<P> circleLine(P c, double r, P a, P b) {
  double h2 = r*r - a.cross(b,c)*a.cross(b,c)/(b-a).dist2()
    ↪;
  if (h2 < 0) return {};
  P p = lineProj(a, b, c), h = (b-a).unit() * sqrt(h2);
  if (h2 == 0) return {p};
  return {p - h, p + h};
} // hash-cpp-all = debf8692dd3065ebd04e25a202df5a42
```

CircleCircleArea.h

Description: Calculates the area of the intersection of 2 circles

12 lines

```
template<class P>
double circleCircleArea(P c, double cr, P d, double dr) {
    if (cr < dr) swap(c, d), swap(cr, dr);
    auto A = [&](double r, double h) {
        return r*r*acos(h/r)-h*sqrt(r*r-h*h);
    };
    auto l = (c - d).dist(), a = (l*l + cr*cr - dr*dr)/(2*l
        ↪);
    if (l - cr - dr >= 0) return 0; // far away
    if (l - cr + dr <= 0) return M_PI*dr*dr;
    if (l - cr >= 0) return A(cr, a) + A(dr, l-a);
    else return A(cr, a) + M_PI*dr*dr - A(dr, a-l);
} // hash-cpp-all = 8bf2b6afed06c7a4f47957f60986f58e
```

CirclePolygonIntersection.h

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

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

"Point.h" — 18 lines

```
#define arg(p, q) atan2(p.cross(q), p.dot(q))
double circlePoly(P c, double r, vector<P> ps) {
  auto tri = [&](P p, P q) {
    auto r2 = r * r / 2;
    P d = q - p;
    auto a = d.dot(p)/d.dist2(), b = (p.dist2()-r*r)/d.
      ↪dist2();
    auto det = a * a - b;
    if (det <= 0) return arg(p, q) * r2;
    auto s = max(0., -a-sqrt(det)), t = min(1., -a+sqrt(det
      ↪));
    if (t < 0 || 1 <= s) return arg(p, q) * r2;
    P u = p + d * s, v = p + d * t;
    return arg(p,u) * r2 + u.cross(v)/2 + arg(v,q) * r2;
  };
  auto sum = 0.0;
  for (int i = 0; i < ps.size(); ++i)
    sum += tri(ps[i] - c, ps[(i + 1) % ps.size()] - c);
  return sum;
} // hash-cpp-all = c6ad2247294cc5ffd45e6ff0a3171191
```

8.3 Polygons

InsidePolygon.h

Description: Returns true if p lies within the polygon. If strict is true, it returns false for points on the boundary. The algorithm uses products in intermediate steps so watch out for overflow.

Usage: `vector<P> v = {P{4,4}, P{1,2}, P{2,1}};`
`bool in = inPolygon(v, P{3, 3}, false);`

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

"Point.h", "OnSegment.h", "SegmentDistance.h" — 12 lines

```
template<class P>
bool inPolygon(vector<P> &p, P a, bool strict = true) {
  int cnt = 0, n = p.size();
  for(int i = 0; i < n; ++i) {
    P q = p[(i + 1) % n];
    if (onSegment(p[i], q, a)) return !strict; // change to
        // -1 if u need to detect points in the boundary
    //or: if (segDist(p[i], q, a) <= eps) return !strict;
    cnt ^= ((a.y<p[i].y) - (a.y<q.y)) * a.cross(p[i], q) >
      ↪0;
  }
  return cnt;
} // hash-cpp-all = f9442d2902bed2ba7b9bccd3adc59cf5
```

PolygonArea.h

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

"Point.h" — 17 lines

```
template<class T>
T polygonArea(vector<Point<T>> &v) {
  T a = v.back().cross(v[0]);
  for(int i = 0; i < v.size()-1; ++i)
      a += v[i].cross(v[i+1]);
  return abs(a)/2.0;
}

Point<T> polygonCentroid(vector<Point<T>> &v) { // not
  ↪tested
  Point<T> cent(0,0); T area = 0;
  for(int i = 0; i < v.size(); ++i) {
    int j = (i+1) % (v.size()); T a = cross(v[i], v[j]);
    cent += a * (v[i] + v[j]);
    area += a;
  }
  return cent/area/(T)3;
} // hash-cpp-all = 3794ee519cca1fca6c95078be8322d3a
```

PolygonCenter.h

Description: Returns the center of mass for a polygon.

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

"Point.h" — 8 lines

```
P polygonCenter(const vector<P>& v) {
  P res(0, 0); double A = 0;
  for (int i = 0, j = v.size() - 1; i < v.size(); j = ++i)
    ↪{
    res = res + (v[i] + v[j]) * v[j].cross(v[i]);
    A += v[j].cross(v[i]);
  }
  return res / A / 3;
} // hash-cpp-all = 26a00ffc18412a7320203b30228127c4
```

PolygonCut.h

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

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

"Point.h", "lineIntersection.h" — 11 lines

```
vector<P> polygonCut(const vector<P>& poly, P s, P e) {
  vector<P> res;
  for(int i = 0; i < poly.size(); ++i) {
    P cur = poly[i], prev = i ? poly[i-1] : poly.back();
    bool side = s.cross(e, cur) < 0;
    if (side != (s.cross(e, prev) < 0))
      res.push_back(lineInter(s, e, cur, prev).second);
    if (side) res.push_back(cur);
  }
  return res;
} // hash-cpp-all = 7df36f82a18db63fd1145e017dc27368
```

ConvexHull.h

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

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

"Point.h" — 13 lines

```
vector<P> convexHull(vector<P> pts) {
  if (pts.size() <= 1) return pts;
```

```
  sort(pts.begin(), pts.end());
  vector<P> h(pts.size()+1);
  int s = 0, t = 0;
  for (int it = 2; it--; s = --t, reverse(pts.begin(), pts.
    ↪end()))
    for (P p : pts) {
      while (t >= s + 2 && h[t-2].cross(h[t-1], p) <= 0) t
        ↪--;
      h[t++] = p;
    }
  return {h.begin(), h.begin() + t - (t == 2 && h[0] == h
    ↪[1])};
}
// hash-cpp-all = 3612d7a02c6a3e51900c38449c059647
```

HullDiameter.h

Description: Returns the two points with max distance on a convex hull (ccw, no duplicate/colinear points). 11 lines

```
array<P, 2> hullDiameter(vector<P> S) {
  int n = S.size(), j = n < 2 ? 0 : 1;
  pair<lint, array<P, 2>> res({0, {S[0], S[0]}});
  for(int i = 0; i < j; ++i)
    for (;; j = (j + 1) % n) {
      res = max(res, {(S[i] - S[j]).dist2(), {S[i], S[j]}})
        ↪;
      if ((S[(j + 1) % n] - S[j]).cross(S[i + 1] - S[i]) >=
        ↪ 0)
        break;
    }
  return res.second;
} // hash-cpp-all = 0e0c1f7f1175965bc6920eab2f1dab0f
```

PointInsideHull.h

Description: Determine whether a point t lies inside a convex hull (CCW order, with no colinear points). Returns true if point lies within the hull. If strict is true, points on the boundary aren't included.
Time: $\mathcal{O}(\log N)$

"Point.h", "sideOf.h", "OnSegment.h" 12 lines

```
bool inHull(const vector<P> &l, P p, bool strict = true) {
  int a = 1, b = l.size() - 1, r = !strict;
  if (l.size() < 3) return r && onSegment(l[0], l.back(), p
    ↪);
  if (sideOf(l[0], l[a], l[b]) > 0) swap(a, b);
  if (sideOf(l[0], l[a], p) >= r || sideOf(l[0], l[b], p)<=
    ↪ -r)
    return false;
  while (abs(a - b) > 1) {
    int c = (a + b) / 2;
    (sideOf(l[0], l[c], p) > 0 ? b : a) = c;
  }
  return sgn(l[a].cross(l[b], p)) < r;
} // hash-cpp-all = 7b851472e103148c8d9eb2d488bbeba6
```

PolyUnion.h

Description: Calculates the area of the union of n polygons (not necessarily convex). The points within each polygon must be given in CCW order. Guaranteed to be precise for integer coordinates up to 3e7. If epsilons are needed, add them in sideOf as well as the definition of sgn.
Time: $\mathcal{O}(N^2)$, where N is the total number of points

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

```
double rat(P a, P b) { return sgn(b.x) ? a.x/b.x : a.y/b.y;
  ↪ }
double polyUnion(vector<vector<P>>& poly) {
  double ret = 0;
  for(int i = 0; i < poly.size(); ++i)
    for(int v = 0; v < poly[i].size(); ++v) {
      P A = poly[i][v], B = poly[i][(v + 1) % poly[i].size
        ↪()];
      vector<pair<double, int>> segs = {{0, 0}, {1, 0}};
      for(int j = 0; j < poly.size(); ++j) if (i != j) {
        for(int u = 0; u < poly[j]; ++u) {
          P C = poly[j][u], D = poly[j][(u + 1) % poly[j].
            ↪size()];
          int sc = sideOf(A, B, C), sd = sideOf(A, B, D);
          if (sc != sd) {
            double sa = C.cross(D, A), sb = C.cross(D, B);
            if (min(sc, sd) < 0)
              segs.emplace_back(sa / (sa - sb), sgn(sc - sd
                ↪));
          } else if (!sc && !sd && j<i && sgn((B-A).dot(D-C
            ↪))>0){
            segs.emplace_back(rat(C - A, B - A), 1);
            segs.emplace_back(rat(D - A, B - A), -1);
          }
        }
      }
      sort(segs.begin(), segs.end());
      for(auto& s : segs) s.first = min(max(s.first, 0.0),
        ↪1.0);
      double sum = 0;
      int cnt = segs[0].second;
      for(int j = 1; j < segs.size(); ++j) {
        if (!cnt) sum += segs[j].first - segs[j - 1].first;
        cnt += segs[j].second;
      }
      ret += A.cross(B) * sum;
    }
  return ret / 2;
} // hash-cpp-all = a45bd4451dd4dfcf191932fa0db01fa7
```

LineHullIntersection.h

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

"Point.h" 39 lines

```
typedef array<P, 2> Line;
#define cmp(i,j) sgn(dir.perp().cross(poly[(i)%n]-poly[(j)%
  ↪n]))
#define extr(i) cmp(i + 1, i) >= 0 && cmp(i, i - 1 + n) < 0
int extrVertex(vector<P>& poly, P dir) { // hash-cpp-1
  int n = poly.size(), left = 0, right = n;
  if (extr(0)) return 0;
  while (left + 1 < right) {
    int m = (left + right) / 2;
    if (extr(m)) return m;
    int ls = cmp(left + 1, left), ms = cmp(m + 1, m);
    (ls < ms || (ls == ms && ls == cmp(left, m)) ? right :
      ↪left) = m;
  }
  return left;
} // hash-cpp-1 = 99da02a2645a6c072258fcdaf6294dc3

#define cmpL(i) sgn(line[0].cross(poly[i], line[1]))
array<int, 2> lineHull(Line line, vector<P> poly) { // hash
  ↪-cpp-2
  int endA = extrVertex(poly, (line[0] - line[1]).perp());
  int endB = extrVertex(poly, (line[1] - line[0]).perp());
  if (cmpL(endA) < 0 || cmpL(endB) > 0)
    return {-1, -1};
  array<int, 2> res;
  for(int i = 0; i < 2; ++i) {
    int left = endB, right = endA, n = poly.size();
    while ((left + 1) % n != right) {
      int m = ((left + right + (left < right ? 0 : n)) / 2)
        ↪ % n;
      (cmpL(m) == cmpL(endB) ? left : right) = m;
    }
    res[i] = (left + !cmpL(right)) % n;
    swap(endA, endB);
  }
  if (res[0] == res[1]) return {res[0], -1};
  if (!cmpL(res[0]) && !cmpL(res[1]))
    switch ((res[0] - res[1] + sz(poly) + 1) % poly.size())
      ↪ {
      case 0: return {res[0], res[0]};
      case 2: return {res[1], res[1]};
    }
  return res;
} // hash-cpp-2 = 3e0265a348f4f3ff92f451fd599a582b
```

HalfPlane.h

Description: Halfplane intersection area

"Point.h", "lineIntersection.h" 59 lines

