Competições de Algoritmos e Programação UFRJ

Federal University of Rio de Janeiro

UFRJ - Time Feliz ^_^

Chris Ciafrino, Gustavo Miguel 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
```

Makefile

25 lines

```makefile
CXX = g++
CXXFLAGS = -O2 -std=gnu++14 -Wall -Wextra -Wno-unused-result -pedantic -Wshadow -Wformat=2 -Wfloat-equal -Wconversion -Wlogical-op -Wshift-overflow=2 -Wduplicated-cond -Wcast-qual -Wcast-align
# pause:#pragma GCC diagnostic {ignored|warning} "-Wshadow"
DEBUGFLAGS = -D_GLIBCXX_DEBUG -D_GLIBCXX_DEBUG_PEDANTIC -fsanitize=address -fsanitize=undefined -fno-sanitize-recover=all -fstack-protector -D_FORTIFY_SOURCE=2
CXXFLAGS += $(DEBUGFLAGS) # flags with speed penalty
TARGET := $(notdir $(CURDIR))
EXECUTE := ./$(TARGET)
CASES := $(sort $(basename $(wildcard *.in)))
TESTS := $(sort $(basename $(wildcard *.out)))
all: $(TARGET)
clean:
	-rm -rf $(TARGET) *.res
%: %.cpp
	$(LINK.cpp) $< $(LOADLIBES) $(LDLIBS) -o $@
run: $(TARGET)
	time $(EXECUTE)
%.res: $(TARGET) %.in
	time $(EXECUTE) < $*.in > $*.res
%.out: %
test_%: %.res %.out
	diff $*.res $*.out
runs: $(patsubst %,%.res,$(CASES))
test: $(patsubst %,test_%,$(TESTS))
.PHONY: all clean run test test_% runs
.PRECIOUS: %.res
```

vimrc

29 lines

```vim
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

52 lines

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

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

Runtime error:
Have you tested all corner cases locally?
Any uninitialized variables?
Are you reading or writing outside the range of any vector?
Any assertions that might fail?
Any possible division by 0? (mod 0 for example)
Any possible infinite recursion?
Invalidated pointers or iterators?
Are you using too much memory?
Debug with resubmits (e.g. remapped signals, see Various).

Time limit exceeded:
Do you have any possible infinite loops?
What is the complexity of your algorithm?
Are you copying a lot of unnecessary data? (References)
How big is the input and output? (consider scanf)
Avoid vector, map. (use arrays/unordered_map)
What do your team mates think about your algorithm?

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

Mathematics (2)

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

2.4 Geometry

2.4.1 Triangles

Side lengths: a, b, c

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

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

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

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

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

Length of bisector (divides angles in two):

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

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

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

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

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)

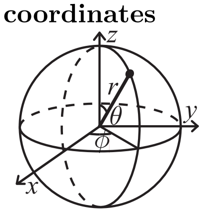
2.4.2 Quadrilaterals

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

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

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

2.4.3 Spherical coordinates



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

2.4.4 Centroid of a polygon

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

2.5 Derivatives/Integrals

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

Integration by parts:

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

2.6 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 $\text{Bin}(n, p)$, $n = 1, 2, \ldots$, $0 \le p \le 1$.

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

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

$\text{Bin}(n, p)$ 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 $\text{Fs}(p)$, $0 \le p \le 1$.

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

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

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 $\text{Exp}(\lambda)$, $\lambda > 0$.

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

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

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

Data Structures (3)

HashMap.h

Description: Hash map with the same API as unordered_map, but ~3x faster. 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 {
  const uint64_t C = ll(2e18 * M_PI) + 71; // large odd
    ↪number
  ll operator()(ll x) const { return __builtin_bswap64(x*C)
    ↪; }
};
__gnu_pbds::gp_hash_table<ll,int,chash> h
  ↪({},{},{},{},{1<<16}); // hash-cpp-all = 1443
  ↪bc6c3b4062862ebb83553bc131a0
```

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

20 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]);
    }
    int unite(int a, int b) {
        a = find(a);
        b = find(b);
        if (a == b) return a;
        if (rank[a] > rank[b]) swap(a, b);
        parent[a] = b;
        if (rank[a] == rank[b]) ++rank[b];
        return b;
    }
}; // hash-cpp-all = b237fabe1fcbfbf7f52205b112487f5e
```

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: Time and space efficient Segment Tree. Point update and range query.
Time: $\mathcal{O}(\log N)$

48 lines

```
template<class T>
struct segtree_t {
    int size;
    vector<T> t;
    segtree_t(int N) : size(N), t(2 * N) {}
    segtree_t(const vector<T> &other) :
            size(other.size()),
            t(2 * other.size()) {
        copy(other.begin(), other.end(), t.begin() + size);
        for (int i = size; i-- > 1;)
            t[i] = combine(t[2 * i], t[2 * i + 1]);
    }
    T get(int p) {
        return t[p + size];
    }
    void update(int p, T value) {
        p += size;
        t[p] = value;
        while (p > 1) {
            p /= 2;
            t[p] = combine(t[2 * p], t[2 * p + 1]);
        }
    }
    T query(int l, int r) {
        l += size; r += size;
        T left = init();
        T right = init();
        while (l < r) {
            if (l & 1) {
                left = combine(left, t[l]);
                l++;
            }
            if (r & 1) {
                r--;
                right = combine(t[r], right);
            }
            l /= 2; r /= 2;
        }
        return combine(left, right);
    }
private:
    T combine(T left, T right) {
        return (left + right);
    }
    T init() {
        return T();
    }
}; // hash-cpp-all = 87858a9bfd15027e584eb785bc0b0e29
```

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.

50 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(2*v,l, m), build(2*v+1, m
          ↪+1, r));
    }
```

```
    void propagate(int v, int l, int r) {
        if (!lazy[v]) return;
        int m = l + (r - l)/2;
        tree[2*v] += lazy[v] * (m - l + 1);
        tree[2*v+1] += lazy[v] * (r - (m+1) + 1);
        lazy[2*v] += lazy[v];
        lazy[2*v+1] += 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, 2*v,l, m), query(a, b, 2*v+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, 2*v, l, m),
            update(a, b, delta, 2*v+1, m+1, r));
    }
};
// hash-cpp-all = 3af20e17da7a1a0e4e2d3ac3d108286a
```

DynamicSegTree.h

Description: Dynamic Segment Tree with lazy propagation.
Usage: vector<int> a;
node *segtree = build(0, n, a);

79 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;
  }
  update(v->l, lx, rx, delta);
  update(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 1e16;
  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, vector<int>& a) {
  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 = a[lx];
  }
  else {
    v->l = build(lx, (lx + rx)/2, a);
    v->r = build((lx + rx)/2 + 1, rx, a);
    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 = 4c6bdc9d86fd353743d4f29c5b774da5
```

MergeSortTree.h

37 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]; });
        make_tree(1, 0, v.size()-1);
    }
    void make_tree(int id, int left, int right) {
        if (left == right)
            tree[id].push_back(id[left]);
        else {
            int mid = (left + right)/2;
            make_tree(2*id, left, mid);
            make_tree(2*id+1, mid+1, right);
            tree[id] = vector<int>(right - left + 1);
            merge(tree[2*i].begin(), tree[2*i].end(),
                tree[2*id+1].begin(), tree[2*id+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)/2;
        int lcount = how_many(2*id, a, b);
        if (lcount >= x) return query(2*id, left, mid, a, b
            ↪, x);
        else return query(2*id+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 = 01e250d36257c202f6f6713e170d49d3
```

SqrtDecomposition.h

Description: Provides two operations on an array A (the same as a Fenwick tree): 1) Add x to A[i]. Runs in O(1). 2) Query the sum of A[left] through A[right - 1].
Time: $\mathcal{O}\left(\sqrt{n}\right)$

51 lines

```
template<typename T>
struct sqrt_sums {
    int n, bucket_size, n_buckets;
    vector<T> values, bucket_sums;
    sqrt_sums(int _n = 0) : n(_n), bucket_size(1.2*sqrt(n)
        ↪+1),
        n_buckets((n+bucket_size-1)/bucket_size), values(n)
            ↪,
        bucket_sums(n_buckets) {}
    sqrt_sums(const vector<T> &other) : n(other.size()),
        bucket_size(1.2*sqrt(n)+1), n_buckets((n+
            ↪bucket_size-1)/bucket_size),
        values(other), bucket_sums(n_buckets) {
            for (int b = 0; b < n_buckets; b++) {
                int left = get_bucket_left(b);
                int right = get_bucket_right(b);
                for (int i = left; i < right; i++)
                    bucket_sums[b] += values[i];
            }
    }
    int which_bucket(int index) const { return index < n ?
        ↪index / bucket_size : n_buckets; }
    int get_bucket_left(int b) const { return bucket_size *
        ↪ b; }
```

```
    int get_bucket_right(int b) const { return min(
        bucket_size * (b + 1), n);}
    void update(int index, T change) {
        assert(0 <= index && index < n);
        values[index] += change;
        bucket_sums[which_bucket(index)] += change;
    }
    T query(int left, int right) const {
        assert(0 <= left && left <= right && right <= n);
        T sum = 0;
        int left_b = which_bucket(left), right_b =
            which_bucket(right);
        int bucket_left = get_bucket_left(left_b);
        int bucket_right = get_bucket_right(left_b);
        if (left - bucket_left < bucket_right - left)
            while (left > bucket_left)
                sum -= values[--left];
        else while (left < bucket_right)
                sum += values[left++];
        bucket_left = get_bucket_left(right_b);
        bucket_right = get_bucket_right(right_b);
        if (right - bucket_left < bucket_right - right)
            while (right > bucket_left)
                sum += values[--right];
        else
            while (right < bucket_right)
                sum -= values[right++];
        left_b = which_bucket(left);
        right_b = which_bucket(right);
        for (int b = left_b; b < right_b; b++)
            sum += bucket_sums[b];
        return sum;
    }
}; // hash-cpp-all = 6bcb506d32a5c27c84f5396e387b950c
```

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

37 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 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, obviously doesn't work with sum or similar queries.
Usage: `RMQ<int> rmq(values);`
`rmq.query(inclusive, inclusive);`
Time: $\mathcal{O}(|V|\log|V| + Q)$

33 lines

```
// change cmp for max query or similar
template<typename T, typename Cmp=less<pair<T, int>>>
struct RMQ {
  Cmp cmp;
  vector<vector<pair<T, int>>> table;
  RMQ(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); // Change if max
        //table[l][i].first = (table[l-1][i].first +
            table[l-1][i+(1<<(l-1))].first); //
            example of sum
      }

    }
  }
  pair<T, int> query(int a, int b) { // min query
    int l = __lg(b-a+1);
    return min(table[l][a], table[l][b-(1<<l)+1], cmp);
  }
  int sum_query(int a, int b) {
      int l = b-a+1, ret = 0;
      for (int i = (int)table.size(); i >= 0; --i)
          if ((1 << i) <= l) {
              ret += table[i][a].first; a += (1 << i);
              l = b - a + 1;
          }
      return ret;
  }
}; // hash-cpp-all = a4b96ac4510d8a21d788aadcb7621b46
```

FenwickTree.cpp

Description: Classic FT with linear initialization. All queries are [a, b). get(pos) function returns the element at index pos in O(1) amortized. lowerbound(sum) returns the largest i in [0, n] st query(i) <= sum. Returns -1 if no such i exists (sum < 0). Can be used as an ordered set on indices in [0, n) by using the tree as a 0/1 array: update(index, +1) is equivalent to insert(index); be careful not to re-insert. get(index) provides whether index is present or not. query(index) provides the count of elements < index. lowerbound(k) finds the k-th smallest element (0-indexed).
Time: Both operations are $\mathcal{O}(\log N)$.

47 lines