```
#define eps 1e-8
struct Line { // hash-cpp-1
  P P1, P2;
  // Right hand side of the ray P1 -> P2
  explicit Line(P a = P(), P b = P()) : P1(a), P2(b) {};
  P intpo(Line y) {
    P r;
    assert(lineIntersection(P1, P2, y.P1, y.P2, r) == 1);
    return r;
  }
  P dir() { return P2 - P1; }
  bool contains(P x) { return (P2 - P1).cross(x - P1) < eps
    ↪; }
  bool out(P x) { return !contains(x); }
}; // hash-cpp-1 = 5bca174c3e03ed1b546e4ac3a5416d28

template<class T>
bool mycmp(Point<T> a, Point<T> b) { // hash-cpp-2
  // return atan2(a.y, a.x) < atan2(b.y, b.x);
  if (a.x * b.x < 0) return a.x < 0;
  if (abs(a.x) < eps) {
    if (abs(b.x) < eps) return a.y > 0 && b.y < 0;
    if (b.x < 0) return a.y > 0;
    if (b.x > 0) return true;
  }
  if (abs(b.x) < eps) {
    if (a.x < 0) return b.y < 0;
    if (a.x > 0) return false;
  }
  return a.cross(b) > 0;
} // hash-cpp-2 = 5a80cc8032965e28a1894939bb91f3ec

bool cmp(Line a, Line b) { return mycmp(a.dir(), b.dir());
  ↪}
double Intersection_Area(vector <Line> b) { // hash-cpp-3
  sort(b.begin(), b.end(), cmp);
  int n = b.size();
  int q = 1, h = 0, i;
  vector<Line> c(b.size() + 10);
  for (i = 0; i < n; i++) {
```

```
    while (q < h && b[i].out(c[h].intpo(c[h - 1]))) h--;
    while (q < h && b[i].out(c[q].intpo(c[q + 1]))) q++;
    c[++h] = b[i];
    if (q < h && abs(c[h].dir().cross(c[h - 1].dir())) <
       ↪eps) {
      h--;
      if (b[i].out(c[h].P1))  c[h] = b[i];
    }
  }
  while (q < h - 1 && c[q].out(c[h].intpo(c[h - 1]))) h--;
  while (q < h - 1 && c[h].out(c[q].intpo(c[q + 1]))) q++;
  // Intersection is empty. This is sometimes different
     ↪from the case when
  // the intersection area is 0.
  if (h - q <= 1) return 0;
  c[h + 1] = c[q];
  vector <P> s;
  for (i = q; i <= h; i++) s.push_back(c[i].intpo(c[i + 1])
     ↪);
  s.push_back(s[0]);
  double ans = 0;
  for (i = 0; i < (int)s.size()-1; i++) ans += s[i].cross(s
     ↪[i + 1]);
  return ans/2;
} // hash-cpp-3 = 42e408a367c0ed9cff988abd9b4b64ca
```

8.4 Misc. Point Set Problems

ClosestPair.h

Description: Finds the closest pair of points.
Time: $\mathcal{O}(n \log n)$

"Point.h" 16 lines

```
pair<P, P> closest(vector<P> v) {
  assert(v.size() > 1);
  set<P> S;
  sort(v.begin(), v.end(), [](P a, P b) { return a.y < b.y;
     ↪ });
  pair<int64_t, pair<P, P>> ret{LLONG_MAX, {P(), P()}};
  int j = 0;
  for(P &p : v) {
    P d{1 + (int64_t)sqrt(ret.first), 0};
    while (v[j].y <= p.y - d.x) S.erase(v[j++]);
    auto lo = S.lower_bound(p - d), hi = S.upper_bound(p +
       ↪d);
    for (; lo != hi; ++lo)
      ret = min(ret, {(*lo - p).dist2(), {*lo, p}});
    S.insert(p);
  }
  return ret.second;
} // hash-cpp-all = 32b14fb38db71dbab5ee948422612570
```

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(vp.begin(),vp.end(), 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 = vp.size()/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({vp.begin(),
     ↪vp.end()})) {}

  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);
  }
}; // hash-cpp-all = 915562277c057ca45f507138a826fa7d
```

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

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

```
template<class P, class F>
void delaunay(vector<P>& ps, F trifun) {
  if (ps.size() == 3) { int d = (ps[0].cross(ps[1], ps[2])
     ↪< 0);
    trifun(0,1+d,2-d); }
  vector<P3> p3;
  for(auto &p : ps) p3.emplace_back(p.x, p.y, p.dist2());
  if (ps.size() > 3) for(auto &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);
} // hash-cpp-all = f6175a3c9680ae285374fb369c3af995
```

FastDelaunay.h

Description: Fast Delaunay triangulation. There must be no duplicate points. If all points are on a line, no triangles will be returned. Should work for doubles as well, though there may be precision issues in 'circ'. Returns triangles in order {t[0][0], t[0][1], t[0][2], t[1][0], ... }, all counter-clockwise.
Time: $\mathcal{O}(n \log n)$

"Point.h" 90 lines

```
typedef Point<ll> P;
typedef struct Quad* Q;
typedef __int128_t lll; // (can be ll if coords are < 2e4)
P arb(LLONG_MAX,LLONG_MAX); // not equal to any other point

struct Quad { // hash-cpp-1
  bool mark; Q o, rot; P p;
  P F() { return r()->p; }
  Q r() { return rot->rot; }
  Q prev() { return rot->o->rot; }
  Q next() { return rot->r()->o->rot; }
}; // hash-cpp-1 = ae7c00e56c665d4b1231ab65e4a209f7
// hash-cpp-2
bool circ(P p, P a, P b, P c) { // is p in the circumcircle
   ↪?
  lll p2 = p.dist2(), A = a.dist2()-p2,
      B = b.dist2()-p2, C = c.dist2()-p2;
  return p.cross(a,b)*C + p.cross(b,c)*A + p.cross(c,a)*B >
     ↪ 0;
} // hash-cpp-2 = 6aff7b12fbc9bf3e4cdc9425f5a62137
Q makeEdge(P orig, P dest) { // hash-cpp-3
  Q q0 = new Quad{0,0,0,orig}, q1 = new Quad{0,0,0,arb},
    q2 = new Quad{0,0,0,dest}, q3 = new Quad{0,0,0,arb};
  q0->o = q0; q2->o = q2; // 0-0, 2-2
  q1->o = q3; q3->o = q1; // 1-3, 3-1
  q0->rot = q1; q1->rot = q2;
  q2->rot = q3; q3->rot = q0;
  return q0;
} // hash-cpp-3 = 81016dffd34a695006075996590c4d6a
void splice(Q a, Q b) { // hash-cpp-4
  swap(a->o->rot->o, b->o->rot->o); swap(a->o, b->o);
}
Q connect(Q a, Q b) {
  Q q = makeEdge(a->F(), b->p);
  splice(q, a->next());
  splice(q->r(), b);
  return q;
} // hash-cpp-4 = 7e71f74a90f6e01fedeeb98e1fcb3d65

pair<Q,Q> rec(const vector<P>& s) { // hash-cpp-5
  if (sz(s) <= 3) {
    Q a = makeEdge(s[0], s[1]), b = makeEdge(s[1], s.back()
       ↪);
    if (sz(s) == 2) return { a, a->r() };
    splice(a->r(), b);
    auto side = s[0].cross(s[1], s[2]);
    Q c = side ? connect(b, a) : 0;
    return {side < 0 ? c->r() : a, side < 0 ? c : b->r() };
```

```
  }

#define H(e) e->F(), e->p
#define valid(e) (e->F().cross(H(base)) > 0)
  Q A, B, ra, rb;
  int half = (sz(s) + 1) / 2;
  tie(ra, A) = rec({s.begin(), s.begin() + half});
  tie(B, rb) = rec({s.begin() + half, s.end()});
  while ((B->p.cross(H(A)) < 0 && (A = A->next())) ||
         (A->p.cross(H(B)) > 0 && (B = B->r()->o)));
  Q base = connect(B->r(), A);
  if (A->p == ra->p) ra = base->r();
  if (B->p == rb->p) rb = base;

#define DEL(e, init, dir) Q e = init->dir; if (valid(e)) \
    while (circ(e->dir->F(), H(base), e->F())) { \
      Q t = e->dir; \
      splice(e, e->prev()); \
      splice(e->r(), e->r()->prev()); \
      e = t; \
    }
  for (;;) {
    DEL(LC, base->r(), o);  DEL(RC, base, prev());
    if (!valid(LC) && !valid(RC)) break;
    if (!valid(LC) || (valid(RC) && circ(H(RC), H(LC))))
      base = connect(RC, base->r());
    else
      base = connect(base->r(), LC->r());
  }
  return { ra, rb };
} // hash-cpp-5 = d3b6931a24cfd32c9af3573423c14605

vector<P> triangulate(vector<P> pts) { // hash-cpp-6
  sort(pts.begin(), pts.end());  assert(unique(pts.begin(),
    ↪ pts.end()) == pts.end());
  if (pts.size() < 2) return {};
  Q e = rec(pts).first;
  vector<Q> q = {e};
  int qi = 0;
  while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
#define ADD { Q c = e; do { c->mark = 1; pts.push_back(c->p
  ↪); \
  q.push_back(c->r()); c = c->next(); } while (c != e); }
  ADD; pts.clear();
  while (qi < sz(q)) if (!(e = q[qi++])->mark) ADD;
  return pts;
} // hash-cpp-6 = 4e0ca588db95eeafce87cd00038a4697
```

RectangleUnionArea.h

Description: Sweep line algorithm that calculates area of union of rectangles in the form $[x1, x2) \times [y1, y2)$

Usage: Create vector with both x coordinates and y coordinates of each rectangle.//vector<pair<int,int>,pair<int,int>> rectangles;// rectangles.push_back({{x1, x2}, {y1, y2}});// lint result = rectangle_union_area(rectangles);

59 lines

```
pair<int,int> operator+(const pair<int,int>& l, const pair<
  ↪int,int>& r) {
    if (l.first != r.first) return min(l,r);
    return {l.first, l.second + r.second};
}
struct segtree_t { // stores min + # of mins
    int n;
    vector<int> lazy;
    vector<pair<int,int>> tree; // set n to a power of two
    segtree_t(int _n) : n(_n), tree(2*n, {0,0}), lazy(2*n,
      ↪0) { }
    void build() {
        for(int i = n-1; i >= 1; --i)
            tree[i] = tree[i<<1] + tree[i<<1|1]; }
    void push(int v, int lx, int rx) {
        tree[v].first += lazy[v];
        if (lx != rx) {
            lazy[v<<1] += lazy[v];
            lazy[v<<1|1] += lazy[v];
        }
        lazy[v] = 0;
    }
    void update(int a, int b, int delta) { update(1,0,n-1,a
      ↪,b,delta); }
    void update(int v, int lx, int rx, int a, int b, int
      ↪delta) {
        push(v, lx, rx);
        if (b < lx || rx < a) return;
        if (a <= lx && rx <= b) {
            lazy[v] = delta; push(v, lx, rx);
        }
        else {
            int m = lx + (rx - lx)/2;
            update(v<<1, lx, m, a, b, delta);
            update(v<<1|1, m+1, rx, a, b, delta);
            tree[v] = (tree[v<<1] + tree[v<<1|1]);
        }
    }
};
lint rectangle_union_area(vector<pair<pair<int,int>,pair<
  ↪int,int>>> v) { // area of union of rectangles
    const int n = 1<<18;
    segtree_t tree(n);
    vector<int> y; for(auto &t : v) y.push_back(t.second.
      ↪first), y.push_back(t.second.second);
    sort(y.begin(), y.end()); y.erase(unique(y.begin(), y.
      ↪end()),y.end());
    for(int i = 0; i < y.size()-1; ++i) tree.tree[n+i].
      ↪second = y[i+1]-y[i]; // compress coordinates
    tree.build();
    vector<array<int,4>> ev; // sweep line
    for(auto &t : v) {
        t.second.first = lower_bound(y.begin(), y.end(),t.
          ↪second.first)-begin(y);
        t.second.second = lower_bound(y.begin(), y.end(),t.
          ↪second.second)-begin(y)-1;
        ev.push_back({t.first.first,1,t.second.first,t.
          ↪second.second});
        ev.push_back({t.first.second,-1,t.second.first,t.
          ↪second.second});
    }
    sort(ev.begin(), ev.end());
    lint ans = 0;
    for(int i = 0; i < ev.size()-1; ++i) {
        const auto& t = ev[i];
        tree.update(t[2],t[3],t[1]);
        int len = y.back()-y.front()-tree.tree[1].second;
          ↪// tree.mn[0].firstshould equal 0
        ans += (lint)(ev[i+1][0]-t[0])*len;
    }
    return ans;
} // hash-cpp-all = f66d077028ea326680fc4cc71a4a1e32
```

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;
  for(auto &i : trilist) v += p[i.a].cross(p[i.b]).dot(p[i.
    ↪c]);
  return v / 6;
} // hash-cpp-all = 832599560d46de4dac772525327508df
```

Point3D.h

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

33 lines