```
template<typename T> struct FT {
    T tree_sum;
    const int n;
    vector<T> tree;
    FT(int n) : tree(n) {}
    FT(vector<T> &og) : n(og.size()+1), tree(n+1), tree_sum
        (0) {
        for (int i = 1; i <= n; ++i) {
            tree_sum += og[i-1];
            tree[i] = og[i-1];
            for (int k = (i&-i) >> 1; k > 0; k >>= 1)
                tree[i] += tree[i-k];
        }
    }
    void update(int idx, const T delta) {
        tree_sum += delta;
        for (int i = idx+1; i <= tree.size(); i += i&-i)
            tree[i] += delta;
    }
    T query(int idx){
        T ret = 0;
        for (int i = idx; i > 0; i -= i&-i)
            ret += tree[i];
        return ret;
    }
    T query_suffix(int idx) { return tree_sum - query(idx);
        }
    T query(int a, int b) { return query(b) - query(a); }
    T get(int pos) {
        int above = pos + 1;
        T sum = tree[above];
        above -= above & -above;
        while (pos != above) {
            sum -= tree[pos];
            above -= above&-above;
        }
        return sum;
    }
    int lower_bound(T sum) {
        if (sum < 0) return -1;
        int prefix = 0;
        for (int k = 31 - __builtin_clz(n); k >= 0; k--)
            if (prefix + (1 << k) <= n && tree[prefix + (1
                << k)] <= sum) {
                prefix += 1 << k;
                sum -= tree[prefix];
            }
        return prefix;
    }
}; // hash-cpp-all = 56bf341c4f0aee776ab89f38ec32abe5
```

LazyFenwickTree.h

Description: Fenwick Tree with Lazy Propagation

27 lines

```
struct bit_t { // hash-cpp-1
  int n;
  vector<vector<int>> tree(2);
```

```
  bit_t (int n): n(n+10) {
    tree[0].assign(n, 0);
    tree[1].assign(n, 0);
  } // hash-cpp-1 = 0f9e719127708bbe01730d68a10ecd83
  void update(int bit, int idx, int delta) { // hash-cpp-2
    for (++idx; idx <= n; idx += idx&-idx)
      tree[bit][idx] += delta;
  }
  void update(int lx, int rx, int delta) {
    update(0, lx, delta);
    update(0, rx+1, -delta);
    update(1, lx, (l-1) * delta);
    update(1, rx+1, -rx * delta);
  } // hash-cpp-2 = 6250fe8cf18b3f5d9a24cbca8fa4f96a
  int query(int bit, int idx) { // hash-cpp-3
    int ret = 0;
    for (++idx; idx > 0; idx -= idx&-idx)
      ret += tree[bit][idx];
    return ret;
  }
  int query(int idx) {
    return query(0, idx) * idx - query(1, idx);
  } // hash-cpp-3 = 533d8960bcb2576e15997cb4dd75f429
};
```

FenwickTree2d.h

Description: Computes sums a[i,j] for all i<I, j<J, and increases single elements a[i,j]. Requires that the elements to be updated are known in advance (call fakeUpdate() before init()).
Time: $\mathcal{O}(\log^2 N)$. (Use persistent segment trees for $\mathcal{O}(\log N)$.)

"FenwickTree.h" 22 lines

```
struct FT2 {
  vector<vi> ys; vector<FT> ft;
  FT2(int limx) : ys(limx) {}
  void fakeUpdate(int x, int y) {
    for (; x < sz(ys); x |= x + 1) ys[x].push_back(y);
  }
  void init() {
    for(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, ll dif) {
    for (; x < ys.size(); x |= x + 1)
      ft[x].update(ind(x, y), dif);
  }
  ll query(int x, int y) {
    ll sum = 0;
    for (; x; x &= x - 1)
      sum += ft[x-1].query(ind(x-1, y));
    return sum;
  }
}; // hash-cpp-all = d69016552f1286eca884f46081b7feb6
```

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

32 lines

```
bool Q;
struct Line {
  mutable lint k, m, p;
  bool operator<(const Line& o) const {
    return Q ? p < o.p : k < o.k;
  }
};

struct LineContainer : multiset<Line> { // hash-cpp-1
  // (for doubles, use inf = 1/.0, div(a,b) = a/b)
  const lint inf = lintONG_MAX;
  lint div(lint a, lint b) { // floored division
    return a / b - ((a ^ b) < 0 && a % b); } // hash-cpp-1
      ↪= c86e64c1b59fe34a6f603b19420a916b
  bool isect(iterator x, iterator y) { // hash-cpp-2
    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;
  } // hash-cpp-2 = ea780949e14e74de80f1cf68e8e866b4
  void add(lint k, lint m) { // hash-cpp-3
    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));
  } // hash-cpp-3 = 0b051363d3992bb6e6b9d193f10494cb
  lint query(lint x) { // hash-cpp-4
    assert(!empty());
    Q = 1; auto l = *lower_bound({0,0,x}); Q = 0;
    return l.k * x + l.m;
  } // hash-cpp-4 = e513e009847713fe7c68e103adbbbc5b
};
```

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;
  }
}; // hash-cpp-all = ac78976eee0ad16cad5450c4dfecd3a0
```

Treap3.h

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

69 lines

```
const int N = ; typedef int num;
num X[N]; int en = 1, Y[N], sz[N], L[N], R[N];

void calc(int u) {
    sz[u] = sz[L[u]] + 1 + sz[R[u]];
    // code here, no recursion
}

void unlaze(int u) {
    if (!u) return;
    // code here, no recursion
}

void split_val(int u, num x, int &lx, int &rx) {
    unlaze(u); if (!u) return (void)(lx = rx = 0);
    if (X[u] <= x) {
        split_val(R[u], x, lx, rx);
        R[u] = lx;
        lx = u;
    }
    else {
        split_val(L[u], x, lx, rx);
        L[u] = rx;
        rx = u;
    }
    calc(u);
}
void split_sz(int u, int s, int &lx, int &rx) {
    unlaze(u); if (!u) return (void)(lx = rx = 0);
    if (sz[L[u]] < s) {
        split_sz(R[u], s-sz[L[u]]-1, lx, rx);
        R[u] = lx;
        lx = u;
    }
    else {
        split_sz(L[u], s, lx, rx);
        L[u] = rx;
```

```
        rx = u;
    }
    calc(u);
}

int merge(int lx, int rx) {
    unlaze(lx); unlaze(rx); if (!lx || !rx) return lx+rx;
    int u;
    if (Y[lx] > Y[rx]) {
        R[lx] = merge(R[lx], rx);
        u = lx;
    }
    else {
        L[rx] = merge(lx, L[rx]);
        u = rx;
    }
    calc(u);
    return u;
}

void build(int n = N-1) {
    for (int i = en = 1; i <= n; ++i) {
        Y[i] = i;
        sz[i] = 1;
        L[i] = R[i] = 0;
    }
    random_shuffle(Y + 1, Y + n + 1);
}

// hash-cpp-all = 3584d09d8794275b37f50c27be4d14e6
```

LCT.cpp

Description: Link-Cut Tree. Supports BBST = like augmentation, can fully replace Heavylight Decomposition.

106 lines

```
struct T {
  bool rr;
  T *son[2], *pf, *fa;
} f1[N], *ff = f1, *f[N], *null;

void downdate(T *x) {
  if (x -> rr) {
    x -> son[0] -> rr = !x -> son[0] -> rr;
    x -> son[1] -> rr = !x -> son[1] -> rr;
    swap(x -> son[0], x -> son[1]);
    x -> rr = false;
  }
  // add stuff
}

void update(T *x) {
  // add stuff
}

void rotate(T *x, bool t) { // hash-cpp-1
  T *y = x -> fa, *z = y -> fa;
  if (z != null)  z -> son[z -> son[1] == y] = x;
  x -> fa = z;
  y -> son[t] = x -> son[!t];
  x -> son[!t] -> fa = y;
  x -> son[!t] = y;
  y -> fa = x;
  update(y);
} // hash-cpp-1 = 28958e1067126a5892dcaa67307d2f1d

void xiao(T *x) {
  if (x -> fa != null)  xiao(x -> fa), x -> pf = x -> fa ->
    pf;
  downdate(x);
}

void splay(T *x) { // hash-cpp-2
  xiao(x);
  T *y, *z;
  while (x -> fa != null) {
    y = x -> fa; z = y -> fa;
    bool t1 = (y -> son[1] == x), t2 = (z -> son[1] == y);
    if (z != null) {
      if (t1 == t2) rotate(y, t2), rotate(x, t1);
      else  rotate(x, t1), rotate(x, t2);
    }else rotate(x, t1);
  }
  update(x);
} // hash-cpp-2 = 0bc1a3b77275f92cebc947211444fdb7

void access(T *x) { // hash-cpp-3
  splay(x);
  x -> son[1] -> pf = x;
  x -> son[1] -> fa = null;
  x -> son[1] = null;
  update(x);
  while (x -> pf != null) {
    splay(x -> pf);
    x -> pf -> son[1] -> pf = x -> pf;
    x -> pf -> son[1] -> fa = null;
    x -> pf -> son[1] = x;
    x -> fa = x -> pf;
    splay(x);
  }
  x -> rr = true;
} // hash-cpp-3 = db89159f01a2099d67e93163c3bfa384

bool Cut(T *x, T *y) { // hash-cpp-4
  access(x);
  access(y);
  downdate(y);
  downdate(x);
  if (y -> son[1] != x || x -> son[0] != null)
    return false;
  y -> son[1] = null;
  x -> fa = x -> pf = null;
  update(x);
  update(y);
  return true;
} // hash-cpp-4 = 42850d63565f84698378e8c2c23df1fe

bool Connected(T *x, T *y) {
  access(x);
  access(y);
  return x == y || x -> fa != null;
}

bool Link(T *x, T *y) {
  if (Connected(x, y))
    return false;
  access(x);
  access(y);
  x -> pf = y;
  return true;
}

int main() {
  read(n); read(m); read(q);
  null = new T; null -> son[0] = null -> son[1] = null ->
    fa = null -> pf = null;
  for (int i = 1; i <= n; i++) {
    f[i] = ++ff;
    f[i] -> son[0] = f[i] -> son[1] = f[i] -> fa = f[i] ->
      pf = null;
    f[i] -> rr = false;
  }
  // init null and f[i]
}
```

SplayTree.h

99 lines

```
//const int N = ;
//typedef int num;
int en = 1;
int p[N], sz[N];
int C[N][2]; // {left, right} children
num X[N];

// atualize os valores associados aos nos que podem ser
  calculados a partir dos filhos
void calc(int u) {
  sz[u] = sz[C[u][0]] + 1 + sz[C[u][1]];
}

// Puxa o filho dir de u para ficar em sua posicao e o
  retorna
int rotate(int u, int dir) {
  int v = C[u][dir];
  C[u][dir] = C[v][!dir];
  if(C[u][dir]) p[C[u][dir]] = u;
  C[v][!dir] = u;
  p[v] = p[u];
  if(p[v]) C[p[v]][C[p[v]][1] == u] = v;
  p[u] = v;
  calc(u);
  calc(v);
  return v;
}
// Traz o no u a raiz
void splay(int u) {
  while(p[u]) {
    int v = p[u], w = p[p[u]];
    int du = C[v][1] == u;
    if(!w)
      rotate(v, du);
    else {
      int dv = (C[w][1] == v);
      if(du == dv) {
        rotate(w, dv);
        rotate(v, du);
      } else {
        rotate(v, du);
        rotate(w, dv);
      }
    }
  }
}
// retorna um no com valor x, ou outro no se n foi
  encontrado (n eh floor nem ceiling)
int find_val(int u, num x) {
  int v = u;
  while(u && X[u] != x) {
    v = u;
    if(x < X[u]) u = C[u][0];
    else u = C[u][1];
```

```
  }
  if(!u) u = v;
  splay(u);
  return u;
}
// retorna o s-esimo no (0-indexed)
int find_sz(int u, int s) {
  while(sz[C[u][0]] != s) {
    if(sz[C[u][0]] < s) {
      s -= sz[C[u][0]] + 1;
      u = C[u][1];
    } else u = C[u][0];
  }
  splay(u);
  return u;
}
// junte duas splays, assume que elementos l <= elementos r
int merge(int l, int r) {
  if(!l || !r) return l + r;
  while(C[l][1]) l = C[l][1];
  splay(l);
  assert(!C[l][1]);
  C[l][1] = r;
  p[r] = l;
  calc(l);
  return l;
}
// Adiciona no x a splay u e retorna x
int add(int u, int x) {
  int v = 0;
  while(u) v = u, u = C[u][X[x] >= X[u]];
  if(v) { C[v][X[x] >= X[v]] = x; p[x] = v; }
  splay(x);
  return x;
}
// chame isso 1 vez no inicio
void init() {
  en = 1;
}
// Cria um novo no
int new_node(num val) {
  int i = en++;
  assert(i < N);
  C[i][0] = C[i][1] = p[i] = 0;
  sz[i] = 1;
  X[i] = val;
  return i;
} // hash-cpp-all = 30e14f2069467aa6b27d51912e95775b
```

Wavelet.h

Time: $\mathcal{O}\left(log(MAXN - MINN)\right)$ 41 lines

```
int n, v[MAX];
vector<vector<int> > esq(4*(MAXN-MINN)), pref(4*(MAXN-MINN)
  ↪);

void build(int b = 0, int e = n, int p = 1, int l = MINN,
  ↪int r = MAXN) {
  int m = (l+r)/2; esq[p].push_back(0); pref[p].push_back
    ↪(0);
  for (int i = b; i < e; i++) {
    esq[p].push_back(esq[p].back()+(v[i]<=m));
    pref[p].push_back(pref[p].back()+v[i]);
  }
  if (l == r) return;
  int m2 = stable_partition(v+b, v+e, [=](int i){return i
    ↪<= m;}) - v;
  build(b, m2, 2*p, l, m), build(m2, e, 2*p+1, m+1, r);
}

int count(int i, int j, int x, int y, int p = 1, int l =
  ↪MINN, int r = MAXN) {
  if (y < l or r < x) return 0; //count(i, j, x, y) retorna
    ↪ o numero de elementos
  if (x <= l and r <= y) return j-i; // de v[i, j) que
    ↪pertencem a [x, y]
  int m = (l+r)/2, ei = esq[p][i], ej = esq[p][j];
  return count(ei, ej, x, y, 2*p, l, m)+count(i-ei, j-ej, x
    ↪, y, 2*p+1, m+1, r);
}

int kth(int i, int j, int k, int p=1, int l = MINN, int r =
  ↪ MAXN) {
  if (l == r) return l; //kth(i, j, k) retorna o elemento
    ↪que estaria na
  int m = (l+r)/2, ei = esq[p][i], ej = esq[p][j]; //
    ↪posi ao k-1 de v[i, j), se ele
  if (k <= ej-ei) return kth(ei, ej, k, 2*p, l, m); //
    ↪fosse ordenado
  return kth(i-ei, j-ej, k-(ej-ei), 2*p+1, m+1, r);
}

int sum(int i, int j, int x, int y, int p = 1, int l = MINN
  ↪, int r = MAXN) {
  if (y < l or r < x) return 0; // sum(i, j, x, y) retorna
    ↪a soma dos elementos de
  if (x <= l and r <= y) return pref[p][j]-pref[p][i]; // v
    ↪[i, j) que pertencem a [x, y]
  int m = (l+r)/2, ei = esq[p][i], ej = esq[p][j];
  return sum(ei, ej, x, y, 2*p, l, m) + sum(i-ei, j-ej, x,
    ↪y, 2*p+1, m+1, r);
}

int sumk(int i, int j, int k, int p = 1, int l = MINN, int
  ↪r = MAXN) {
  if (l == r) return l*k; //sumk(i, j, k) retorna a soma
    ↪dos k-esimos menores
  int m = (l+r)/2, ei = esq[p][i], ej = esq[p][j]; //
    ↪elementos de v[i, j) (sum(i, j, 1) retorna o menor)
  if (k <= ej-ei) return sumk(ei, ej, k, 2*p, l, m);
  return pref[2*p][ej]-pref[2*p][ei]+sumk(i-ei, j-ej, k-(ej
    ↪-ei), 2*p+1, m+1, r);
} // hash-cpp-all = 6773008405765704616aeb49df3c207e
```

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: $\mathcal{O}\left(\log((b-a)/\epsilon)\right)$ 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 $x[k] = c*\cos(k/(n-1)*\pi), k = 0 \dots n-1$.

Time: $\mathcal{O}\left(n^2\right)$ 13 lines

```cpp
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
```

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

"../number-theory/ModPow.h" 18 lines

```cpp
vector<lint> BerlekampMassey(vector<lint> s) {
  int n = s.size(), L = 0, m = 0;
  vector<lint> C(n), B(n), T;
  C[0] = B[0] = 1;
  lint b = 1;
  for(int i = 0; i < n; ++i) { ++m;
    lint d = s[i] % mod;
    for(int j = 1; j <= L; ++j) d = (d + C[j] * s[i - j]) %
      ↪ mod;
    if (!d) continue;
    T = C; lint coef = d * modpow(b, mod-2) % mod;
    for(int j = m; j < n; ++j) C[j] = (C[j] - coef * B[j -
      ↪m]) % mod;
    if (2 * L > i) continue;
    L = i + 1 - L; B = T; b = d; m = 0;
  }
  C.resize(L + 1); C.erase(C.begin());
  for(auto &x : C) x = (mod - x) % mod;
  return C;
} // hash-cpp-all = c2cac606a08f46cee205075412e2d163
```

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

22 lines

```cpp
typedef vector<lint> Poly;
lint linearRec(Poly S, Poly tr, lint k) { // hash-cpp-1
  int n = tr.size();
  auto combine = [&](Poly a, Poly b) {
    Poly res(n * 2 + 1);
    for(int i = 0; i < n+1; ++i) for(int j = 0; j < n+1; ++
      ↪j)
      res[i + j] = (res[i + j] + a[i] * b[j]) % mod;
    for (int i = 2 * n; i > n; --i) for(int j = 0; j < n;
      ↪++j)
      res[i - 1 - j] = (res[i - 1 - j] + res[i] * tr[j]) %
        ↪mod;
    res.resize(n + 1);
    return res;
  };
  Poly pol(n + 1), e(pol);
  pol[0] = e[1] = 1;
  for (++k; k; k /= 2) {
    if (k % 2) pol = combine(pol, e);
    e = combine(e, e);
  }
  lint res = 0;
  for(int i = 0; i < n; ++i) res = (res + pol[i + 1] * S[i
    ↪]) % mod;
  return res;
} // hash-cpp-1 = e5c828081215d4d0def337e161d7c0ec
```

HillClimbing.h

Description: Poor man's optimization for unimodal functions.

16 lines

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

double func(P p);

pair<double, P> hillClimb(P start) {
  pair<double, P> cur(func(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, make_pair(func(p), p));
    }
  }
  return cur;
} // hash-cpp-all = ac5d8e54c13316850419d034af305ebb
```

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.

8 lines

```cpp
double quad(double (*f)(double), double a, double b) {
  const int n = 1000;
  double h = (b - a) / 2 / n;
  double v = f(a) + f(b);
  for(int i = 1; i < n*2; ++i)
    v += f(a + i*h) * (i&1 ? 4 : 2);
  return v * h / 3;
} // hash-cpp-all = c777cd1327972e03cd5115614bba0213
```

IntegrateAdaptive.h

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

16 lines

```cpp
typedef double d;
d simpson(d (*f)(d), d a, d b) {
  d c = (a+b) / 2;
  return (f(a) + 4*f(c) + f(b)) * (b-a) / 6;
}
d rec(d (*f)(d), d a, d b, d eps, d S) {
  d c = (a+b) / 2;
  d S1 = simpson(f, a, c);
  d S2 = simpson(f, 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);
}
d quad(d (*f)(d), d a, d b, d eps = 1e-8) {
  return rec(f, a, b, eps, simpson(f, a, b));
} // hash-cpp-all = ad8a754372ce74e5a3d07ce46c2fe0ca
```

Determinant.h

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

15 lines

```cpp
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: $\mathcal{O}\left(N^3\right)$

18 lines

```cpp
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: Gaussian elimination

23 lines

```cpp
using T = double;
constexpr T EPS = 1e-8;

T elimination(vector<vector<double>> &m, int rows) { //
  ↪return the determinant
  int r = 0; T det = 1;             // MODIFIES the input
  for (int c = 0; c < rows && r < rows; ++c) {
    int p = r;
    for (int i = r+1; i < rows; ++i)
      if (fabs(m[i][c]) > fabs(m[p][c])) p=i;
    if (fabs(m[p][c]) < EPS){ det = 0; continue; }
    swap(m[p], m[r]);
```

```
    det = -det;
    T s = 1.0 / m[r][c], t; det *= m[r][c];
    for(int j = 0; j < C; ++j) m[r][j] *= s; // make leading term in row 1
    for(int i = 0; i < rows; ++i)
      if (i != r){
        t = m[i][c];
        for(int j = 0; j < C; ++j) m[i][j] -= t * m[r][j];
      }
    ++r;
  }
  return det;
} // hash-cpp-all = 6bf7c77ee9924912326017117030246c
```

Simplex.h

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

67 lines

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

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

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

  LPSolver(const vvd& A, const vd& b, const vd& c) :
    m(b.size()), n(c.size()), N(n+1), B(m), D(m+2, vd(n+2)) { // hash-cpp-1
      for(int i = 0; i < m; ++i) for(int j = 0; j < n; ++j) D[i][j] = A[i][j];
      for(int i = 0; i < m; ++i) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i];}
      for(int j = 0; j < n; ++j) { N[j] = j; D[m][j] = -c[j]; }
      N[n] = -1; D[m+1][n] = 1;
    } // hash-cpp-1 = 4117b6540107f175bea8c274b78900ec

  void pivot(int r, int s) { // hash-cpp-2
    T *a = D[r].data(), inv = 1 / a[s];
    for(int i = 0; i < m+2; ++i) if (i != r && abs(D[i][s]) > EPS) {
      T *b = D[i].data(), inv2 = b[s] * inv;
      for(int j = 0; j < n+2; ++j) b[j] -= a[j] * inv2;
      b[s] = a[s] * inv2;
    }
    for(int j = 0; j < n+2; ++j) if (j != s) D[r][j] *= inv;
    for(int i = 0; i < m+2; ++i) if (i != r) D[i][s] *= -inv;
    D[r][s] = inv;
    swap(B[r], N[s]);
  } // hash-cpp-2 = eb7407eedd4b75013eb919cdd9b49bb9
  bool simplex(int phase) { // hash-cpp-3
    int x = m + phase - 1;
    while(1) {
      int s = -1;
      for(int j = 0; j <= n; ++j) if (N[j] != -phase) ltj(D[x]);
      if (D[x][s] >= -EPS) return true;
      int r = -1;
      for(int i = 0; i < m; ++i) {
        if (D[i][s] <= EPS) continue;
        if (r == -1 || MP(D[i][n+1] / D[i][s], B[i])
                     < MP(D[r][n+1] / D[r][s], B[r])) r = i;
      }
      if (r == -1) return false;
      pivot(r, s);
    }
  } // hash-cpp-3 = 26e9d7a1fdbdf10716560edf3c22e380

  T solve(vd &x) { // hash-cpp-4
    int r = 0;
    for(int i = 1; i < m; ++i) if (D[i][n+1] < D[r][n+1]) r = i;
    if (D[r][n+1] < -EPS) {
      pivot(r, n);
      if (!simplex(2) || D[m+1][n+1] < -EPS) return -inf;
      for(int i = 0; i < m; ++i) if (B[i] == -1) {
        int s = 0;
        for(int j = 1; j < n+1; ++j) ltj(D[i]);
        pivot(i, s);
      }
    }
    bool ok = simplex(1); x = vd(n);
    for(int i = 0; i < m; ++i) if (B[i] < n) x[B[i]] = D[i][n+1];
    return ok ? D[m][n+1] : inf;
  } // hash-cpp-4 = 62464e86efb3a9eacee43961eba3b2e0
};
```