```
template<class T> struct Point3D { // hash-cpp-1
  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);
  } // hash-cpp-1 = f914db739064a236fa80cdd6cb4a28da
// hash-cpp-2
  T dist2() const { return x*x + y*y + z*z; }
  double dist() const { return sqrt((double)dist2()); }
  //Azimuthal angle (longitude) to x-axis in interval [-pi,
    ↪ pi]
  double phi() const { return atan2(y, x); }
  //Zenith angle (latitude) to the z-axis in interval [0,
    ↪pi]
  double theta() const { return atan2(sqrt(x*x+y*y),z); }
  P unit() const { return *this/(T)dist(); } //makes dist()
    ↪=1
  //returns unit vector normal to *this and p
  P normal(P p) const { return cross(p).unit(); }
  //returns point rotated 'angle' radians ccw around axis
  P rotate(double angle, P axis) const {
    double s = sin(angle), c = cos(angle); P u = axis.unit
      ↪();
    return u*dot(u)*(1-c) + (*this)*c - cross(u)*s;
  }
}; // hash-cpp-2 = c9d0298d203587721eca48adde037c27
```

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 { // hash-cpp-1
  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;
}; // hash-cpp-1 = cf7c9e0e504697f2f68406fa666ee3e4

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

vector<F> hull3d(const vector<P3>& A) { // hash-cpp-2
  assert(A.size() >= 4);
  vector<vector<PR>> E(A.size(), vector<PR>(A.size(), {-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);
  };
  for(int i=0;i<4;i++) for(int j=i+1;j<4;j++) for(k=j+1;k
     ↪<4;k++)
    mf(i, j, k, 6 - i - j - k);
// hash-cpp-2 = 795ac5f92c46fc81467bd587c2cbcfd5
  for(int i=4; i<A.size();++i) { // hash-cpp-3
    for(int j=0;j<FS.size();++j) {
      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 = FS.size();
    for(int j=0;j<nw;j++) {
      F f = FS[j];
#define C(a, b, c) if (E(a,b).cnt() != 2) mf(f.a, f.b, i, f
   ↪.c);
      C(a, b, c); C(a, c, b); C(b, c, a);
    }
  }
  for(auto &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;
}; // hash-cpp-3 = 58a80c2b46187dcdf3c9db71c56711db
```

SphericalDistance.h

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

8 lines

```
double sphericalDistance(double f1, double t1,
    double f2, double t2, double radius) {
  double dx = sin(t2)*cos(f2) - sin(t1)*cos(f1);
  double dy = sin(t2)*sin(f2) - sin(t1)*sin(f1);
  double dz = cos(t2) - cos(t1);
  double d = sqrt(dx*dx + dy*dy + dz*dz);
  return radius*2*asin(d/2);
} // hash-cpp-all = 611f0797307c583c66413c2dd5b3ba28
```

Strings (9)

KMP.cpp

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

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

28 lines

```
template<typename T> struct kmp_t {
    vector<T> word;
    vector<int> failure;
    kmp_t(const vector<T> &_word): word(_word) { // hash-
       ↪cpp-1
        int n = word.size();
        failure.resize(n+1, 0);
        for (int s = 2; s <= n; ++s) {
            failure[s] = failure[s-1];
            while (failure[s] > 0 && word[failure[s]] !=
               ↪word[s-1])
                failure[s] = failure[failure[s]];
            if (word[failure[s]] == word[s-1]) failure[s]
               ↪+= 1;
        }
    } // hash-cpp-1 = c66cf26827fd4607ce1cfa55401f3dea
    vector<int> matches_in(const vector<T> &text) { // hash
       ↪-cpp-2
        vector<int> result;
        int s = 0;
        for (int i = 0; i < (int)text.size(); ++i) {
            while (s > 0 && word[s] != text[i])
                s = failure[s];
            if (word[s] == text[i]) s += 1;
            if (s == (int)word.size()) {
                result.push_back(i-(int)word.size()+1);
                s = failure[s];
            }
        }
        return result;
    } // hash-cpp-2 = 50ada13bcff4322771988e39d05fffe4
};
```

Extended-KMP.h

Description: extended KMP S[i] stores the maximum common prefix between s[i:] and t; T[i] stores the maximum common prefix between t[i:] and t for i>0;

23 lines

```
int S[N], T[N];
void extKMP(const string &s, const string &t) {
    int m = t.size(), maT = 0, maS = 0;
    T[0] = 0;
    for (int i = 1; i < m; i++) {
        if (maT + T[maT] >= i)
            T[i] = min(T[i - maT], maT + T[maT] - i);
        else T[i] = 0;
        while (T[i] + i < m && t[T[i]] == t[T[i] + i])
            T[i]++;
        if (i + T[i] > maT + T[maT]) maT = i;
    }
    int n = s.size();
    for (int i = 0; i < n; i++) {
        if (maS + S[maS] >= i)
            S[i] = min(T[i - maS], maS + S[maS] - i);
        else S[i] = 0;
        while (S[i] < m && i + S[i] < n && t[S[i]] == s[S[i
           ↪] + i])
            S[i]++;
        if (i + S[i] > maS + S[maS]) maS = i;
    }
}
// hash-cpp-all = 678f728e4713de687a5065ac74ae3d66
```

Duval.h

Description: A string is called simple (or a Lyndon word), if it is strictly smaller than any of its own nontrivial suffixes.

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

27 lines

```
template <typename T>
pair<int, vector<string>> duval(int n, const T &s) { //
   ↪hash-cpp-1
    assert(n >= 1);
    // s += s  //uncomment if you need to know the min
       ↪cyclic string
    vector<string> factors; // strings here are simple and
       ↪in non-inc order
    int i = 0, ans = 0;
    while (i < n) { // until n/2 to find min cyclic string
        ans = i;
        int j = i + 1, k = i;
        while (j < n + n && !(s[j % n] < s[k % n])) {
            if (s[k % n] < s[j % n]) k = i;
            else k++;
            j++;
        }
        while (i <= k) {
            factors.push_back(s.substr(i, j-k));
            i += j - k;
        }
    }
    return {ans, factors};
    // returns 0-indexed position of the least cyclic shift
    // min cyclic string will be s.substr(ans, n/2)
} // hash-cpp-1 = cc666b9ac54cacdb7a4172ac1573d84b
template <typename T>
pair<int, vector<string>> duval(const T &s) {
    return duval((int) s.size(), s);
}
```

Z.h

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

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

16 lines

```
vector<int> Z(string& S) {
    vector<int> z(S.size());
    int l = -1, r = -1;
    for(int i = 1; i < S.size(); ++i) {
        z[i] = i >= r ? 0 : min(r - i, z[i - l]);
        while (i + z[i] < S.size() && S[i + z[i]] == S[z[i
           ↪]]])
            z[i]++;
        if (i + z[i] > r) l = i, r = i + z[i];
    }
    return z;
}
vector<int> get_prefix(string a, string b) { // hash-cpp-1
    string str = a + '@' + b;
    vector<int> k = z(str);
    return vector<int>(k.begin()+a.size()+1, k.end());
} // hash-cpp-1 = 6aa08403b9d47a6d0e421c570e0bf941
```

Manacher.h

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

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

13 lines
```
array<vector<int>, 2> manacher(const string &s) { // hash-cpp-1
  int n = s.size();
  array<vector<int>, 2> p = {vector<int>(n+1), vector<int>(n)};
  for(int z = 0; z < 2; ++z) for (int i=0,l=0,r=0; i < n; i++) {
    int t = r-i+!z;
    if (i<r) p[z][i] = min(t, p[z][l+t]);
    int L = i-p[z][i], R = i+p[z][i]-!z;
    while (L>=1 && R+1<n && s[L-1] == s[R+1])
      p[z][i]++, L--, R++;
    if (R > r) l = L, r = R;
    }
    return p;
} // hash-cpp-1 = 87e1f0950281807a59d4f6ef730e6943
```

MinRotation.h

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

8 lines
```
int min_rotation(string s) { // hash-cpp-1
  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;
} // hash-cpp-1 = 2a08fd228bd46d16ef7716c24c0a72ce
```

Trie.h

Description: Trie implementation.

37 lines
```
struct Trie {
    struct node_t {
      unordered_map<char, node_t*> child;
      int cnt = 0, prefixCnt = 0;
    } *root = new node_t();
    void add(node_t *v, const string &s) {
        node_t *cur = v;
        for (char c : s) {
            if (cur->child.count(c)) cur = cur->child[c];
            else cur = cur->child[c] = new node_t();
            cur->prefixCnt++;
        }
        cur->cnt++;
    }
    int count(node_t *v, const string &s) {
        node_t *cur = v;
        for (char c : s) {
            if (cur->child.count(c)) cur = cur->child[c];
            else return 0;
        }
        return cur->cnt;
    }
    int prefixCount(node_t *v, const string &s) {
        node_t *cur = v;
        for (char c : s) {
            if (cur->child.count(c)) cur = cur->child[c];
            else return 0;
        }
        return cur->prefixCnt;
    }
    Trie() {}
    void add(const string &s) { add(root, s); }
    bool contains(const string &s) { return count(root, s) >= 1; }
    bool hasPrefix(const string &s) { return prefixCount(root, s) >= 1; }
    int count(const string &s) { return count(root, s); }
    int prefixCount(const string &s) { return prefixCount(root, s); }
}; // hash-cpp-all = b6b21a1653330d00cf912075293c1b1b
```

TrieXOR.h

Description: Query max xor with some int in the xor$_t$*rie*

27 lines
```
template<int MX, int MXBIT>
struct xor_trie {
    int nxt[MX][2], sz[MX];// num is last node in trie
    int num = 0;
    // change 2 to 26 for lowercase letters
    xor_trie() { memset(nxt, 0, sizeof nxt), memset(sz, 0, sizeof sz); }
    // add or delete
    void add(lint x, int a = 1) {
        int cur = 0; sz[cur] += a;
        for(int i = MXBIT-1; i >= 0; --i) {
            int t = (x & (1 << i)) >> i;
            if (!nxt[cur][t]) nxt[cur][t] = ++num;
            sz[cur = nxt[cur][t]] += a;
        }
    }
    // compute max xor
    lint query(lint x) {
        if (sz[0] == 0) return INT_MIN; // no elements in trie
        int cur = 0;
        for(int i = MXBIT-1; i >= 0; --i) {
            int t = ((x & (1 << i)) >> i) ^ 1;
            if (!nxt[cur][t] || !sz[nxt[cur][t]]) t ^= 1;
            cur = nxt[cur][t]; if (t) x ^= 1lint<<i;
        }
        return x;
    }
}; // hash-cpp-all = 7cb34b6b54a5ae3b19bf5ee68d19ca74
```

HashFunction.h

Description: Simple, short and efficient hashing using pairs to reduce load factor.