Math-Simplex.cpp

Description: Simplex algorithm. WARNING- segfaults on empty (size 0) max cx st Ax<=b, x>=0 do 2 phases; 1st check feasibility; 2nd check boundedness and ans

40 lines

```
vector<double> simplex(vector<vector<double> > A, vector<double> b, vector<double> c) {
  int n = (int) A.size(), m = (int) 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];
  printf("%.2lf\n", D[n][m]);
  return x; // ans: D[n][m]
} // hash-cpp-all = 70201709abdff05eff90d9393c756b95
```

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

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 $A^{-1} = A^{-1}(2I - AA^{-1}) \pmod{p^k}$ where A^{-1} starts as the inverse of A mod p, and k is doubled in each step.
Time: $\mathcal{O}(n^3)$

35 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
```

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

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

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

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

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

20 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;
    for (;;) {
        double nx = x - (f(x)/df(x));
        if (abs(x - nx) < eps)
            break;
        x = nx;

    }
    return x;
} // hash-cpp-all = 0c23f37312d265c04200134bc0c5a5a6
```

NewtonSQRT.h

Description: Square root find method

24 lines

```
double sqrt_newton(double n) {
    const double eps = 1E-15;
    double x = 1;
    for (;;) {
        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;
    for (;;) {
        int nx = (x + n / x) >> 1;
        if (x == nx || nx > x && decreased)
            break;
        decreased = nx < x;
        x = nx;
    }
    return x;
} // hash-cpp-all = 0fb4aaf4827ce1febbd3734769a737d5
```

MarkovChain.cpp

Description: Markov Chain

37 lines

```
//1-indexed
int adj[N][N]; //adj matrix
int out[N]; // out degree of the state
double trans[N][N], prob[N];

void create_prob(int n, int s=1){
  for(int i=1;i<=n;i++) prob[i]=0;
  prob[s]=1;
}
void create_chain(int n){
  for(int i=1;i<=n;i++){
    if(out[i])
      for(int j=1;j<=n;j++){
        if(adj[i][j]) trans[i][j]=((double)adj[i][j])/out[i
            ↪];
        else trans[i][j]=0;
      }
    else
      for(int j=1;j<=n;j++) trans[i][j]=1.0/n;
  }
}
double proxprob[N];
int aplica(int n){
  for(int i=1;i<=n;i++) proxprob[i]=0;
  for(int i=1;i<=n;i++)
    for(int j=1;j<=n;j++)
      proxprob[i]+=prob[j]*trans[j][i];
  int dif=0;
  for(int i=1;i<=n;i++){
    dif+=abs(cmp_double(prob[i],proxprob[i]));
    prob[i]=proxprob[i];
  }
  return dif;
}
void solve(int n){
  while(aplica(n));
}
// hash-cpp-all = a510019cf7664803ea2e0bdc4c24d902
```

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 $N = 2^{22}$)

36 lines

```
typedef complex<long double> doublex;
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);
    }
} fft;
// hash-cpp-all = 26c9ae5b309bb520a31e6e6531b4cb6b
```

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)

63 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; }
};
```

```
// hash-cpp-all = 75ca28e040bf2dc37c26385f44775e38
```

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, 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 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:

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

4.1.2 Generating functions

Ordinary (ogf): $A(x) := \sum_{n=0}^{\infty} a_i x^i$.

Calculate product $c_n = \sum_{k=0}^{n} a_k b_{n-k}$ with FFT.

Exponential (e.g.f.): $A(x) := \sum_{n=0}^{\infty} a_i x^i / i!$,

$c_n = \sum_{k=0}^{n} \binom{n}{k} a_k b_{n-k} = n! \sum_{k=0}^{n} \frac{a_k}{k!} \frac{b_{n-k}}{(n-k)!}$ (use FFT).

4.1.3 General linear recurrences

If $a_n = \sum_{k=0}^{n-1} a_k b_{n-k}$, then $A(x) = \frac{a_0}{1-B(x)}$.

4.1.4 Inverse polynomial modulo x^l

Given $A(x)$, find $B(x)$ such that $A(x)B(x) = 1 + x^l Q(x)$ for some $Q(x)$.

Step 1: Start with $B_0(x) = 1/a_0$

Step 2: $B_{k+1}(x) = (-B_k(x)^2 A(x) + 2B_k(x))$ mod $x^{2^{k+1}}$.

4.1.5 Fast subset convolution

Given array a_i of size 2^k calculate $b_i = \sum_{ji=i} a_j$.

4.1.6 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

4.1.7 Table of non-trigonometric integrals

Some useful integrals are:

$\int \frac{dx}{x^2+a^2}$	$\frac{1}{a}\arctan\frac{x}{a}$
$\int \frac{dx}{x^2-a^2}$	$\frac{1}{2a}\ln\frac{x-a}{x+a}$
$\int \frac{dx}{a^2-x^2}$	$\frac{1}{2a}\ln\frac{a+x}{a-x}$
$\int \frac{dx}{\sqrt{a^2-x^2}}$	$\arcsin\frac{x}{a}$
$\int \frac{dx}{\sqrt{x^2-a^2}}$	$\ln\left(u+\sqrt{x^2-a^2}\right)$
$\int \frac{dx}{x\sqrt{x^2-a^2}}$	$\frac{1}{a}\operatorname{arcsec}\left\lvert\frac{u}{a}\right\rvert$
$\int \frac{dx}{x\sqrt{x^2+a^2}}$	$-\frac{1}{a}\ln\left(\frac{a+\sqrt{x^2+a^2}}{x}\right)$
$\int \frac{dx}{x\sqrt{a^2+x^2}}$	$-\frac{1}{a}\ln\left(\frac{a+\sqrt{a^2-x^2}}{x}\right)$

4.1.8 Table of trigonometric integrals

A list of common and not-so-common trigonometric integrals:

$\int \tan x dx$	$-\ln\lvert\cos x\rvert$
$\int \cot x dx$	$\ln\lvert\sin x\rvert$
$\int \sec x dx$	$\ln\lvert\sec x+\tan x\rvert$
$\int \csc x dx$	$\ln\lvert\csc x-\cot x\rvert$
$\int \sec^2 x dx$	$\tan x$
$\int \csc^2 x dx$	$\cot x$
$\int \sin^n x dx$	$\frac{-\sin^{n-1}x\cos x}{n}+\frac{n-1}{n}\int\sin^{n-2}x dx$
$\int \cos^n x dx$	$\frac{\cos^{n-1}x\sin x}{n}+\frac{n-1}{n}\int\cos^{n-2}x dx$
$\int \arcsin x dx$	$x\arcsin x+\sqrt{1-x^2}$
$\int \arccos x dx$	$x\arccos x-\sqrt{1-x^2}$
$\int \arctan x dx$	$x\arctan x-\frac{1}{2}\ln\lvert 1-x^2\rvert$

4.1.9 Common integral substitutions

And finally, a list of common substitutions:

$\int F(\sqrt{ax+b})dx$	$u=\sqrt{ax+b}$	$\frac{2}{a}\int uF(u)du$
$\int F(\sqrt{a^2-x^2})dx$	$x=a\sin u$	$a\int F(a\cos u)\cos u du$
$\int F(\sqrt{x^2+a^2})dx$	$x=a\tan u$	$a\int F(a\sec u)\sec^2 u du$
$\int F(\sqrt{x^2-a^2})dx$	$x=a\sec u$	$a\int F(a\tan u)\sec u\tan u du$
$\int F(e^{ax})dx$	$u=e^{ax}$	$\frac{1}{a}\int\frac{F(u)}{u}du$
$\int F(\ln x)dx$	$u=\ln x$	$\int F(u)e^u du$

4.1.10 Determinants and PM

$$det(A)=\sum_{\sigma\in S_n}\operatorname{sgn}(\sigma)\prod_{i=1}^{n}a_{i,\sigma(i)}$$

$$perm(A)=\sum_{\sigma\in S_n}\prod_{i=1}^{n}a_{i,\sigma(i)}$$

$$pf(A)=\frac{1}{2^n n!}\sum_{\sigma\in S_{2n}}\operatorname{sgn}(\sigma)\prod_{i=1}^{n}a_{\sigma(2i-1),\sigma(2i)}$$
$$=\sum_{M\in \mathrm{PM}(n)}\operatorname{sgn}(M)\prod_{(i,j)\in M}a_{i,j}$$

Number theory (5)

5.1 Modular arithmetic

ModTemplate.h

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

58 lines

```
template <int MOD_>  struct modnum {
private:
  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 = 1a737d9328214e97ad11f393d949d0af
```

PairNumTemplate.h

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

54 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 $ax \equiv 1 (\bmod\ m)$. The inverse only exist if a and m are coprimes.

14 lines

```
lint modinv(lint a, int 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);
} // hash-cpp-all = c736e149bf535a5b25c73ab2528a0ef1
```

Modpow.h

6 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;
} // hash-cpp-all = 31ce91e32da17e303efb71194e126157
```

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.

19 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;
}

ll modsum(ull to, ll c, ll k, ll m) {
  c = ((c % m) + m) % m;
  k = ((k % m) + m) % m;
  return to * c + k * sumsq(to) - m * divsum(to, c, k, m);
} // hash-cpp-all = 8d6e082e0ea6be867eaea12670d08dcc
```

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

13 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. $x^2 = a \mod p$
Time: $\mathcal{O}(\log^2 p)$ worst case, often $\mathcal{O}(\log p)$

"ModPow.h" 30 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);
  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;
  int r = 0;
  while (s % 2 == 0)
    ++r, s /= 2;
  lint n = 2; // find a non-square mod p
  while (modpow(n, (p - 1) / 2, p) != p - 1) ++n;
  lint x = modpow(a, (s + 1) / 2, p);
  lint b = modpow(a, s, p);
  lint g = modpow(n, s, p);
  for (;;) {
    lint t = b;
    int m = 0;
    for (; m < r; ++m) {
      if (t == 1) break;
      t = t * t % p;
    }
    if (m == 0) return x;
    lint gs = modpow(g, 1 << (r - m - 1), p);
    g = gs * gs % p;
    x = x * gs % p;
    b = b * g % p;
    r = m;
  }
} // hash-cpp-all = c5802872a799af812a29e13208ef8e63
```

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);
    pair<int,int> 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 = b3fb0f17b93555f29edba04fd05433b9
```

Quadratic.h

Description: Solve $x^2 \equiv n \mod p (0 \le a < p)$ where p is prime in $O(\log p)$.

18 lines

```
struct quadric {
  void multiply(lint &c, lint &d, lint a, lint b, lint w,
     ↪lint p) { // hash-cpp-1
    int cc = (a * c + b * d % p * w) % p;
    int dd = (a * d + b * c) % p; c = cc, d = dd; }
  bool solve(int n, int p, int &x) {
    if (n == 0) return x = 0, true; if (p == 2) return x =
       ↪1, true;
    if (mod_pow(n, p / 2, p) == p - 1) return false;
    lint c = 1, d = 0, b = 1, a, w;
    do { a = rand() % p; w = (a * a - n + p) % p;
      if (w == 0) return x = a, true;
    } while (mod_pow(w, p / 2, p) != p - 1);
    for (int times = (p + 1) / 2; times; times >>= 1) {
      if (times & 1) multiply (c, d, a, b, w, p);
      multiply (a, b, a, b, w, p);
      }
    return x = c, true;
    } // hash-cpp-1 = 7b06e39b96dbf9618c8735bc05ee61f4
};
```

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 prime.
Time: lim=100'000'000 $\approx$ 0.8 s. Runs 30% faster if only odd indices are stored.

12 lines