<ModTemplate.h>, <PairNumTemplate.h>

40 lines
```
using num = modnum<int(1e9)+7>;
using hsh = pairnum<num, num>;
const hsh BASE(163, 311);
// uniform_int_distribution<int> MULT_DIST(0.1*MOD,0.9*MOD);
// constexpr hsh BASE(MULT_DIST(rng), MULT_DIST(rng));
struct hash_t {
    int n;
    string str;
    vector<hsh> hash, basePow;
    hash_t(const string& s) : n(s.size()), str(s), hash(n+1), basePow(n) {
        basePow[0] = 1;
        for (int i = 1; i < n; ++i) basePow[i] = basePow[i-1] * BASE;
        for (int i = 0; i < n; ++i)
            hash[i+1] = hash[i] * BASE + hsh(s[i]);
    }
    hsh get_hash(int left, int right) {
        assert(left <= right);
        return hash[right] - hash[left] * basePow[right - left];
    }
    int lcp(hash_t &other) { // need some testing
        int left = 0, right = min(str.size(), other.str.size());
        while (left < right) {
            int mid = (left + right + 1)/2;
            if (hash[mid] == other.hash[mid]) left = mid;
            else right = mid-1;
        }
        return left;
    }
};
vector<int> rabinkarp(string t, string p) {
    vector<int> matches;
    hsh h(0, 0);
    for (int i = 0; i < p.size(); ++i)
        h = BASE * h + hsh(p[i]);
    hash_t result(t);
    for (int i = 0; i + p.size() <= t.size(); ++i)
        if (result.get_hash(i, i + p.size()) == h)
            matches.push_back(i);
    return matches;
} // hash-cpp-all = 13369fd0b40b2a886e961bb6f6014dce
```

SuffixTree.h

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

47 lines
```
struct SuffixTree {
  enum { N = 200010, ALPHA = 26 }; // N ~ 2*maxlen+10
  int toi(char c) { return c - 'a'; }
  string a; // v = cur node, q = cur position
  int t[N][ALPHA],l[N],r[N],p[N],s[N],v=0,q=0,m=2;
  void ukkadd(int i, int c) { suff:
    if (r[v]<=q) {
      if (t[v][c]==-1) { t[v][c]=m;  l[m]=i;
        p[m++]=v; v=s[v]; q=r[v];  goto suff; }
      v=t[v][c]; q=l[v];
    }
    if (q==-1 || c==toi(a[q])) q++; else {
      l[m+1]=i;  p[m+1]=m;  l[m]=l[v];  r[m]=q;
      p[m]=p[v];  t[m][c]=m+1;  t[m][toi(a[q])]=v;
      l[v]=q;  p[v]=m;  t[p[m]][toi(a[l[m]])]=m;
      v=s[p[m]];  q=l[m];
      while (q<r[m]) { v=t[v][toi(a[q])];  q+=r[v]-l[v]; }
      if (q==r[m])  s[m]=v;  else s[m]=m+2;
      q=r[v]-(q-r[m]);  m+=2;  goto suff;
    }
  }
  SuffixTree(string a) : a(a) {
    fill(r,r+N,a.size());
    memset(s, 0, sizeof s);
    memset(t, -1, sizeof t);
    fill(t[1],t[1]+ALPHA,0);
    s[0] = 1; l[0] = l[1] = -1; r[0] = r[1] = p[0] = p[1] = 0;
    for(int i = 0; i < a.size(); ++i) ukkadd(i, toi(a[i]));
  }
```

```
  // example: find longest common substring (uses ALPHA =
     ↪28)
  pair<int,int> best;
  int lcs(int node, int i1, int i2, int olen) {
    if (l[node] <= i1 && i1 < r[node]) return 1;
    if (l[node] <= i2 && i2 < r[node]) return 2;
    int mask = 0, len = node ? olen + (r[node] - l[node]) :
       ↪ 0;
    for(int c = 0; c < ALPHA; ++c) if (t[node][c] != -1)
      mask |= lcs(t[node][c], i1, i2, len);
    if (mask == 3)
      best = max(best, {len, r[node] - len});
    return mask;
  }
  static pair<int,int> LCS(string s, string t) {
    SuffixTree st(s + (char)('z' + 1) + t + (char)('z' + 2)
       ↪);
    st.lcs(0, s.size(), s.size() + 1 + t.size(), 0);
    return st.best;
  }
}; // hash-cpp-all = 6c2a8bdd2a7412aab755d53b9d18fdc5
```

AhoCorasick.cpp

Description: String searching algorithm that matches all strings simultaneously. To use with stl string: (char *)stringname.c_str() 87 lines

```
struct node_t {
    int fail;
    vector<pair<int,int>> out; // num e tamanho do padrao
    //bool marc;  // p/ decisao
    map<char,int> link;
  int next; // aponta para o proximo sufixo que tenha out.
     ↪size > 0
};
node_t tree[1000003]; // quantida maxima de nos
struct AhoCorasick {
  //bool encontrado[1005]; // quantidade maxima de padroes,
     ↪ p/ decisao
  int qtdNos, qtdPadroes;
  vector<vector<int>> result;
  AhoCorasick() { // Construtor para inicializar
    result.resize(0);
      tree[0].fail = -1;
      tree[0].link.clear();
      tree[0].out.clear();
      tree[0].next = -1;
      qtdNos = 1;
      qtdPadroes = 0;
      //tree[0].marc = false; // p/ decisao
      //memset(encontrado, false, sizeof(encontrado)); // p
         ↪/ decisao
  }
  void add(string &pat) {
    vector<int> v;
    result.push_back(v);
      int no = 0, len = 0;
      for (int i = 0; i < pat.size(); ++i, ++len) {
          if (tree[no].link.find(pat[i]) == tree[no].link.
             ↪end()) {
              tree[qtdNos].link.clear(); tree[qtdNos].out.
                 ↪clear();
              //tree[qtdNos].marc = false; // p/ decisao
              tree[no].link[pat[i]] = qtdNos;
              no = qtdNos++;
          } else no = tree[no].link[pat[i]];
      }
      tree[no].out.push_back({++qtdPadroes,len});
  }
  void activate() {
      int no, v, f, w;
      vector<int> bfs;
      for (auto  it = tree[0].link.begin();
         it != tree[0].link.end() ; ++it) {
          tree[no = it->second].fail = 0;
          tree[no].next = tree[0].out.size() ? 0 : -1;
          bfs.push_back(no);
      }
      for (int i = 0; i < bfs.size(); ++i) {
          no = bfs[i];
          for (auto it = tree[no].link.begin();
               it != tree[no].link.end(); it++) {
              char c = it->first;
              v = it->second;
              bfs.push_back(v);
              f = tree[no].fail;
              while (tree[f].link.find(c) == tree[f].link.
                 ↪end()) {
                  if (f == 0) { tree[0].link[c] = 0; break;
                     ↪ }
                  f = tree[f].fail;
              }
              w = tree[f].link[c];
              tree[v].fail = w;
              tree[v].next = tree[w].out.size() ? w : tree[
                 ↪w].next;
          }
      }
  }
  void matches(string& text) {
      int v, no = 0;
      for (int i = 0; i < text.size(); ++i) {
          while (tree[no].link.find(text[i]) == tree[no].
             ↪link.end()) {
              if (no == 0) { tree[0].link[text[i]] = 0;
                 ↪break; }
              no = tree[no].fail;
          }
          v = no = tree[no].link[text[i]];
          // marcar os encontrados
          while (v != -1 /* && !tree[v].marc */) { // p/
             ↪decisao
              //tree[v].marc = true; // p/ decisao: nao
                 ↪continua a link
              for (int k = 0 ; k < tree[v].out.size() ; k
                 ↪++) {
                  //encontrado[tree[v].out[k].first] = true
                     ↪; // p/ decisao
                  result[tree[v].out[k].first].push_back(i-
                     ↪tree[v].out[k].second+1);
                  printf("Padrao %d na posicao %d\n", tree[
                     ↪v].out[k].first,
                          i-tree[v].out[k].second+1);
              }
              v = tree[v].next;
          }
      }
  }
};
// hash-cpp-all = c54b4bd636023d367718352ccbca23a6
```

Suffix-Array.h

Description: Builds suffix array for a string. `sa[i]` is the starting index of the suffix which is i'th in the sorted suffix array. The returned vector is of size $n+1$, and `sa[0] = n`. The `lcp` array contains longest common prefixes for neighbouring strings in the suffix array: `lcp[i] = lcp(sa[i], sa[i-1])`, `lcp[0] = 0`. The input string must not contain any zero bytes.
Time: $\mathcal{O}(n \log n)$ 23 lines

```
struct SuffixArray {
  vector<int> sa, lcp;
  SuffixArray(string& s, int lim=256) { // or basic_string<
     ↪int>
    int n = s.size()+1, k = 0, a, b;
    vector<int> x(s.begin(), s.end()+1), y(n), ws(max(n,
       ↪lim)), rank(n);
    sa = lcp = y, iota(sa.begin(), sa.end(), 0);
    for (int j = 0, p = 0; p < n; j = max(1, j * 2), lim =
       ↪p) {
      p = j, iota(y.begin(), y.end(), n - j);
      for(int i=0;i<n;++i) if (sa[i] >= j) y[p++] = sa[i] -
         ↪ j;
      fill(ws.begin(), ws.end(), 0);
      for(int i=0;i<n;++i) ws[x[i]]++;
      for(int i=1;i<lim;++i) ws[i] += ws[i - 1];
      for (int i=n;i--;) sa[--ws[x[y[i]]]] = y[i];
      swap(x, y), p = 1, x[sa[0]] = 0;
      for(int i=1;i<n;++i) a = sa[i - 1], b = sa[i], x[b] =
        (y[a] == y[b] && y[a + j] == y[b + j]) ? p - 1 : p
           ↪++;
    }
    for(int i=1;i<n;++i) rank[sa[i]] = i;
    for (int i=0,j;i<n-1;lcp[rank[i++]]=k)
      for (k && k--, j = sa[rank[i] - 1];
           s[i + k] == s[j + k]; k++);
  }
}; // hash-cpp-all = dc6caa155393cfe4a922768e1a0c851d
```

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: $\mathcal{O}(\log N)$ 23 lines

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

void removeInterval(set<pair<int,int>> &is, int L, int R) {
  if (L == R) return;
  auto it = addInterval(is, L, R);
  auto r2 = it->second;
```

```
  if (it->first == L) is.erase(it);
  else (int&)it->second = L;
  if (R != r2) is.emplace(R, r2);
} // hash-cpp-all = f47dfb9edd525539da08472171658898
```

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>
vector<int> cover(pair<T, T> G, vector<pair<T, T>> I) {
  vector<int> S(I.size()), R;
  iota(S.begin(), S.end(), 0);
  sort(S.begin(), S.end(), [&](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 = {cur, -1};
    while (at < I.size() && I[S[at]].first <= cur) {
      mx = max(mx, {I[S[at]].second, S[at]});
      at++;
    }
    if (mx.second == -1) return {};
    cur = mx.first;
    R.push_back(mx.second);
  }
  return R;
} // hash-cpp-all = 133eb4becbdaf3b99371a1e364b33a2b
```

ConstantIntervals.h

Description: Split a monotone function on [from, to) into a minimal set of half-open intervals on which it has the same value. Runs a callback g for each such interval.
Usage: `constantIntervals(0, sz(v), [&](int x){return v[x];}, [&](int lo, int hi, T val){...});`
Time: $\mathcal{O}\left(k \log \frac{n}{k}\right)$

19 lines

```
template<class F, class G, class T>
void rec(int from, int to, F f, G g, int& i, T& p, T q) {
  if (p == q) return;
  if (from == to) {
    g(i, to, p);
    i = to; p = q;
  } else {
    int mid = (from + to) >> 1;
    rec(from, mid, f, g, i, p, f(mid));
    rec(mid+1, to, f, g, i, p, q);
  }
}
template<class F, class G>
void constantIntervals(int from, int to, F f, G g) {
  if (to <= from) return;
  int i = from; auto p = f(i), q = f(to-1);
  rec(from, to-1, f, g, i, p, q);
  g(i, to, q);
} // hash-cpp-all = 792e7d94c54ab04f9efdb6834b12feca
```

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 = ternSearch(0,n-1,[&](int i){return a[i];});`
Time: $\mathcal{O}(\log(b-a))$

12 lines

```
template<class F>
int ternSearch(int a, int b, F 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;
} // hash-cpp-all = 35ef733e0d208d74285a33ac625eb20c
```

LowerBound.h

11 lines

```
int LowerBound(vector<int> v, int n, int x){
    int l = 1, r = n, m;
    while(l <= r){
        m= (l+r)/2;
        if(v[m] >= x && (m == 1 || v[m-1] < x))
            return m;
        else if(v[m] >= x) r=m-1;
        else l=m+1;
    }
    return m;
} // hash-cpp-all = 7422d7a27dbb4142bd13b8cc1f0f3686
```

UpperBound.h

11 lines

```
int UpperBound(vector<int> v, int n, int x){
    int l = 1, r = n, m;
    while(l <= r){
        m=(l+r)/2;
        if(v[m] > x && (m == 1 || v[m-1] <= x))
            return m;
        else if(v[m] > x) r=m-1;
        else l=m+1;
    }
    return m;
} // hash-cpp-all = 381d15e1acc45839a99189533b42d5eb
```

MergeSort.h

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

25 lines

```
vector<int> merge(vector<int> &values, int l, int r) {
    static vector<int> result(values.size());
    int i = l, j = l + (r - l)/2;
    int mid = j, k = i, inversions = 0;
    while (i < mid && j < r) {
        if (values[i] < values[j]) result[k++] = values[i
          ↪++];
        else {
            result[k++] = values[j++];
            inversions += (mid - i);
        }
    }
    while (i < mid) result[k++] = values[i++];
    while (j < r) result[k++] = values[j++];
    for (k = l; k < r; ++k) values[k] = result[k];
    return result;
}

vector<int> msort(vector<int> &values, int l, int r) {
    if (r - l > 1) {
        int mid = l + (r - l)/2;
        msort(values, l, mid); msort(values, mid, r);
        return merge(values, l, r);
    }
    return {};
} // hash-cpp-all = fac159b88d21579cb8ae468bde7adfe6
```

CoordCompression.h

9 lines

```
vector<int> comp_coord(vector<int> &y, int N) {
    vector<int> result;
    for (int i = 0; i < N; ++i) result.emplace_back(y[i]);
    sort(result.begin(), result.end());
    result.resize(unique(result.begin(), result.end())-
      ↪result.begin());
    for (int i = 0; i < N; ++i)
        y[i] = lower_bound(result.begin(), result.end(), y[
          ↪i]) - result.begin();
    return result;
} // hash-cpp-all = 809d6ae9d2b00e4d11b3e8500c82eb70
```

CountTriangles.h

Description: Counts x, y >= 0 such that Ax + By <= C.

8 lines

```
lint count_triangle(lint A, lint B, lint C) {
  if (C < 0) return 0;
  if (A > B) swap(A, B);
  lint p = C / B;
  lint k = B / A;
  lint d = (C - p * B) / A;
  return count_triangle(B - k * A, A, C - A * (k * p + d +
    ↪1)) + (p + 1) * (d + 1) + k * p * (p + 1) / 2;
} // hash-cpp-all = 8d67b384e4591dd4f0ba9538ad3bc5d9
```

sqrt.h

9 lines

```
template<typename T> T isqrt(T n) {
    T left = 0, right = 10000000;
    while (right - left > 1) {
        T mid = (left + right)/2;
        if (mid * mid <= n) left = mid;
        else right = mid;
    }
    return left;
} // hash-cpp-all = 939108303562838487197520bf6deb2e
```

Karatsuba.h

Description: Faster-than-naive convolution of two sequences: $c[x] = \sum a[i]b[x-i]$. Uses the identity $(aX+b)(cX+d) = acX^2 + bd + ((a+c)(b+d) - ac - bd)X$. Doesn't handle sequences of very different length welint. See also FFT, under the Numerical chapter.
Time: $\mathcal{O}\left(N^{1.6}\right)$

30 lines

```
int size(int s) { return s > 1 ? 32-__builtin_clz(s-1) : 0;
  ↪ }

void karatsuba(lint *a, lint *b, lint *c, lint *t, int n) {
    int ca = 0, cb = 0;
    for(int i = 0; i < n; ++i) ca += !!a[i], cb += !!b[i];
    if (min(ca, cb) <= 1500/n) { // few numbers to multiply
        if (ca > cb) swap(a, b);
        for(int i = 0; i < n; ++i)
            if (a[i]) FOR(j,n) c[i+j] += a[i]*b[j];
```

```
    }
    else {
        int h = n >> 1;
        karatsuba(a, b, c, t, h); // a0*b0
        karatsuba(a+h, b+h, c+n, t, h); // a1*b1
        for(int i = 0; i < h; ++i) a[i] += a[i+h], b[i] +=
            ↪b[i+h];
        karatsuba(a, b, t, t+n, h); // (a0+a1)*(b0+b1)
        for(int i = 0; i < h; ++i) a[i] -= a[i+h], b[i] -=
            ↪b[i+h];
        for(int i = 0; i < n; ++i) t[i] -= c[i]+c[i+n];
        for(int i = 0; i < n; ++i) c[i+h] += t[i], t[i] =
            ↪0;
    }
}

vector<lint> conv(vector<lint> a, vector<lint> b) {
    int sa = a.size(), sb = b.size(); if (!sa || !sb)
        ↪return {};
    int n = 1<<size(max(sa,sb)); a.resize(n), b.resize(n);
    vector<lint> c(2*n), t(2*n);
    for(int i = 0; i < 2*n; ++i) t[i] = 0;
    karatsuba(&a[0], &b[0], &c[0], &t[0], n);
    c.resize(sa+sb-1); return c;
} // hash-cpp-all = 94626586a3d1b8e95703da4c97fb6c83
```

CountInversions.h

Description: Count the number of inversions to make an array sorted. Merge sort has another approach.
Time: $\mathcal{O}(n * log(n))$

<FenwickTree.h> 22 lines

```
FT<int, 10010> bit;
int inv = 0;
for (int i = n-1; i >= 0; --i) {
    inv += bit.sum(values[i]); // careful with the interval
    bit.update(values[i], 1);    // [0, x) or [0, x] ?
}

// using D&C,the constant is quite high but still nlogn
lint msort(vector<int> &values, int left, int right) {
  if ((right - left) <= 1) return 0;
  int mid = left + (right - left)/2;
  lint result = msort(values, left, mid) + msort(values,
      ↪mid, right);
  auto cmp = [](int i, int j) { return i > j; };
  sort(values.begin() + left, values.begin() + mid, cmp);
  sort(values.begin() + mid, values.begin() + right, cmp);
  int pos = left;
  for (int i = mid; i < right; ++i) {
    while (pos != mid && values[pos] > values[i]) ++pos;
    result += (pos - left);
  }
  return result;
} // hash-cpp-all = a5135a66a13dad5edbd35c77640ac241
```

Histogram.h

Description: Maximum area of a histogram.
Time: $\mathcal{O}(n)$

16 lines

```
template<typename T = int>
T max_area(vector<int> v) {
    T ret = T();
    stack<int> s;
    v.insert(v.begin(), -1);
    v.insert(v.end(), -1);
    s.push(0);
    for(int i = 0; i < v.size(); ++i) {
        while (v[s.top()] > v[i]) {
            int h = v[s.top()]; s.pop();
            ret = max(ret, h * (i - s.top() - 1));
        }
        s.push(i);
    }
    return ret;
} // hash-cpp-all = 51da1acb56b7aba5750ea9568e40ba1b
```

DateManipulation.h

43 lines

```
string week_day_str[7] = {"Sunday", "Monday", "Tuesday", "
    ↪Wednesday", "Thursday", "Friday", "Saturday"};
string month_str[13] = {"", "January", "February", "March",
    ↪ "April", "May", "June", "July", "August", "September"
    ↪, "October", "November", "December"};
map<string, int> week_day_int = {{"Sunday", 0}, {"Monday",
    ↪1}, {"Tuesday", 2}, {"Wednesday", 3}, {"Thursday", 4},
    ↪ {"Friday", 5}, {"Saturday", 6}};
map<string, int> month_int = {{"January", 1}, {"February",
    ↪2}, {"March", 3}, {"April", 4}, {"May", 5}, {"June",
    ↪6}, {"July", 7}, {"August", 8}, {"September", 9}, {"
    ↪October", 10}, {"November", 11}, {"December", 12}};
int month[2][13] = {{0, 31, 28, 31, 30, 31, 30, 31, 31, 30,
    ↪ 31, 30, 31}, {0, 31, 29, 31, 30, 31, 30, 31, 31, 30,
    ↪31, 30, 31}};

/* O(1) - Checks if year y is a leap year. */
bool leap_year(int y){
  return (y % 4 == 0 && y % 100 != 0) || y % 400 == 0;
}

/* O(1) - Increases the day by one. */
void update(int &d, int &m, int &y){
  if (d == month[leap_year(y)][m]){
    d = 1;
    if (m == 12) {
      m = 1;
      y++;
    }
    else m++;
  }
  else d++;
}

int intToDay(int jd) { return jd % 7; }
int dateToInt(int y, int m, int d) {
  return 1461 * (y + 4800 + (m - 14) / 12) / 4 +
    367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
    3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 +
    d - 32075; }
void intToDate(int jd, int &y, int &m, int &d) {
  int x, n, i, j;
  x = jd + 68569;
  n = 4 * x / 146097;
  x -= (146097 * n + 3) / 4;
  i = (4000 * (x + 1)) / 1461001;
  x -= 1461 * i / 4 - 31;
  j = 80 * x / 2447;
  d = x - 2447 * j / 80;
  x = j / 11;
  m = j + 2 - 12 * x;
  y = 100 * (n - 49) + i + x; }
// hash-cpp-all = 0884598494c930f822d30e062be1cceb
```

NQueens.cpp

Description: NQueens

43 lines

```
int ans;
bitset<30> rw, ld, rd; //2*MAX_N -1
bitset<30> iniqueens; //2*MAX_N -1
vector<int> col;
void init(int n){
  ans=0;
    rw.reset();
    ld.reset();
    rd.reset();
    col.assign(n,-1);
}
void init(int n, vector<pair<int,int>> initial_queens){
    //it does NOT check if initial queens are at valid
        ↪positions
    init(n);
    iniqueens.reset();
    for(pair<int,int> pos: initial_queens){
        int r=pos.first, c= pos.second;
        rw[r] = ld[r-c+n-1] = rd[r+c]=true;
        col[c]=r;
        iniqueens[c] = true;
    }
}
void backtracking(int c, int n){
    if(c==n){
      ans++;
        for(int r:col) cout<<r+1<<" ";
        cout<<"\n";
        return;
    }
    else if(iniqueens[c]){
        backtracking(c+1,n);
    }
    else for(int r=0;r<n;r++){
        if(!rw[r] && !ld[r-c+n-1] && !rd[r+c]){
        // if(board[r][c]!=blocked && !rw[r] && !ld[r-c+n
            ↪-1] && !rd[r+c]){ // if there are blocked
            ↪possitions
            rw[r] = ld[r-c+n-1] = rd[r+c]=true;
            col[c]=r;
            backtracking(c+1,n);
            col[c]=-1;
            rw[r] = ld[r-c+n-1] = rd[r+c]=false;
        }
    }
} // hash-cpp-all = e97e9e9198bfdafeb93f5b1021de2577
```

SudokuSolver.h

42 lines

```
int N,m; // N = n*n, m = n; where n equal number of rows or
    ↪ columns
array<array<int, 10>, 10> grid;
struct SudokuSolver {
    bool UsedInRow(int row,int num){
        for(int col = 0; col < N; ++col)
            if(grid[row][col] == num) return true;
        return false;
    }
    bool UsedInCol(int col,int num){
        for(int row = 0; row < N; ++row)
            if(grid[row][col] == num) return true;
        return false;
    }
    bool UsedInBox(int row_0,int col_0,int num){
        for(int row = 0; row < m; ++row)
```

```
            for(int col = 0; col < m; ++col)
                if(grid[row+row_0][col+col_0] == num)
                    return true;
        return false;
    }
    bool isSafe(int row,int col,int num){
        return !UsedInRow(row,num) && !UsedInCol(col,num)
            && !UsedInBox(row-row%m,col-col%m,num);
    }
    bool find(int &row,int &col){
        for(row = 0; row < N; ++row)
            for(col = 0; col < N; ++col)
                if(grid[row][col] == 0) return true;
        return false;
    }
    bool Solve(){
        int row, col;
        if(!find(row,col)) return true;
        for(int num = 1; num <= N; ++num){
            if(isSafe(row,col,num)){
                grid[row][col] = num;
                if(Solve()) return true;
                grid[row][col] = 0;
            }
        }
        return false;
    }
};
// hash-cpp-all = 6be9065d036cb0cb4f35ee043083f733
```

FloydCycle.h

Description: Detect loop in a list. Consider using mod template to avoid overflow.
Time: $\mathcal{O}(n)$

10 lines

```
template<class F>
pair<int,int> find(int x0, F f) {
    int t = f(x0), h = f(t), mu = 0, lam = 1;
    while (t != h) t = f(t), h = f(f(h));
    h = x0;
    while (t != h) t = f(t), h = f(h), ++mu;
    h = f(t);
    while (t != h) h = f(h), ++lam;
    return {mu, lam};
} // hash-cpp-all = b456abce4eee035d72fa359eb8704ec0
```

SubsetXOR.h

Description: Given an array compute the maximum/minimum subset xor.