```
const int MAX_PR = 5000000;
bitset<MAX_PR> isprime;
vector<int> run_sieve(int lim) {
  isprime.set(); isprime[0] = isprime[1] = 0;
  for (int i = 4; i < lim; i += 2) isprime[i] = 0;
  for (int i = 3; i*i < lim; i += 2) if (isprime[i])
    for (int j = i*i; j < lim; j += i*2) isprime[j] = 0;
  vector<int> pr;
  for(int i = 2; i < lim; ++i) if (isprime[i]) pr.push_back
      ↪(i);
  return pr;
}
// hash-cpp-all = 90c90fa5012933c478f6aa1f7cb230f8
```

LinearSieve.h

Description: Prime sieve for generating all primes up to a certain limit.
Time: $\mathcal{O}(n)$

19 lines

```
vector<int> least = {0, 1};
vector<int> primes;
int precalculated = 1;
void LinearSieve(int n) {
    n = max(n, 1);
    least.assign(n + 1, 0);
    primes.clear();
    for (int i = 2; i <= n; i++) {
        if (least[i] == 0) {
            least[i] = i;
            primes.push_back(i);
        }
        for (int p : primes) {
            if (p > least[i] || i * p > n) break;
            least[i * p] = p;
        }
    }
    precalculated = n;
} // hash-cpp-all = 126ac7f141d28a888e2d52e4be549215
```

MobiusSieve.h

Description: Pre calculate all mobius values.
Time: $\mathcal{O}(sqrt(n))$

19 lines

```
vector<int> mobius, lp;
void run_sieve(int n) {
    mobius.assign(n, -1);
    lp.assign(n, 0);
    mobius[1] = 1;
    vector<int> prime;
    for (int i = 2; i <= n; ++i) {
        if (!lp[i]) {
            lp[i] = i;
            prime.push_back(i);
        }
        for (int p : prime) {
            if (p > lp[i] || p*i > n) break;
            if (i % p == 0) mobius[i*p] = 0;
            lp[p*i] = p;
            mobius[p*i] *= mobius[i];
        }
    }
} // hash-cpp-all = 703869420dc1768d2e5c331701a3d2df
```

Mobius.h

Description: 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: $\mathcal{O}(sqrt(n))$

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

"ModMulLL.h" 16 lines

```
bool prime(ull p) {
  if (p == 2) return true;
  if (p == 1 || p % 2 == 0) return false;
  ull s = p - 1;
  while (s % 2 == 0) s /= 2;
  for(int i = 0; i < 15; ++i) {
    ull a = rand() % (p - 1) + 1, tmp = s;
    ull mod = mod_pow(a, tmp, p);
    while (tmp != p - 1 && mod != 1 && mod != p - 1) {
      mod = mod_mul(mod, mod, p);
      tmp *= 2;
    }
    if (mod != p - 1 && tmp % 2 == 0) return false;
  }
  return true;
} // hash-cpp-all = fb55ec6f40b2863372ede8e76b147391
```

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: $\mathcal{O}\left(n^{1/4}\right)$ gcd calls, 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) % n;
      ↪};
  if (!(n & 1)) return 2;
  for (ull i = 2;; i++) {
    ull x = i, y = f(x), p;
    while ((p = __gcd(n + y - x, n)) == 1)
      x = f(x), y = f(f(y));
    if (p != n) return p;
  }
}
vector<ull> factor(ull n) {
  if (n == 1) return {};
  if (isPrime(n)) return {n};
  ull x = pollard(n);
  auto l = factor(x), r = factor(n / x);
  l.insert(l.end(), all(r));
  return l;
} // hash-cpp-all = f5adaa4517c8c7f5812dd65047dab785
```

5.3 Divisibility

ExtendedEuclidean.h

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

11 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;
    T g = egcd(b - p * a, a, y, x);
    x -= y * p;
    return g;
} // hash-cpp-all = a11e6c47ddaed024be9201844cff1da9
```

Euclid.java

Description: Finds {x, y, d} s.t. ax + by = d = gcd(a, b).

11 lines

```
static BigInteger[] euclid(BigInteger a, BigInteger b) {
  BigInteger x = BigInteger.ONE, yy = x;
  BigInteger y = BigInteger.ZERO, xx = y;
  while (b.signum() != 0) {
    BigInteger q = a.divide(b), t = b;
    b = a.mod(b); a = t;
    t = xx; xx = x.subtract(q.multiply(xx)); x = t;
    t = yy; yy = y.subtract(q.multiply(yy)); y = t;
  }
  return new BigInteger[]{x, y, a};
}
```

DiophantineEquation.h

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

39 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;
    } // hash-cpp-1 = b6de1e1af6bb4f670fb53e9f8abf08b5
// hash-cpp-2
    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|)
} // hash-cpp-2 = a8604c857ce66f7c6cb5d318ece21e1c
```

Divisors.h

Description: Get all divisors of n.

15 lines

```
vector<int> divisors(int n) {
    vector<int> ret, ret1;
    for (int i = 1; i*i <= n; ++i) {
        if (n % i == 0) {
            ret.push_back(i);
            int d = n / i;
            if (d != i) ret1.push_back(d);
        }
    }
    if (!ret1.empty()) {
        reverse(ret1.begin(), ret1.end());
        ret.insert(ret.end(), ret1.begin(), ret1.end());
    }
    return ret;
 } // hash-cpp-all = 325815a4263d6fd7fac1bf3aee29d4d6
```

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 $x_{k+1} = x_0x_k + ny_0y_k, y_{k+1} = x_0y_k + y_0x_k$.

17 lines

```
template <int MAXN = 100000>
struct pell {
  pair <lint, lint> solve (lint n) { // hash-cpp-1
    static lint p[MAXN], q[MAXN], g[MAXN], h[MAXN], a[MAXN
       ↪];
    p[1] = q[0] = h[1] = 1; p[0] = q[1] = g[1] = 0;
    a[2] = (lint)(floor(sqrtl(n) + 1e-7L));
    for (int i = 2; ; ++i) {
      g[i] = -g[i - 1] + a[i] * h[i - 1];
      h[i] = (n - g[i] * g[i]) / h[i - 1];
      a[i + 1] = (g[i] + a[2]) / h[i];
      p[i] = a[i] * p[i - 1] + p[i - 2];
      q[i] = a[i] * q[i - 1] + q[i - 2];
      if (p[i] * p[i] - n * q[i] * q[i] == 1)
        return { p[i], q[i] };
      }
    } // hash-cpp-1 = bf2eeb000f9cca352ec13820f6fd8002
};
```

PrimeFactors.h

Description: Find all prime factors of n.

13 lines

```
vector<lint> primeFac(lint n){
    vector<int> factors;
    lint 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 = 018bb495892889b74fb4a13e722eb642
```

NumDiv.h

Description: Count the number of divisors of n.

14 lines

```
lint NumDiv(lint n){
    lint idx = 0, prime_factors = primes[idx], ans = 1;
    while (prime_factors * prime_factors <= n) {
        lint power = 0;
        while (n % prime_factors == 0) {
            n /= prime_factors;
            power++;
        }
        ans *= (power + 1);
        prime_factors = primes[++idx];
    }
    if (n != 1) ans *= 2;
    return ans;
} // hash-cpp-all = 267d11d419ad89e15f3a1320a6a9998e
```

NumPF.h

Description: Find the number o prime factors of n.

12 lines

```
lint nPrimeFac(lint n){
    lint idx = 0, prime_factors = primes[idx], ans = 0;
    while (prime_factors * prime_factors <= n){
        while (n % prime_factors == 0) {
            n /= prime_factors;
            ans++;
        }
        prime_factors = primes[++idx];
    }
    if (n != 1) ans++;
    return ans;
} // hash-cpp-all = 4e5c87d13b378e5b10ec0e472be9a3c8
```

SumDiv.h

Description: Sum of all divisors of n.

14 lines

```
lint nPrimeFac(lint n){
    lint idx = 0, prime_factors = primes[idx], ans = 1;
    while (prime_factors * prime_factors <= n){
        lint power = 0;
        while (n % prime_factors == 0) {
            n /= prime_factors;
            power++;
        }
        ans *= ((lint)pow((double)prime_factors, power+1.0)
           ↪-1)/(prime_factors-1);
        prime_factors = primes[++idx];
    }
    if (n != 1) ans *= ((lint)pow((double)n, 2.0)-1)/(n-1);
    return ans;
} // hash-cpp-all = 55a3ae63024dc0e124b029679ece3bb4
```

GoldbachConjecture.cpp

Description: Every even integer greater than 2 can be expressed as the sum of two primes.

8 lines

```
vector<pair<int, int>> Goldbach(int n){
    int ret = 0;
    for(int i = 2; i <= n/2; ++i)
        if (primes[i] && primes[n-i]){
              g.emplace_back(i, n-i);
        }
    return g;
} // hash-cpp-all = ea3600c179a4474b61d1ddc2720a53e2
```

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
```

EulerPhi.h

17 lines

```
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 = dc8aed24643ac9dab4044ef1930fdae5
```

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

7 lines

```
const int LIM = 500000;
vector<lint> phi(LIM);
iota(phi.begin(), phi.end(), 0);
for(int i = 1; i <= LIM; ++i)
    for (int j = i+i; j <= LIM; j += i)
        phi[j] -= phi[i];
// hash-cpp-all = 810d2a94056a165391351309be03d9e9
```

DiscreteLogarithm.h

Description: Returns the smallest $x \geq 0$ s.t. $a^x = b \pmod{m}$. a and m must be coprime.
Time: $\mathcal{O}\left(\sqrt{m}\right)$

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

19 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;
}

template<typename T>
T order(T a, T n) {
    vector<T> d = divisors(phi(n));
    for (int i : v)
        if (mod_pow(a, i, n) == 1) return i;
    return -1;
} // hash-cpp-all = 018bfc5c9e761dd00e925b251f8991b8
```

Bet.h

5.4 Fractions

Fractions.h

Description: Template that helps deal with fractions.

37 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) {}
// hash-cpp-1 = 17a225028ef124d7c631b9429ca0a2f5
// hash-cpp-2
    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;
    } // hash-cpp-2 = 8ede570ec532c0d2ce01dbec6f97bc9f
};
```

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: $\mathcal{O}\left(\log N\right)$

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: $\mathcal{O}\left(\log(N)\right)$

24 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 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 $0 \le x < \text{lcm}(m, n)$. Assumes $mn < 2^{62}$.

Time: $\mathcal{O}(\log(n)) - \mathcal{O}(n \log(LCM(m)))$ 22 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;
}
```

5.6 Pythagorean Triples

The Pythagorean triples are uniquely generated by

$$a = k \cdot (m^2 - n^2), \;\; b = k \cdot (2mn), \;\; c = k \cdot (m^2 + n^2),$$

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

5.7 Primes

$p = 962592769$ is such that $2^{21} \mid p - 1$, which may be useful. For hashing use 970592641 (31-bit number), 31443539979727 (45-bit), 3006703054056749 (52-bit). There are 78498 primes less than 1 000 000.

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

5.7.1 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).

Sum of primes

For any multiplicative f:

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

5.7.2 Chicken McNugget Theorem

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

5.7.3 Wilson's Theorem

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

5.7.4 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.

5.7.5 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

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

5.7.6 Möbius Inversion Formula

Se $F(n) = \sum_{d|n} f(d)$, então $f(n) = \sum_{d|n} \mu(d) F(n/d)$.

5.7.7 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 $n < 1e19$.

5.7.8 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: Precalculate factorials 21 lines

```
void pre(int lim) {
    fact.resize(lim + 1);
    fact[0] = 1;
    for (int i = 1; i <= lim; ++i)
        fact[i] = (lint)i * fact[i - 1] % mod;
    inv_fact.resize(lim + 1);
    inv_fact[lim] = inv(fact[lim], mod);
    for (int i = lim - 1; i >= 0; --i)
        inv_fact[i] = (lint)(i + 1) * inv_fact[i + 1] % mod
           ↪;
}

void init() {
  fact = {1};
  for(int i = 1; i < 1010; i++)
    fact.push_back(i * fact[i-1]);
  ifact.resize(fact.size());
  ifact.back() = 1/fact.back();
  for(int i = (int)ifact.size()-1; i > 0; i--)
    ifact[i-1] = i * ifact[i];
}
// hash-cpp-all = 8335c8e6a73532159f4162c49cb51ae6
```

IntPerm.h

Description: Permutation -> integer conversion. (Not order preserving.)