21 lines

```
template<typename T> struct XorGauss {
    int n; vector<T> a;
    XorGauss(int bits) : n(bits), a(bits) {}
    T reduce(T x) {
        for(int i = n-1; i >= 0; i--)
            x = max(x, x ^ a[i]);
        return x;
    }
    T augment(T x) { return ~reduce(~x); }
    bool add(T x) {
        for(int i = n-1; i >= 0; i--) {
            if (!(x & (1ll << i))) continue;
            if (a[i]) x ^= a[i];
            else {
                a[i] = x;
                return true;
            }
        }
        return false;
    }
}; // hash-cpp-all = a722525d26898448fb9ea60924a780ec
```

10.3 Dynamic programming

DivideAndConquerDP.h

Description: Optimizes dp of the form (or similar) $dp[i][j] = min_{k<i}(dp[k][j-1] + f(k+1,i))$. The classical case is a partitioning dp, where k determines the break point for the next partition. In this case, i is the number of elements to partition and j is the number of partitions allowed.
Let $opt[i][j]$ be the values of k which minimize the function. (in case of tie, choose the smallest) To apply this optimization, you need $opt[i][j] \leq opt[i+1][j]$. That means the when you add an extra element $(i+1)$, your partitioning choice will not be to include more elements than before (e.g. will no go from choosing $[k,i]$ to $[k-1,i+1]$). This is usually intuitive by the problem details.
. To apply try to write the dp in the format above and verify if the property holds.
Time: Time goes from $\mathcal{O}\left(n^2 m\right)$ to $\mathcal{O}\left(nm\log(n)\right)$

54 lines

```
const int INF = 1<<31;
int n, m;
template<typename MAXN, typename MAXM>
struct dp_task {
    array<array<int, MAXN>, MAXN> u;
    array<array<int, MAXN>, MAXM> dp;
    inline f(int i, int j) {
        return (u[j][j] - u[j][i-1] - u[i-1][j] + u[i-1][i
            -1]) / 2;
    }
    // This is responsible for computing tab[1...r][j],
        knowing that opt[1...r][j] is in range [low_opt...
        high_opt]
    void solve(int j, int l, int r, int low_opt, int
        high_opt) {
        int mid = (l + r) / 2, opt = -1;
        dp[mid][j] = INF;
        for (int k = low_opt; k <= high_opt && k < mid; ++k
            )
            if (dp[k][j-1] + f(k + 1, mid) < dp[mid][j]) {
                dp[mid][j] = dp[k][j-1] + f(k + 1, mid);
                opt = k;
            }
      // New bounds on opt for other pending computation.
      if (l <= mid - 1)
        solve(j, l, mid - 1, low_opt, opt);
      if (mid + 1 <= r)
        solve(j, mid + 1, r, opt, high_opt);
    }
};

int main() {
    dp_task<4123, 812> DP;
    cin >> n >> m;
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
            cin >> DP.u[i][j];

  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      DP.u[i][j] += DP.u[i - 1][j] + DP.u[i][j - 1] - DP.u[
          i - 1][j - 1];

  for (int i = 1; i <= n; i++)
    DP.dp[i][0] = INF;

  // Original dp
  // for (int i = 1; i <= n; i++)
  //  for (int j = 1; j <= m; j++) {
  //    dp[i][j] = INF;
  //    for (int k = 0; k < i; k++)
  //      dp[i][j] = min(dp[i][j], dp[k][j-1] + f(k + 1,i));
  //  }

  for (int j = 1; j <= m; j++)
    DP.solve(j, 1, n, 0, n - 1);

  cout << DP.dp[n][m] << endl;
}
// hash-cpp-all = 9963fb5a8ba5aaa937eb85e7d242a141
```

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 $a \leq b \leq c \leq d$. Generally, Optimizes dp of the form (or similar) $dp[i][j] = min_{i<=k<=j}(dp[i][k-1]+dp[k+1][j]+f(i,j))$. The classical case is building a optimal binary tree, where k determines the root. Let $opt[i][j]$ be the value of k which minimizes the function. (in case of tie, choose the smallest) To apply this optimization, you need $opt[i][j-1] \leq opt[i][j] \leq opt[i+1][j]$. That means the when you remove an element form the left $(i+1)$, you won't choose a breaking point more to the left than before. Also, when you remove an element from the right $(j-1)$, you won't choose a breking point more to the right than before. This is usually intuitive by the problem details. To apply try to write the dp in the format above and verify if the property holds. Be careful with edge cases for opt. Consider also: LineContainer (ch. Data structures), monotone queues, ternary search.
Time: from $\mathcal{O}\left(N^3\right)$ to $\mathcal{O}\left(N^2\right)$

30 lines

```
array<array<lint, 1123>, 1123> dp;
array<array<int, 1123>, 1123> opt;
array<int, 1123> b;
int l, n;

inline f(int i, int j) {
    return b[j+1] - b[i-1];
}

int main() {
    while(cin >> l >> n) {
        for (int i = 1; i <= n; ++i) cin >> b[i];
        b[0] = 0;
        b[n + 1] = l;
        for (int i = 1; i <= n+1; ++i) {
            dp[i][i - 1] = 0
            opt[i][i - 1] = i;
        }
        for (int i = n; i > 0; --i)
            for (int j = i; j <= n; ++j) {
                dp[i][j] = LLONG_MAX; // INF
                for (int k = max(i, opt[i][j - 1]); k <= j
                    && k <= opt[i + 1][j]; ++k)
                    if (dp[i][k - 1] + dp[k + 1][j] + f(i,
                        j) < dp[i][j]) {
                        dp[i][j] = dp[i][k - 1] + dp[k +
                            1][j] + f(i, j);
                        opt[i][j] = k;
                    }
            }
```

```
        cout << dp[1][n] << '\n';
    }
} // hash-cpp-all = 134f0b098e66977bad949936d0b80a0b
```

ConvexHullTrick.h

Description: Transforms dp of the form (or similar) $dp[i] = min_{j<i}(dp[j] + b[j] * a[i])$. Time goes from $O(n^2)$ to $O(n \log n)$, if using online line container, or $O(n)$ if lines are inserted in order of slope and queried in order of x. To apply try to find a way to write the factor inside minimization as a linear function of a value related to i. Everything else related to j will become constant.

<LineContainer.h> 22 lines

```
array<lint, 112345> dyn, a, b;

int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; ++i) cin >> a[i];
    for (int i = 0; i < n; ++i) cin >> b[i];
    dyn[0] = 0;
    LineContainer cht;
    cht.add(-b[0], 0);
    for (int i = 1; i < n; ++i) {
        dyn[i] = cht.query(a[i]);
        cht.add(-b[i], dyn[i]);
    }
    // Original DP O(n^2).
  // for (int i = 1; i < n; i++) {
  //  dyn[i] = INF;
  //  for (int j = 0; j < i; j++)
  //    dyn[i] = min(dyn[i], dyn[j] + a[i] * b[j]);
  // }
  cout << -dyn[n-1] << '\n';
} // hash-cpp-all = 1e5a567f134332193437ca3ce8ce967d
```

Coin.h

Description: Number of ways to make value K with X coins
Time: $\mathcal{O}(NC)$

6 lines

```
for (int i = 0; i < n; ++i)
  for (int j = coins[i]; j <= k; ++j)
    dp[j] += dp[j - coins[i]];

// hash-cpp-all = 20875975dcfbe3880631fce7db3f311e
```

MinCoin.h

Description: minimum number of coins to make K
Time: $\mathcal{O}(kV)$

8 lines

```
int coin(vector<int> &c, int k) {
    vector<int> dp(k+1, INF); dp[0] = 0;
    for (int i = 0; i < c.size(); ++i)
        for (int j = c[i]; j <= k; ++j)
            dp[j] = min(dp[j], 1 + dp[j-c[i]]);
    return dp[k];
}
// hash-cpp-all = 5fe4b1893507d900689285cdb60f4642
```

EditDistance.h

Description: Find the minimum numbers of edits required to convert string s into t. Only insertion, removal and replace operations are allowed.

13 lines

```
vector<vector<int>> dp(MAX_SIZE, vector<int>(MAX_SIZE));
int levDist(const string &s, const string &t) {
    for (int i = 0; i <= s.size(); ++i) dp[i][0] = i;
    for (int i = 0; i <= t.size(); ++i) dp[0][i] = i;
    for (int i = 1; i <= s.size(); ++i) {
        for (int j = 1; j <= t.size(); ++j) {
            dp[i][j] = min(1 + min(dp[i-1][j], dp[i][j-1]),
                dp[i-1][j-1]+(s[i-1] != t[i-1]));
        }
    }
    return dp[s.size()][t.size()];
}

// hash-cpp-all = bc7965e87ec60f5f908915db5495cf76
```

LIS.h

Description: Compute indices for the longest increasing subsequence.
Time: $\mathcal{O}(N \log N)$

17 lines

```
template<class I> vector<int> lis(const vector<I>& S) {
  if (S.empty()) return {};
  vector<int> prev(S.size());
  typedef pair<I, int> p;
  vector<p> res;
  for(int i = 0; i < (int)S.size(); i++) {
    // change 0 -> i for longest non-decreasing subsequence
    auto it = lower_bound(res.begin(), res.end(), p {S[i],
      0});
    if (it == res.end()) res.emplace_back(), it = res.end()
      -1;
    *it = {S[i], i};
    prev[i] = it == res.begin() ? 0 : (it-1)->second;
  }
  int L = res.size(), cur = res.back().second;
  vector<int> ans(L);
  while (L--) ans[L] = cur, cur = prev[cur];
  return ans;
} // hash-cpp-all = 0675f2d50356ddedf96d5db7d84ea048
```

LIS2.h

Description: Compute the longest increasing subsequence.
Time: $\mathcal{O}(N \log N)$

9 lines

```
template<typename T> int lis(const vector<T> &a) {
    vector<T> u;
    for (const T &x : a) {
        auto it = lower_bound(u.begin(), u.end(), x);
        if (it == u.end()) u.push_back(x);
        else *it = x;
    }
    return (int)u.size();
} // hash-cpp-all = 6182d9febfde6942e9eeaee00eec8bed
```

LCS.h

Description: Finds the longest common subsequence.
Memory: $\mathcal{O}(nm)$.
Time: $\mathcal{O}(nm)$ where n and m are the lengths of the sequences.

15 lines

```
template<class T> T lcs(const T &X, const T &Y) {
  int a = X.size(), b = Y.size();
  vector<vvector<int>> dp(a+1, vector<int>(b+1));
  for(int i = 1; i <= a; ++i) for(int j = 1; j <= b; j++)
    dp[i][j] = X[i-1]==Y[j-1] ? dp[i-1][j-1]+1 :
      max(dp[i][j-1],dp[i-1][j]);
  int len = dp[a][b];
  T ans(len, 0);
  while (a && b)
    if (X[a-1] == Y[b-1]) ans[--len] = X[--a], --b;
    else if (dp[a][b-1] > dp[a-1][b]) --b;
    else --a;
  return ans;
}
// hash-cpp-all = 50f3641a242f9568be0038e955aa5cce
```

Knapsack.h

Description: Same 0-1 Knapsack problem, but returns a vector that holds each chosen item.
Time: $\mathcal{O}(nW)$

16 lines

```
vector<int> Knapsack(int limit, vector<int> &v, vector<int>
    &w) {
    vector<vector<int>> dp(v.size()+1);
    dp[0].resize(limit+1);
    for (int i = 0; i < v.size(); ++i) {
        dp[i+1] = dp[i];
        for (int j = 0; j <= limit - w[i]; ++j)
            dp[i+1][w[i]+j] = max(dp[i+1][w[i]+j], dp[i][j]
                + v[i]);
    }
    vector<int> result;
    for (int i = v.size()-1; i >= 0; --i)
        if (dp[i][limit] != dp[i+1][limit]) {
            limit -= w[i];
            result.push_back(i);
        }
    return result;
} // hash-cpp-all = 2b2ab2ea31c6df6578f0e05563c4ea48
```

01Knapsack.h

Description: Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value possible. More efficient space-wise since we work in only one row.
Time: $\mathcal{O}(NW)$

12 lines

```
int knapsack(int limit, vector<int> &v, vector<int> &w) {
    vector<int> dp(limit+1, -1); int n = w.size();
    dp[0] = 0;
    for (int i = 0; i < n; ++i)
        for (int j = limit; j >= w[i]; --j)
            if (dp[j - w[i]] >= 0)
                dp[j] = max(dp[j], dp[j - w[i]] + v[i]);
    int result = 0;
    for (int i = 0; i <= limit; ++i)
        result = max(result, dp[i]);
    return result;
} // hash-cpp-all = e3f23359408b0caf24178c400cbb149b
```

LargeKnapsack.h

Description: Knapsack with definition changed. Support large values because the weight isn't a dimension in our dp anymore.
Time: $\mathcal{O}(vW)$ where v is the sum of values.

9 lines

```
constexpr int limit = (int)1e5+10;
int knapsack(int capacity, vector<lint> &v, vector<lint> &w
    ) {
    vector<lint> dp(limit, 1ll << 60); dp[0] = 0;
    for (int i = 0; i < v.size(); ++i)
        for (int j = limit-v[i]-1; j >= 0; --j)
            dp[j + v[i]] = min(dp[j + v[i]], dp[j] + w[i]);
    for (int i = limit-1; i >= 0; --i)
        if (dp[i] <= capacity) return i;
} // hash-cpp-all = 47c6d1318e8f5bc813fbfeb7824d69f6
```

KnapsackUnbounded.h

Description: Knapsack problem but now take the same item multiple items is allowed.