Time: $\mathcal{O}(n)$ 6 lines

```
int permToInt(vector<int>& v) {
  int use = 0, i = 0, r = 0;
  for (auto &x : v) r=r * ++i + __builtin_popcount(use &
     ↪-(1 << x)),
    use |= 1 << x;                // (note: minus, not ~!)
  return r;
} // hash-cpp-all = 06f786fbb6d782621d3ecfd9a38c2601
```

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 Cycles

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

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

6.1.3 Derangements

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

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

6.1.4 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

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

6.1.5 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.	$\left\{{n \atop 1}\right\}+\ldots+\left\{{n \atop x}\right\}$	$[n \le x]$	$\left\{{n \atop k}\right\}$
indist.	indist.	$p_1(n)+\ldots p_x(n)$	$[n \le x]$	$p_x(n)$

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).

6.1.6 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

6.1.7 Burnside

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

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

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

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

6.2.3 Binomials

nCr.h
Description: ${}^nC_r$
13 lines

```
lint ncr(lint n, lint r){
    if(r < 0 || n < 0) return 0;
    if(n < r) return 0;
    lint a = fact[n];
    a = (a * invfact[r]) % mod;
    a = (a * invfact[n-r]) % mod;
    return a;
}

num ncr(int n, int k){
  if(k < 0 || k > n) return 0;
  return fact[n] * ifact[k] * ifact[n-k];
} // hash-cpp-all = 321ddb6eb353b8c75a4c0be672ceb75d
```

NWayDistribute.h
Description: Stars and Bars technique. How many ways can one distribute k indistinguishable objects into n bins. $\binom{n+k-1}{k}$
6 lines

```
int get_nway_distribute(int many, int npile) {
  if (many == 0)
    return npile == 0;
  many -= npile;
  return ncr(many + npile - 1, npile - 1);
} // hash-cpp-all = 71dd7e7dc0c40896d1e7f8ce428304ad
```

PascalTriangle.h
Description: Pre-compute all binomial coefficient
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
```

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

Catalan.h
Description: Pre calculate Catalan numbers.
<ModTemplate.h>
9 lines

```
num catalan[MAX];
void pre() {
    catalan[0] = catalan[1] = 1;
    for (int i = 2; i <= n; ++i) {
        catalan[i] = 0;
        for (int j = 0; j < i; ++j)
            catalan[i] += catalan[j] * catalan[i-j-1];
    }
} // hash-cpp-all = e99e44501c3c9cd841cf3a61de1a8e6b
```

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.

$c(n,k) = c(n-1,k-1) + (n-1)c(n-1,k), c(0,0) = 1$
$\sum_{k=0}^{n} c(n,k)x^k = x(x+1)\dots(x+n-1)$

$c(8,k) = 8, 0, 5040, 13068, 13132, 6769, 1960, 322, 28, 1$
$c(n,2) =$
$0, 0, 1, 3, 11, 50, 274, 1764, 13068, 109584, \dots$

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. $B(n) = 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, \dots$.

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

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

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

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

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

- 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 $X \ne 0$. 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 $\bigoplus_{i=1}^{n} x_i$.

Grundy.h

20 lines

```
typedef unsigned long long ulint;
const int max_size = 60;
map<pair<int, ulint>, int> grundy;

int get_grundy(int n, ulint used) {
    int contains_adj[max_size];
    auto it = grundy.find({n, used});
    if (it != grundy.end()) return it->second;
    fill(contains_adj, contains_adj + max_size, 0);
    for (int remove = 1; remove <= n; ++remove)
        if (!(used & (1ULL << remove))) {
            int adj_state = get_grundy(n - remove, used |
                (1ULL << remove));
            if (adj_state < max_size)
                contains_adj[adj_state] = 1;
        }
    int result = 0;
    while (result < max_size && contains_adj[result])
        ++result;
    return grundy[{n, used}] = result;
} // hash-cpp-all = d8af5a876c8ce49f1f7a986de56bf686
```

Nim.cpp

Description: Sprague-grundy theorem. Example.

41 lines

```
const int MAXN = 1010;
int version;
int used[MAXN];

int mex() {
  for(int i=0; ; ++i)
    if(used[i] != version)
      return i;
}

int g[MAXN];
//remover 1, 2, 3
void grundy(){
  //Base case depends on the problem
  g[0] = 0;
  g[1] = 1;
  g[2] = 2;
  g[3] = 3;
  //Inductive case
  for(int i = 3; i < MAXN; ++i) {
    version++;
    used[g[i-1]] = version;
    used[g[i-2]] = version;
    used[g[i-3]] = version;
    g[i] = mex();
  }
}

int main() {
  grundy();
  int n;
  cin >> n;
  int ans = 0;
  for(int i=0; i<n; i++){
    int x;
    cin >> x;
    ans ^= g[x];
  }
  cout << ((ans != 0) ? "First" : "Second") << endl;
  return 0;
} // hash-cpp-all = 546385acc4ace07fc387d40e191d68c3
```

Nim-Product.cpp

Description: Nim Product.

17 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)));
}
ull nimProd(ull x, ull y) {
  ull res = 0;
  for (int i = 0; x >> i; i++)
    if ((x >> i) & 1)
      for (int j = 0; y >> j; j++)
        if ((y >> j) & 1)
          res ^= nimProd2(i, j);
  return res;
} // hash-cpp-all = e0411498c7a77d77ae793efab5500851
```

Schreier-Sims.cpp

Description: Check group membership of permutation groups

52 lines

```
struct Perm {
  int a[N];
  Perm() {
    for (int i = 1; i <= n; ++i) a[i] = i;
  }
  friend Perm operator* (const Perm &lhs, const Perm &rhs)
    {
    static Perm res;
    for (int i = 1; i <= n; ++i) res.a[i] = lhs.a[rhs.a[i
      ]];
    return res;
  }
  friend Perm inv(const Perm &cur) {
    static Perm res;
    for (int i = 1; i <= n; ++i) res.a[cur.a[i]] = i;
    return res;
  }
};
class Group {
```

```
  bool flag[N];
  Perm w[N];
  std::vector<Perm> x;
public:
  void clear(int p) {
    memset(flag, 0, sizeof flag);
    for (int i = 1; i <= n; ++i) w[i] = Perm();
    flag[p] = true;
    x.clear();
  }
  friend bool check(const Perm&, int);
  friend void insert(const Perm&, int);
  friend void updateX(const Perm&, int);
} g[N];
bool check(const Perm &cur, int k) {
  if (!k) return true;
  int t = cur.a[k];
  return g[k].flag[t] ? check(g[k].w[t] * cur, k - 1) :
    ↪false;
}
void updateX(const Perm&, int);
void insert(const Perm &cur, int k) {
  if (check(cur, k)) return;
  g[k].x.push_back(cur);
  for (int i = 1; i <= n; ++i) if (g[k].flag[i]) updateX(
    ↪cur * inv(g[k].w[i]), k);
}
void updateX(const Perm &cur, int k) {
  int t = cur.a[k];
  if (g[k].flag[t]) {
    insert(g[k].w[t] * cur, k - 1);
  } else {
    g[k].w[t] = inv(cur);
    g[k].flag[t] = true;
    for (int i = 0; i < g[k].x.size(); ++i) updateX(g[k].x[
      ↪i] * cur, k);
  }
} // hash-cpp-all = 949a6e50dbdaea9cda09928c7eabedbc
```

RandomWalk.h

Description: Probability of reaching N(winning) Variation - Loser gives a coin to the winner

<Modpow.h> 6 lines

```
// pmf = probability of moving forward
double random_walk(double p, int i, int n) {
    double q = 1 - p;
    if (fabs(p - q) < EPS) return 1.0 * i/n;
    return (1 - modpow(q/p, i))/(1 - modpow(q/p, n));
} // hash-cpp-all = 71c0095f96b65c6e75a9016180a4c3b5
```

Partitions.cpp

Description: Fills array with partition function p(n) for 0 <= i <= n

17 lines

```
array<int, 122> part; // 121 is max partition that will fit
  ↪ into int
void get_part(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 = b65a851e64795540d1c97c809b312d11
```

Lucas.h

Description: Lucas theorem

Time: $\mathcal{O}(log_p(n) * mod_inverse())$

<ModTemplate.h>, <nCr.h> 11 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: $\mathcal{O}(VE)$

21 lines

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

void bellmanFord(vector<Node>& nodes, vector<Ed>& eds, int
  ↪s) {
  nodes[s].dist = 0;
  sort(eds.begin(), eds.end(), [](Ed a, Ed 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 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 = 62f3d4db997360483e6628d5373994af
```

FloydWarshall.h

Description: Calculates alint-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: $\mathcal{O}(N^3)$

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

14 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

31 lines

```
vector<int> cut, mark, low, par;
vector<vector<int>> edges;
int Time = 0;

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

```
    low.resize(n, 0);
    par.resize(n, 0);
    for (int i = 0; i < n; ++i)
        if (mark[i] == -1) {
            Time = 0;
            dfs(i, i);
        }
} // hash-cpp-all = 23e6fcdbd3ffa84a303354844e44c8bb
```

Bridges.h

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

19 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: $\mathcal{O}(E \log V)$

25 lines

```
struct prim_t {
    int n;
    vector<vector<pair<int,int>>> edges;
    vector<bool> chosen;
    priority_queue<pair<int, int>> pq;
    prim_t(int _n) : n(_n), edges(n), chosen(n, false) {}
    void process(int u) { //inicializa com process(0)
        chosen[u] = true;
        for (int j = 0; j < (int) edges[u].size(); j++) {
            pair<int, int> v = edges[u][j];
            if (!chosen[v.first]) pq.push(make_pair(-v.
               ↪second, -v.first));
        }
    }
    int solve() {
        int mst_cost = 0;
        while (!pq.empty()) {
            pair<int,int> front = pq.top();
            pq.pop();
            int u = -front.second, w = -front.first;
            if (!chosen[u]) mst_cost += w;
          process(u);
        }
        return mst_cost;
    }
}; // hash-cpp-all = 90c7fbd244c2256ac8a3f1904a719ca5
```