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

10 lines

```
int knapsack(vector<int> &v, vector<int> &w, int total) {
    vector<int> dp(total+1, -1);
    int result = 0; dp[0] = 0;
    for (int i = 0; i <= total; ++i) for (int j = 0; j < n;
       ↪ ++j)
        if (w[j] <= i && dp[i - w[j]] >= 0)
            dp[i] = max(dp[i], dp[i - w[j]] + v[j]);
    int result = 0;
    for (int i = 0; i <= total; ++i) result = max(result,
       ↪dp[i]);
    return result;
} // hash-cpp-all = 1acfe0783ad35c5ba1d12e2ace7bd8c5
```

KnapsackBounded.h

Description: You are given n types of items, each items has a weight and a quantity. Is possible to fill a knapsack with capacity k using any subset of items?

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

15 lines

```
vector<int> how_many(n+1), dp(k+1);
dp[0] = 1;
for (int i = 1; i <= n; ++i) cin >> how_many[i];
for (int i = 1; i <= n; ++i) {
  for (int j = k-items[i]; j >= 0; --j) {
    if (dp[j]) {
      int x = 1;
      while (x <= how_many[i] &&
        j + x*items[i] <= k && !dp[j + x*items[i]]) {
        dp[j + x*items[i]] = 1;
        ++x;
      }
    }
  }
} // hash-cpp-all = 9bddad842d1780cb3d854abc04574609
```

KnapsackBoundedCosts.h

Description: You are given n types of items, you have $e[i]$ items of i-th type, and each item of i-th type weight $w[i]$ and cost $c[i]$. What is the minimal cost you can get by picking some items weighing at most W in total?

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

<MinQueue.h>

28 lines

```
const int maxn = 1000;
const int maxm = 100000;
const int inf = 0x3f3f3f;

minQueue<int> q[maxm];

array<int, maxm> dp; // the minimum cost dp[i] I need to
   ↪pay in order to fill the knapsack with total weight i
int w[maxn], e[maxn], c[maxn]; // weight, number, cost

int main() {
  int n, m;
  cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> w[i] >> c[i] >> e[i];
  for (int i = 1; i <= m; i++) dp[i] = inf;
  for (int i = 1; i <= n; i++) {
    for (int j = 0; j < w[i]; j++) q[j].clear();
    for (int j = 0; j <= m; j++) {
      minQueue<int> &mq = q[j % w[i]];
      if (mq.size() > e[i]) mq.pop();
      mq.add(c[i]);
      mq.push(dp[j]);
      dp[j] = mq.getMin();
    }
  }
  cout << "Minimum value i can pay putting a total weight "
     ↪ << m << " is " << dp[m] << '\n';
  for (int i = 0; i <= m; i++) cout << dp[i] << " " << i <<
     ↪ '\n';
  cout << "\n";
} // hash-cpp-all = 3ade3c585df1b6af71dce978887ff0fe
```

KnapsackBitset.h

Description: Find first value greater than m that cannot be formed by the sums of numbers from v.

11 lines

```
bitset<int(1e7)> dp, dp1;
int knapsack(vector<int> &items, int n, int m) {
    dp[0] = dp1[0] = true;
    for (int i = 0; i < n; ++i) {
        dp1 <<= items[i];
        dp |= dp1;
        dp1 = dp;
    }
    dp.flip();
    return dp._Find_next(m);
} // hash-cpp-all = 25166e1e3548855c879386d1d513d579
```

TSP.h

Description: Solve the Travelling Salesman Problem.

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

17 lines

```
const int MX = 15;
array<array<int, MX>, 1<<N> dp;
array<array<int, MX>, MX> dist;
int N;
int TSP(int n) {
    dp[0][1] = 0;
    for (int j = 0; j < (1 << n); ++j)
        for (int i = 0; i < n; ++i)
            if (j & (1<<i))
                for (int k = 0; k < n; ++k)
                    if (!(j & (1<<k)))
                        dp[k][j^(1<<k)] = min(dp[k][j^(1<<k
                           ↪)], dp[i][j]+dist[i][k]);
    int ret = (1 << 31); // = INF
    for (int i = 1; i < n; ++i)
        ret = min(ret, dp[i][(1<<n)-1] + dist[i][0]);
    return ret;
} // hash-cpp-all = 9c40a0dd624797eaa12e7898a3960dfd
```

DistinctSubsequences.h

Description: DP eliminates overcounting. Number of different strings that can be generated by removing any number of characters, without changing the order of the remaining.

<ModTemplate.h>

8 lines

```
num tot[30];
num distinct(const string &str) {
    num ans = 1; // tot[i] stands for number of distinct
       ↪strings ending with character 'a'+i
    for(auto &c : str)
      tie(ans, tot[c-'a']) = {2*ans-tot[c-'a'], ans};
    return ans-1;
}
// hash-cpp-all = 6fa2c5ff72e7ea3db04e70d5125fd9ee
```

CircularLCS.h

Description: For strings a, b calculates LCS of a with all rotations of b

Time: $\mathcal{O}(N^2)$

48 lines

```
pair<int,int> dp[2001][4001];
string A,B;

void init() {
  for(int i = 1; i <= A.size(); ++i)
    for(int j = 1; j <= B.size(); ++j) { // naive LCS,
       ↪store where value came from
      pair<int,int>& bes = dp[i][j]; bes = {-1,-1};
      bes = max(bes,{dp[i-1][j].first,0});
      bes = max(bes,{dp[i-1][j-1].first+(A[i-1] == B[j-1])
         ↪,-1});
      bes = mex(bes,{dp[i][j-1].first,-2});
      bes.second *= -1;
    }
}
void adjust(int col) { // remove col'th character of b,
   ↪adjust DP
  int x = 1;
  while (x <= A.size() && dp[x][col].second == 0) x ++;
  if (x > A.size()) return; // no adjustments to dp
  pair<int,int> cur = {x,col}; dp[cur.first][cur.second].
     ↪second = 0;
  while (cur.first <= A.size() && cur.second <= B.size()) {
    // essentially decrease every dp[cur.first][y >= cur.
       ↪second].first by 1
    if (cur.second < B.size() && dp[cur.first][cur.s+1].
       ↪second == 2) {
      cur.second ++;
      dp[cur.first][cur.second].second = 0;
    } else if (cur.first < A.size() && cur.second < B.size
       ↪()
      && dp[cur.first+1][cur.s+1].second == 1) {
      cur.first ++, cur.second ++;
      dp[cur.first][cur.second].second = 0;
    } else cur.first ++;
  }
}
int getAns(pair<int,int> x) {
  int lo = x.second-B.size()/2, ret = 0;
  while (x.first && x.second > lo) {
    if (dp[x.first][x.second].second == 0) x.first --;
    else if (dp[x.first][x.second].second == 1) ret ++, x.
       ↪first --, x.second --;
    else x.second --;
  }
  return ret;
}
int circLCS(str a, str b) {
  A = a, B = b+b; init();
  int ans = 0;
  for(int i = 0; i < B.size(); ++i) {
    ans = max(ans,getAns({A.size(),i+B.size()}));
    adjust(i+1);
  }
  return ans;
} // hash-cpp-all = a573993743cf9eb44b62bfd179cc65a4
```

MaxNonConsecutiveSum.h

Description: Computes the maximum sum of a non consecutive subsequence.

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

8 lines

```
const int MAXN = 100, MAXM = 100;
```

```
array<int, MAXN> A, dp;
int solve(int N) {
    dp[0] = A[0] > 0 ? A[0] : 0;
    if (N > 1) dp[1] = max(dp[0], A[1]);
    for (int i = 2; i < N; i++) dp[i] = max(dp[i - 2] + A[i
       ↪], dp[i - 1]);
    return dp[N - 1];
} // hash-cpp-all = a200bca981e5d321d717bb8cc022d03f
```

MaxSubarraySumSkip.h

Description: Computes the subarray with the maximum sum, removing at most one element from the array.

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

10 lines

```
const int MAXN = 100, MAXM = 100;
array<int, MAXN> A, fw, bw;
int solve(int N) {
    T curMax = fw[0] = A[0], maxSum = A[0];
    for (int i = 1; i < N; i++) { fw[i] = curMax = max(A[i
       ↪], curMax + A[i]); maxSum = max(maxSum, curMax); }
    curMax = maxSum = bw[N - 1] = A[N - 1];
    for (int i = N - 2; i >= 0; i--) { bw[i] = curMax = max
       ↪(A[i], curMax + A[i]); maxSum = max(maxSum, curMax
       ↪); }
    for (int i = 1; i < N - 1; i++) maxSum = max(maxSum, fw
       ↪[i - 1] + bw[i + 1]);
    return maxSum;
} // hash-cpp-all = 5c42f35251ae84c29dc5ff6d7dd4ad13
```

MaxZeroSubmatrix.h

Description: Computes the area of the largest submatrix that contains only 0s

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

18 lines

```
const int MAXN = 100, MAXM = 100;
array<array<int, MAXN>, MAXM> A, H;
int solve(int N, int M) {
    stack<int, vector<int>> s; int ret = 0;
    for (int j = 0; j < M; j++) for (int i = N - 1; i >= 0;
       ↪ i--) H[i][j] = A[i][j] ? 0 : 1 + (i == N - 1 ? 0
       ↪: H[i + 1][j]);
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            int minInd = j;
            while (!s.empty() && H[i][s.top()] >= H[i][j])
               ↪{
                ret = max(ret, (j - s.top()) * (H[i][s.top
                   ↪()]));
                minInd = s.top(); s.pop(); H[i][minInd] = H
                   ↪[i][j];
            }
            s.push(minInd);
        }
        while (!s.empty()) ret = max(ret, (M - s.top()) * H
           ↪[i][s.top()]); s.pop();
    }
    return ret;
} // hash-cpp-all = d7bff28fbc9f249fa8daf1325f932613
```

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).

FastInput.h

Description: Returns an integer. Usage requires your program to pipe in input from file. Can replace calls to *gc*() with

```
struct GC {
    char buf[1 << 16 | 1];
    int bc = 0, be = 0;
    char operator()() {
        if (bc >= be) {
            be = fread(buf, 1, sizeof(buf) - 1, stdin);
            buf[be] = bc = 0;
        }
        return buf[bc++]; // return 0 on EOF
    }
} gc;

void read_int() {}
template <class T, class... S>
inline void read_int(T &a, S &... b) {
    char c, s = 1;
    while (isspace(c = gc()));
    if (c == '-') s = -1, c = gc();
    for (a = c - '0'; isdigit(c = gc()); a = a * 10 + c - '
       ↪0');
    a *= s;
    read_int(b...);
}

void read_float() {}
template <class T, class... S> inline void read_float(T &a,
   ↪ S &... b) {
    int c, s = 1, fp = 0, fpl = 1;
    while (isspace(c = gc()));
    if (c == '-') s = -1, c = gc();
    for (a = c - '0'; isdigit(c = gc()); a = a * 10 + c - '
       ↪0');
    a *= s;
    if (c == '.')
        for (; isdigit(c = gc()); fp = fp * 10 + c - '0',
           ↪fpl *= 10);
    a += (double)fp / fpl;
    read_float(b...);
} // hash-cpp-all = de7573cedad7d78ab4967eb4c26e1fc0
```

Pragmas.h

Description: Be careful.

7 lines

```
#pragma GCC optimize("Ofast") // enable all O3
   ↪optimizations
#pragma GCC target("sse,sse2,sse3,ssse3,sse4,popcnt,abm,mmx
   ↪,avx,avx2,fma")
#pragma GCC optimize("unroll-loops")
#pragma GCC optimize("Ofast","unroll-loops","omit-frame-
   ↪pointer","inline") //Optimization flags
#pragma GCC option("arch=native","tune=native","no-zero-
   ↪upper") //Enable AVX
#pragma GCC target("avx2")  //Enable AVX
// hash-cpp-all = 375d376eab58e7f13af540360ff82cb6
```

BumpAllocator.h

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

9 lines

```
// Either globally or in a single class:
static char buf[450 << 20];
void* operator new(size_t s) {
  static size_t i = sizeof buf;
  assert(s < i);
  return (void*)&buf[i -= s];
}
void operator delete(void*) {}
// hash-cpp-all = 745db225903de8f3cdfa051660956100
```

SmallPtr.h

Description: A 32-bit pointer that points into BumpAllocator memory.