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(vector<pair<T, pair<int,int>>> &edges) {
    sort(edges.begin(), edges.end());
    T cost = 0;
    UF dsu(edges.size());
    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 = f407f7a7396721b7868a52e8cf876e95
```

7.1.1 Landau

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

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

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

7.1.2 Matroid Intersection Theorem

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

7.1.3 Vizing's Thereom

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

- $\chi(G) = \delta$ ou $\chi(G) = \delta + 1$.

7.1.4 Euler's Theorem

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

7.1.5 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.

7.1.6 Dilworth's Thereom

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.

7.1.7 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 $|N(S)| \ge |S|$ ∀ S $\subset V_1$.

7.1.8 Maximum Density Subgraph

Given (weighted) undirected graph G. Binary search density. If g is current density, construct flow network: (S, u, m), $(u, T, m + 2g - d_u)$, $(u, v, 1)$, where m is a large constant (larger than sum of edge weights). Run floating-point max-flow. If minimum cut has empty S-component, then maximum density is smaller than g, otherwise it's larger. Distance between valid densities is at least $1/(n(n-1))$. Edge case when density is 0. This also works for weighted graphs by replacing d_u by the weighted degree, and doing more iterations (if weights are not integers).

7.1.9 Fecho de Peso Máximo

Dado um digrafo G com peso nos vértices. Transforme G numa rede de fluxo, colocando o peso de cada aresta como ∞. Adicione vértices S, T. Para cada vértice v de peso w, adicione uma aresta (S, v, w) se $w \geq 0$, ou a aresta $(v, T, -w)$ se $w < 0$. A soma de todos os pesos positivos menos o corte mínimo $c(S, T)$ é a resposta. Vértices que são alcançados a partir de S estão no fecho. O fecho de peso máximo é o mesmo que o complemento do fecho de peso mínimo num grafo com as arestas invertidas.

7.1.10 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.

7.1.11 Erdös-Gallai Theorem

Existe um grafo simples com graus $d_1 \geq d_2 \geq \ldots \geq d_n$ sse:

- $d_1 + d_2 + \ldots + d_n$ é par
- $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k) \quad \forall 1 \leq k \leq n.$

Para construir, ligamos 1 com $2, 3, \ldots, d_1 + 1$ e seguimos recursivamente.

7.1.12 Synchronizing word problem

A DFA has a synchronizing word (an input sequence that moves all states to the same state) iff. each pair of states has a synchronizing word. That can be checked using reverse DFS over pairs of states. Finding the shortest synchronizing word is NP-complete.

7.1.13 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.

7.1.14 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.

7.1.15 Dirac's theorem

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

7.1.16 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.

7.1.17 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
$q_{ij} = [i = j]\text{indeg}(i) - \#(i, j) \in E$.

7.1.18 Fatos úteis

O número de vértices de um grafo é igual a sua cobertura mínima mais a cardinalidade do conjunto independente máximo.

7.2 Euler walk

EulerWalk.h

Description: Eulerian undirected/directed path/cycle algorithm. 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, also put it->second in s (and then ret).
Time: $\mathcal{O}(E)$ where E is the number of edges. 17 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
```

7.3 Network flow

PushRelabel.h

Description: Push-relabel using the highest label selection rule and the gap heuristic. Quite fast in practice. To obtain the actual flow, look at positive values only.
Time: $\mathcal{O}\left(V^2\sqrt{E}\right)$ Better for dense graphs - Slower than Dinic (in practice) 47 lines

```
typedef lint Flow;

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 add_edge(int s, int t, Flow cap, Flow rcap=0) {
    if (s == t) return;
    g[s].push_back({t, g[t].size(), 0, cap});
    g[t].push_back({s, g[s].size(), 0, rcap});
  }
  void add_flow(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]) add_flow(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)
          add_flow(*cur[u], min(ec[u], cur[u]->c));
        else ++cur[u];
```

```
    }
  }
  bool leftOfMintCut(int a) { return H[a] >= g.size(); }
}; // hash-cpp-all = c4c114b51fa640b1ca9b9ad83a73ad56
```

Dinic.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 the actual flow, look at positive values only. 58 lines

```
template<typename T = lint>
struct Dinic {
  struct Edge {
    int to, rev; T c, f;
  };
  vector<int> lvl, ptr, q;
  vector<int> partition; //call findMinCut before use it
  vector<pair<pair<int,int>,int>> cut; //u,v,c
  vector<vector<Edge>> adj;
  Dinic(int n) : lvl(n), ptr(n), q(n), adj(n),partition(n),
     ↪cut(0) {}
  void addEdge(int a, int b, T c, int rcap = 0) {
    adj[a].push_back({b, adj[b].size(), c, 0});
    adj[b].push_back({a, adj[a].size() - 1, rcap, 0});
  }
  T dfs(int v, int t, T f) {
    if (v == t || !f) return f;
    for (int& i = ptr[v]; i < adj[v].size(); i++) {
      Edge& 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;
  }
  T maxflow(int s, int t) {
    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 &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, LLONG_MAX)) flow += p;
    } while (lvl[t]);
    return flow;
  }
  //only if you want the edges of the cut
  void dfsMC(int u){
    partition[u] = 1;
    for (Edge &e : adj[u])
      if (!partition[e.to])
        if (e.c - e.f == 0)
          cut.push_back({{u,e.to},e.c});
        else if (e.c - e.f > 0)
          dfsMC(e.to);
  }
  //only if you want the edges of the cut
  vector<pair<pair<int,int>,int>> findMinCut(int u,int t){
    maxflow(u,t); //DONT call again if you already called
       ↪it
    dfsMC(u);
    return cut;
  }
}; // hash-cpp-all = 00d2fea8e1f8098f86dde65c62d7131f
```

HLPP.h

Description: Highest label preflow relabel 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). 79 lines

```
template <int MAXN, class T = int> struct HLPP {
    const T INF = numeric_limits<T>::max();
    struct edge_t { int to, rev; T flow; };
    int s = MAXN - 1, t = MAXN - 2;
    vector<edge_t> adj[MAXN];
    vector<int> lst[MAXN], gap[MAXN];
    T excess[MAXN];
    int highest, height[MAXN], cnt[MAXN], work;
    void addEdge(int from, int to, int flow, bool
       ↪isDirected = true) {
        adj[from].push_back({to, adj[to].size(), flow});
        adj[to].push_back({from, adj[from].size() - 1,
           ↪isDirected ? 0 : flow});
    }
    void updHeight(int v, int nh) {
        work++;
        if (height[v] != MAXN) cnt[height[v]]--;
        height[v] = nh;
        if (nh == MAXN) return;
        cnt[nh]++, highest = nh;
        gap[nh].push_back(v);
        if (excess[v] > 0) lst[nh].push_back(v);
    }
    void globalRelabel() {
        work = 0;
        fill(height.begin(), height.end(), MAXN);
        fill(cnt.begin(), cnt.end(), 0);
        for (int i = 0; i < highest; i++)
            lst[i].clear(), gap[i].clear();
        height[t] = 0;
        queue<int> q({t});
        while (!q.empty()) {
            int v = q.front(); q.pop();
            for (auto &e : adj[v])
                if (height[e.to] == MAXN && adj[e.to][e.rev
                   ↪].flow > 0)
                    q.push(e.to), updHeight(e.to, height[v]
                       ↪ + 1);
            highest = height[v];
        }
    }
    void push(int v, edge_t &e) {
        if (excess[e.to] == 0)
            lst[height[e.to]].push_back(e.to);
        T df = min(excess[v], e.flow);
        e.flow -= df, adj[e.to][e.rev].flow += df;
        excess[v] -= df, excess[e.to] += df;
    }
    void discharge(int v) {
        int nh = MAXN;
        for (auto &e : adj[v]) {
            if (e.flow > 0) {
                if (height[v] == height[e.to] + 1) {
                    push(v, e);
                    if (excess[v] <= 0) return;
                }
                else nh = min(nh, height[e.to] + 1);
            }
        }
        if (cnt[height[v]] > 1) updHeight(v, nh);
        else {
            for (int i = height[v]; i <= highest; i++) {
                for (auto j : gap[i]) updHeight(j, MAXN);
                gap[i].clear();
            }
        }
    }
    T maxflow(int heur_n = MAXN) {
        fill(excess.begin(), excess.end(), 0);
        excess[s] = INF, excess[t] = -INF;
        globalRelabel();
        for (auto &e : adj[s]) push(s, e);
        for (; highest >= 0; highest--) {
            while (!lst[highest].empty()) {
                int v = lst[highest].back();
                lst[highest].pop_back();
                discharge(v);
                if (work > 4 * heur_n) globalRelabel();
            }
        }
        return excess[t] + INF;
    }
}; // hash-cpp-all = fa1d36a82b0ee3ea819a0f3cd2e1c3cb
```

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

```
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

```
// 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! 73 lines

```
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: $\mathcal{O}\left(V^3\right)$

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};
}
```

7.3.1 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.

7.4 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: $\mathcal{O}\left(\sqrt{V}E\right)$

42 lines

```
bool dfs(int a, int layer, const vector<vector<int>> &g,
    ↪vector<int> &btoa, vector<int> &A, vector<int> &B) {
    ↪// hash-cpp-1
  if (A[a] != layer) return 0;
  A[a] = -1;
  for(auto &b : g[a]) if (B[b] == layer + 1) {
    B[b] = -1;
    if (btoa[b] == -1 || dfs(btoa[b], layer+2, g, btoa, A,
        ↪B))
      return btoa[b] = a, 1;
  }
  return 0;
} // hash-cpp-1 = 1707b0c00c4eecb14a7d272f189c7330

int hopcroftKarp(const vector<vector<int>> &g, vector<int>
    ↪&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(), -1);
    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 += 2) {
      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] == -1) {
          B[b] = lay;
          next.push_back(btoa[b]);
        }
      }
      if (islast) break;
      if (next.empty()) return res;
      for(auto &a : next) A[a] = lay+1;
      cur.swap(next);
    }
    for(int a = 0; a < g.size(); ++a)
```

```
      res += dfs(a, 0, g, btoa, A, B)
  }
} // hash-cpp-2 = a6307328121207f4d652941106e00936
```

DFSMatching.h

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

22 lines

```
bool find(int j, vector<vector<int>>& g, vector<int>& btoa,
   ↪ vector<int>& seen) {
    if (btoa[j] == -1) return 1;
    seen[j] = 1; int di = btoa[j];
    for(auto &e : g[di])
        if (!seen[e] && find(e, g, btoa, seen)) {
            btoa[e] = di;
            return 1;
        }
    return 0;
}
int dfsMatching(vector<vector<int>>& g, vector<int>& btoa)
   ↪{
    vector<int> seen;
    for(int i = 0 i < g.size(); ++i) {
        seen.assign(btoa.size(), 0);
        for(auto &j : g[i])
            if (find(j, g, btoa, seen)) {
                btoa[j] = i;
                break;
            }
    }
    return btoa.size() - (int)count(btoa.begin(), btoa.end
       ↪(), -1);
} // hash-cpp-all = 454d41328791c911422c5e2abfdb25b0
```

WeightedMatching.h

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

75 lines

```
typedef vector<double> vd;
bool zero(double x) { return fabs(x) < 1e-10; }
double MinCostMatching(const vector<vd>& cost, vector<int>&
   ↪ L, vector<int>& R) {
  int n = cost.size(), mated = 0;
  vd dist(n), u(n), v(n);
  vector<int> dad(n), seen(n);

  for(int i = 0; i < n; ++i) {
    u[i] = cost[i][0];
    for(int j = 1; j < n; ++j) u[i] = min(u[i], cost[i][j])
       ↪;
  }
  for(int j = 0; j < n; ++j) {
    v[j] = cost[0][j] - u[0];
    for(int i = 1; i < n; ++i) v[j] = min(v[j], cost[i][j]
       ↪- u[i]);
  }

  L = R = vector<int>(n, -1);
  for(int i = 0; i < n; ++i) for(int j = 0; j < n; ++j) {
    if (R[j] != -1) continue;
    if (zero(cost[i][j] - u[i] - v[j])) {
      L[i] = j;
      R[j] = i;
      mated++;
      break;
    }
  }

  for (; mated < n; mated++) { // until solution is
     ↪feasible
    int s = 0;
    while (L[s] != -1) s++;
    fill(dad.begin(), dad.end(), -1);
    fill(seen.begin(), seen.end(), 0);
    for(int k = 0; k < n; ++k)
      dist[k] = cost[s][k] - u[s] - v[k];

    int j = 0;
    for (;;) {
      j = -1;
      for(int k = 0; k < n; ++k) {
        if (seen[k]) continue;
        if (j == -1 || dist[k] < dist[j]) j = k;
      }
      seen[j] = 1;
      int i = R[j];
      if (i == -1) break;
      for(int k = 0; k < n; ++k) {
        if (seen[k]) continue;
        auto new_dist = dist[j] + cost[i][k] - u[i] - v[k];
        if (dist[k] > new_dist) {
          dist[k] = new_dist;
          dad[k] = j;
        }
      }
    }

    for(int k = 0; k < n; ++k) {
      if (k == j || !seen[k]) continue;
      auto w = dist[k] - dist[j];
      v[k] += w, u[R[k]] -= w;
    }
    u[s] += dist[j];

    while (dad[j] >= 0) {
      int d = dad[j];
      R[j] = R[d];
      L[R[j]] = j;
      j = d;
    }
    R[j] = s;
    L[s] = j;
  }
  auto value = vd(1)[0];
    for(int i = 0; i < n; ++i) value += cost[i][L[i]];
  return value;
} // hash-cpp-all = 397d41cb6586b3fd523ec3c8ed48db8a
```