"BumpAllocator.h" 10 lines

```
template<class T> struct ptr {
  unsigned ind;
  ptr(T* p = 0) : ind(p ? unsigned((char*)p - buf) : 0) {
```

```
    assert(ind < sizeof buf);
  }
  T& operator*() const { return *(T*)(buf + ind); }
  T* operator->() const { return &**this; }
  T& operator[](int a) const { return (&**this)[a]; }
  explicit operator bool() const { return ind; }
}; // hash-cpp-all = 2dd6c9773f202bd47422e255099f4829
```

BumpAllocatorSTL.h

Description: BumpAllocator for STL containers.
Usage: `vector<vector<int, small<int>>> ed(N);`

14 lines

```
char buf[450 << 20] alignas(16);
size_t buf_ind = sizeof buf;

template<class T> struct small {
  typedef T value_type;
  small() {}
  template<class U> small(const U&) {}
  T* allocate(size_t n) {
    buf_ind -= n * sizeof(T);
    buf_ind &= 0 - alignof(T);
    return (T*)(buf + buf_ind);
  }
  void deallocate(T*, size_t) {}
}; // hash-cpp-all = bb66d4225a1941b85228ee92b9779d4b
```

ycombinator.h

Description: Efficient recursive lambda without overhead of std::function.
Usage: `std::y_combinator([] (auto self, int v) -> void {})(0, 0);`

```
auto check = std::y_combinator([] (auto self, int v) ->
void {});
```

16 lines

```
namespace std {
template<class F>
class y_combinator_result {
  F fun_;
public:
  template<class T>
  explicit y_combinator_result(T &&fun): fun_(std::forward<
    ↪T>(fun)) {}
  template<class ...Args>
  decltype(auto) operator()(Args &&...args) {
    return fun_(std::ref(*this), std::forward<Args>(args)
      ↪...);
  }
};
template<class F>
decltype(auto) y_combinator(F &&fun) {
  return y_combinator_result<std::decay_t<F>>(std::forward<
    ↪F>(fun));
}} // hash-cpp-all = 7ab7f45dc4a106443f0e365e9c866ce1
```

Hashmap.h

Description: Faster/better hash maps, taken from CF

20 lines

```
#include<bits/extc++.h>
struct splitmix64_hash {
    static uint64_t splitmix64(uint64_t x) {
        x += 0x9e3779b97f4a7c15;
        x = (x^(x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x^(x >> 27)) * 0x94d049bb133111eb;
        return x^(x >> 31);
    }
    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM = std::chrono::
            ↪steady_clock::now().time_since_epoch().count()
            ↪;
        return splitmix64(x + FIXED_RANDOM);
    }
};

template <typename K, typename V, typename Hash =
    ↪splitmix64_hash>
using hash_map = __gnu_pbds::gp_hash_table<K, V, Hash>;

template <typename K, typename Hash = splitmix64_hash>
using hash_set = hash_map<K, __gnu_pbds::null_type, Hash>;
// hash-cpp-all = 09a72f779dabd74c7e9f2ce85f2aec1e
```

PQueue.h

Description: Efficient priority queue implementation. Initialize with highest possible value. Can obviously be extended to minheap/max-heap.

15 lines

```
template<typename T> struct PQ {
    int sz;
    vector<T> q;
    T offset = 0;
    PQ(int n) : sz(n+1), q(2*n, -n) {}
    T top() { return -q[0]+offset; }
    void push(T x) {
        q[sz++] = -(x-offset);
        push_heap(q.begin(),q.begin()+sz);
    }
    void shift(T x) { offset+=x; }
    void pop() {
        pop_heap(q.begin(),q.begin()+sz); --sz;
    }
}; // hash-cpp-all = ffd97e5129dcbb5cc4a7811cd07f7e37
```

OwnFunctions.h

18 lines

```
template <typename T>
T mabs(T v) {
  return v < 0 ? -v : v;
}

template <typename T>
T mceil(T v) {
  T x = ceil((long double)v) - 1.0;
  while (x < v) x += 1.0;
  return x;
}

template <typename T>
T mfloor(T v) {
  T x = floor((long double)v) + 1.0;
  while (x > v) x -= 1.0;
  return x;
} // hash-cpp-all = 786225192828898cbd2b5b423b2ec67b
```

FastMod.h

Description: Compute $a\%b$ about 4 times faster than usual, where b is constant but not known at compile time. Fails for $b = 1$.

10 lines

```
typedef unsigned long long ull;
typedef __uint128_t L;
struct FastMod {
  ull b, m;
  FastMod(ull b) : b(b), m(ull((L(1) << 64) / b)) {}
  ull reduce(ull a) {
    ull q = (ull)((L(m) * a) >> 64), r = a - q * b;
    return r >= b ? r - b : r;
  }
}; // hash-cpp-all = c977c57b95c3db104536d47fe92e5016
```

10.6 Bit Twiddling Hack

Hacks.h

51 lines

```
// Returns one plus the index of the least significant 1-
   ↪bit of x, or if x is zero, returns zero.
__builtin_ffs(x)

// Returns the number of leading 0-bits in x, starting at
   ↪the most significant bit position. If x is 0, the
   ↪result is undefined.
__builtin_clz(x)

// Returns the number of trailing 0-bits in x, starting at
   ↪the least significant bit position. If x is 0, the
   ↪result is undefined.
__builtin_ctz(x)

// Returns the number of 1-bits in x.
__builtin_popcount(x)

// For long long versions append ll (e.g.
   ↪__builtin_popcountll)

// Least significant bit in x.
x & -x

// Iterate on non-empty submasks of a bitmask.
for (int submask = mask; submask > 0; submask = (mask & (
   ↪submask - 1)))

// Iterate on non-zero bits of a bitset.
for (int j = btset._Find_next(0); j < MAXV; j = btset.
   ↪_Find_next(j))

int __builtin_clz(int x); // number of leading zero
int __builtin_ctz(int x); // number of trailing zero
int __builtin_clzll(lint x); // number of leading zero
int __builtin_ctzll(lint x); // number of trailing zero
int __builtin_popcount(int x); // number of 1-bits in x
int __builtin_popcountll(lint x); // number of 1-bits in x

// compute next perm. i.e. 00111, 01011, 01101, 10011, ...
lint next_perm(lint v) {
    lint t = v | (v-1);
    return (t + 1) | (((~t & -~t) - 1) >> (__builtin_ctz(v)
      ↪ + 1));
}

template<typename F>  // All subsets of size k of {0..N-1}
void iterate_k_subset(ll N, ll k, F f){
  ll mask = (1ll << k) - 1;
  while (!(mask & 1ll<<N)) { f(mask);
    ll t = mask | (mask-1);
    mask = (t+1) | (((~t & -~t) - 1) >> (__builtin_ctzll(
      ↪mask)+1));
  }
}
template<typename F>  // All subsets of set
void iterate_mask_subset(ll set, F f){ ll mask = set;
  do  f(mask), mask = (mask-1) & set;
  while (mask != set);
} // hash-cpp-all = 59c333b5627ba2e7fea7f2a5da6d2881
```

Bitset.h

Description: Some bitset functions

17 lines

```
int main() {
    bitset<100> bt;
    cin >> bt;
    cout << bt[0] << "\n";
    cout << bt.count() << "\n"; // number of bits set
    cout << (~bt).none() << "\n"; // return true if has no
       ↪bits set
    cout << (~bt).any() << "\n"; // return true if has any
       ↪bit set
    cout << (~bt).all() << "\n"; // retun true if has all
       ↪bits set
    cout << bt._Find_first() << "\n"; // return first set
       ↪bit
    cout << bt._Find_next(10) << "\n";// returns first set
       ↪bit after index i
    cout << bt.flip() << '\n'; // flip the bitset
    cout << bt.test(3) << '\n'; // test if the ith bit of
       ↪bt is set
    cout << bt.reset(3) << '\n'; // reset the ith bit
    cout << bt.set() << '\n'; // turn all bits on
    cout << bt.set(4, 1) << '\n'; // set the 4th bit to
       ↪value 1
    cout << bt << "\n";
} // hash-cpp-all = b9f55a20e426e6ea81485e438f9f3325
```

10.7 Random Numbers

RandomNumbers.h

Description: An example on the usage of generator and distribution.

5 lines

```
mt19937_64 mt (time (0));
uniform_int_distribution <int> uid (1, 100);
uniform_real_distribution <double> urd (1, 100);
cout << uid (mt) << " " << urd (mt) << "\n";
// hash-cpp-all = 63c591021510cd5bc0d42c6bb21c7c51
```

10.8 Other languages

Python3.py

50 lines

```
/**
 * Author: BenQ
 * Description: python3 (not pypy3) demo, solves
  * CF Good Bye 2018 Factorisation Collaboration
 * Source: own
 * Verification:
  * https://codeforces.com/contest/1091/problem/G
  * https://open.kattis.com/problems/catalansquare
 */

from math import *
import sys
import random

def nextInt():
  return int(input())
def nextStrs():
  return input().split()
def nextInts():
  return list(map(int,nextStrs()))

n = nextInt()
v = [n]
def process(x):
  global v
  x = abs(x)
  V = []
  for t in v: # print(type(t)) -> <class 'int'>
    g = gcd(t,x)
    if g != 1:
      V.append(g)
    if g != t:
      V.append(t//g)
  v = V
for i in range(50):
  x = random.randint(0,n-1)
  if gcd(x,n) != 1:
    process(x)
  else:
    sx = x*x%n # assert(gcd(sx,n) == 1)
    print(f"sqrt {sx}") # print value of var
    sys.stdout.flush()
    X = nextInt()
    process(x+X)
    process(x-X)
print(f'! {len(v)}',end='')
for i in v:
  print(f' {i}',end='')
print()
sys.stdout.flush()
```

Main.java

Description: Basic template/info for Java

15 lines

```
import java.util.*;
import java.math.*;
import java.io.*;

public class Main {
  public static void main(String[] args) throws Exception {
    BufferedReader br = new BufferedReader(new
       ↪InputStreamReader(System.in));
    PrintStream out = System.out;
    StringTokenizer st = new StringTokenizer(br.readLine())
       ↪;
    assert st.hasMoreTokens(); // enable with java -ea main
    out.println("v=" + Integer.parseInt(st.nextToken()));
    ArrayList<Integer> a = new ArrayList<>();
    a.add(1234); a.get(0); a.remove(a.size()-1); a.clear();
  }
}
```

MiscJava.java

Description: Basic template/info for Java

47 lines

```
import java.math.BigInteger;
import java.util.*;

public class prob4 {
  void run() {
    Scanner scanner = new Scanner(System.in);
    while (scanner.hasNextBigInteger()) {
      BigInteger n = scanner.nextBigInteger();
      int k = scanner.nextInt();
      if (k == 0) {
        for (int p = 2; p <= 100000; p++) {
          BigInteger bp = BigInteger.valueOf(p);
          if (n.mod(bp).equals(BigInteger.ZERO)) {
            System.out.println(bp.toString() + " * " + n.
               ↪divide(bp).toString());
            break;
          }
        }
      } else {
        BigInteger ndivk = n.divide(BigInteger.valueOf(k));
        BigInteger sqndivk = sqrt(ndivk);
        BigInteger left = sqndivk.subtract(BigInteger.
           ↪valueOf(100000)).max(BigInteger.valueOf(2));
        BigInteger right = sqndivk.add(BigInteger.valueOf
           ↪(100000));
        for (BigInteger p = left; p.compareTo(right) != 1;
           ↪p = p.add(BigInteger.ONE)) {
          if (n.mod(p).equals(BigInteger.ZERO)) {
            BigInteger q = n.divide(p);
            System.out.println(p.toString() + " * " + q.
               ↪toString());
            break;
          }
        }
      }
    }
  }
  BigInteger sqrt(BigInteger n) {
    BigInteger left = BigInteger.ZERO;
    BigInteger right = n;
    while (left.compareTo(right) != 1) {
      BigInteger mid = left.add(right).divide(BigInteger.
         ↪valueOf(2));
      int s = n.compareTo(mid.multiply(mid));
      if (s == 0) return mid;
      if (s > 0) left = mid.add(BigInteger.ONE); else right
         ↪ = mid.subtract(BigInteger.ONE);
    }
    return right;
  }
  public static void main(String[] args) {
    (new prob4()).run();
  }
}
```

10.8.1 BigInteger

BigInteger To convert to a `BigInteger`, use `BigInteger.valueOf (int)` or `BigInteger (String, radix)`.

To convert from a `BigInteger`, use `.intValue ()`, `.longValue ()`, `.toString (radix)`.

Common unary operations include `.abs ()`, `.negate ()`, `.not ()`.

Common binary operations include `.max`, `.min`, `.add`, `.subtract`, `.multiply`, `.divide`, `.remainder`, `.gcd`, `.modInverse`, `.and`, `.or`, `.xor`, `.shiftLeft (int)`, `.shiftRight (int)`, `.pow (int)`, `.compareTo`.

Divide and remainder: `Biginteger[] .divideAndRemainder (Biginteger val)`.

Power module: `.modPow (BigInteger exponent, module)`.

Primality check: `.isProbablePrime (int certainty)`.