GeneralMatching.h

Description: Maximum Matching for general graphs (undirected and non bipartite) using Edmond's Blossom Algorithm.
Time: $\mathcal{O}(EV^2)$

70 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 find_match() {
        int ans = 0;
        // find random matching (not necessary, constant
           ↪improvement)
        vector<int> V(n-1); iota(V.begin(), V.end(), 1);
        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 = 7603b5274164025932e18a2a9a22ccc8
```

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.

"DFSMatching.h" 20 lines

```
vector<int> cover(vector<vector<int>>& g, int n, int m) {
    vector<int> match(m, -1);
    int res = dfsMatching(g, match);
    vector<bool> lfound(n, true), seen(m);
    for(int &it : match) if (it != -1) lfound[it] = false;
    vector<int> q, cover;
    for(int i = 0; i < n; ++i) if (lfound[i]) q.push_back(i
        ↪);
    while (!q.empty()) {
        int i = q.back(); q.pop_back();
        lfound[i] = 1;
        for(e, g[i]) if (!seen[e] && match[e] != -1) {
            seen[e] = true;
            q.push_back(match[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 = 99d5f60de2e305a84ef0397a263bd046
```

Koenig.cpp

Description: Given a bipartite graph G find a vertex set $S \subseteq U \cup V$ of minimum size that cover all edges.

37 lines

```
struct BipartiteVertexCover { // hash-cpp-1
  int nleft, nright;
  vector<bool> mark;
  Dinic din;
  BipartiteVertexCover(int nleft, int nright)
    : nleft(nleft), nright(nright), mark(1+nleft+nright+1)
    , din(1+nleft+nright+1, 0, 1+nleft+nright) {
    for (int l = 0; l < nleft; ++l) din.add_edge(0, 1+l, 1)
        ↪;
    for (int r = 0; r < nright; ++r) din.add_edge(1+nleft+r
        ↪, 1+nleft+nright, 1);
  }
  void add_edge(int l, int r) {
    din.add_edge(1+l, 1+nleft+r, 1);
  } // hash-cpp-1 = dd7c60a358106b1cde84313e37100a1f
  void dfs(int v) { // hash-cpp-2
    mark[v] = true;
    for (int edid : din.adj[v]) {
      Dinic::edge &ed = din.edges[edid];
      if (ed.flow < ed.cap && !mark[ed.u])
      dfs(ed.u);
    }
  } // hash-cpp-2 = 1d76f64fa31fc476fb5dce52eed5cfce
  vector<pair<int, int>> solve() { // hash-cpp-3
    int maxflow = din.maxflow();
    dfs(0);
    vector<pair<int, int>> result;
    for (int i = 0; i < (int)din.edges.size(); ++i) {
      Dinic::edge &ed = din.edges[i];
      int to = ed.u, from = din.edges[i^1].u;
      if (mark[from] && !mark[to] && ed.cap > 0) {
      if (from == 0) result.push_back({0, to-1});
      else result.push_back({1, from-1-nleft});
      }
    }
    assert(maxflow == result.size());
    return result;
  } // hash-cpp-3 = c7633b24b741d908236729782b5a555e
};
```

Hungarian.h

Description: finds min cost to complete n jobs w/ m workers each worker is assigned to at most one job (n <= m)

28 lines

```
int HungarianMatch(const vector<vector<int>> &a) { // cost
    ↪array, negative values are ok
    int n = a.size()-1, m = a[0].size()-1; // jobs 1..n,
        ↪workers 1..m
    vector<int> u(n+1), v(m+1), p(m+1); // p[j] -> job
        ↪picked by worker j
    for(int i = 1; i <= n; ++i) { // find alternating path
        ↪with job i
        p[0] = i; int j0 = 0;
        vector<int> dist(m+1, MOD), pre(m+1,-1); // dist,
            ↪previous vertex on shortest path
        vector<bool> done(m+1, false);
        do {
            done[j0] = true;
            int i0 = p[j0], j1; int delta = MOD;
            for(int j = 1; j <= m; ++j) if (!done[j]) {
                auto cur = a[i0][j]-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) // just dijkstra
                ↪with potentials
                if (done[j]) u[p[j]] += delta, v[j] -=
                    ↪delta;
                else dist[j] -= delta;
            j0 = j1;
        } while (p[j0]);
        do { // update values on alternating path
            int j1 = pre[j0];
            p[j0] = p[j1];
            j0 = j1;
        } while (j0);
    }
    return -v[0]; // min cost
} // hash-cpp-all = 52548198c0a8663ab7433602263f7ea0
```

7.5 DFS algorithms

CentroidDecomposition.cpp

Description: Divide and Conquer on Trees.

33 lines

```
struct centroid_t {
    vector<bool> mark;
    vector<int> subtree, level, par_tree, closest;
    vector<vector<int>> edges, dist, parent;
    centroid_t(vector<vector<int>> &e, int n) : mark(n, 0),
        ↪ subtree(n), level(n), par_tree(n), closest(n,
        ↪INT_MAX/2), dist(n, vector<int>(20)), parent(n,
        ↪vector<int>(20)) { edges = e; build(0, -1); update
        ↪(0); }
    void dfs(int v, int par, int parc, int lvl) {
        subtree[v] = 1;
        parent[v][lvl] = parc;
        dist[v][lvl] = 1 + dist[par][lvl];
        for (int u : edges[v]) {
            if (u == par) continue;
            if (!mark[u]) {
                dfs(u, v, parc, lvl);
                subtree[v] += subtree[u];
            }
        }
    }
    int get_centroid(int v, int par, int sz) {
        for (int u : edges[v])
            if (!mark[u] && u != par && subtree[u] > sz/2)
                return get_centroid(u, v, sz);
        return v;
    }
    void build(int v, int p, int lvl = 0) {
        dfs(v, v, p, lvl);
        int x = get_centroid(v, v, subtree[v]);
        mark[x] = 1;
        par_tree[x] = p;
        level[x] = 1 + lvl;
        for (int u : edges[x])
            if (!mark[u]) build(u, x, 1 + lvl);
    }
}; // hash-cpp-all = ab9c35403e7336205ff6e8701fab04c7
```

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

29 lines

```
struct tarjan_t {
    int n, ncnt = 0, time = 0;
    vector<vector<int>> edges;
    vector<int> preorder_of, cnt_of, order;
    stack<int> stack_t;
    tarjan_t(int n): n(n), edges(n), preorder_of(n, 0),
        ↪cnt_of(n, -1) {}
    int dfs(int u) { // hash-cpp-1
        int reach = preorder_of[u] = ++time, v;
        stack_t.push(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.top();
                stack_t.pop();
                order.push_back(v);
                cnt_of[v] = ncnt;
            } while (v != u);
            ++ncnt;
        }
```

```
        return preorder_of[u] = reach;
    } // hash-cpp-1 = 93105086c30ffe6a8c80938302c04fdf
    void solve() {
        time = ncnt = 0;
        for (int i = 0; i < (int)edges.size(); ++i)
            if (cnt_of[i] == -1) dfs(i);
    }
};
```

Kosaraju.h

Description: Find the strongly connected components of a digraph

30 lines

```
struct kosaraju_t {
    int time = 1, n;
    vector<vector<int>> adj, tree;
    vector<bool> vis;
    vector<int> color, s;
    kosaraju_t(int _n) : n(_n), adj(n), tree(n), color(n,
      ↪-1), vis(n, false) {}
    void dfs(int u) {
        vis[u] = true;
        for (int v : adj[u]) if (!vis[v]) dfs(v);
        s.emplace_back(u);
    }
    int e;
    void dfs2(int u, int delta) {
        color[u] = delta;
        for (int v : tree[u])
            if (color[v] == -1) dfs2(v, delta);
    }
    void solve() {
        for (int i = 0; i < n; ++i)
            if (!vis[i]) dfs(i);
        e = 0;
        reverse(s.begin(), s.end());
        for (int i : s) {
            if (color[i] == -1) {
                ++e;
                dfs2(i,i);
            }
        }
    }
}; // hash-cpp-all = ee9c96cdf2fab9563ce12f868663f3e2
```

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

46 lines

```
typedef vector<int> vi;
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>> bicomps() {
  // returns components and its edges ids
  vector<vector<int>> comps;
  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 = 3e7f07e94a887065fdfa6d0cdc978102
```

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.

51 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^1].push_back(j);
    gr[j^1].push_back(f);
  } // hash-cpp-1 = 1140d4116e06cfd5efce120090e3f131
  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);
    for (auto e : gr[i]) if (!comp[e])
      low = min(low, val[e] ?: dfs(e));
    ++time;
    if (low == val[i]) do {
      x = z.back(); z.pop_back();
      comp[x] = time;
      if (values[x>>1] == -1)
        values[x>>1] = !(x&1);
    } while (x != i);
    return val[i] = low;
  } // hash-cpp-3 = 9daa11ba272442daba9b26ba87433109
  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
```

7.6 Heuristics

MaximalCliques.h

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

<bits/stdc++.h> 62 lines

```
#define P 1000000007
#define N 110000

int n, m;
vector <int> go[N], lk[N];

int w[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 deg[N], pos[N], id[N];

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);
}
```

7.7 Trees

Tree.h

Description: Structure that handles tree's, can find its diameter points, diameter length, center vertices, etc;

42 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
```

TreePower.h

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

25 lines

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

int jmp(vector<vector<int>>& tbl, int nod, int steps){
  for(int i = 0; i < tbl.size(); ++i)
    if(steps&(1<<i)) nod = tbl[i][nod];
  return nod;
}

int lca(vector<vector<int>>& tbl, vector<int>& depth, int a
   ↪, int b) {
  if (depth[a] < depth[b]) swap(a, b);
  a = jmp(tbl, a, depth[a] - depth[b]);
  if (a == b) return a;
  for (int i = sz(tbl); i--;) {
    int c = tbl[i][a], d = tbl[i][b];
    if (c != d) a = c, b = d;
  }
  return tbl[0][a];
} // hash-cpp-all = b0614027f8c8b0d0f9c143eced296cb7
```

LCA.cpp

Description: Data structure for computing lowest common ancestors in a tree (with 0 as root). Can also find the distance between two nodes.

47 lines

```
struct lca_t {
    int logn, preorderpos;
    vector<int> invpreorder, height;
    vector<vector<int>> edges;
    vector<vector<int>> jump_binary;
    lca_t(int n, vector<vector<int>>& adj) : height(n),
        ↪invpreorder(n) { // hash-cpp-1
        while((1 << (logn+1)) <= n) ++logn;
        jump_binary.assign(n, vector<int>(logn, 0));
        edges = adj;
        dfs(0, -1, 0);

    } // hash-cpp-1 = 8e31d66d91c9b0271cd7bc82dae601cc
    void dfs(int v, int p, int h) { // hash-cpp-2
        invpreorder[v] = preorderpos++;
        height[v] = h;
        jump_binary[v][0] = (p == -1) ? 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);
        }
    } // hash-cpp-2 = eb1a9e7b68a33e85c80534f07f495ee8
    int climb(int v, int dist) { // hash-cpp-3
        for (int l = 0; l <= logn; ++l)
            if (dist & (1 << l)) v = jump_binary[v][l];
        return v;
    } // hash-cpp-3 = 23190810d4f0892a71472f3ef4ab5907
    int query(int a, int b) { // hash-cpp-4
        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];
    } // hash-cpp-4 = f24a4f62362deb2de108cb3a94d38be0
    int dist(int a, int b) {
        return height[a] + height[b] - 2 * height[query(a,b
            ↪)];
    }
    bool is_parent(int p, int v) { // hash-cpp-5
        if (height[p] > height[v]) return false;
        return p == climb(v, height[v] - height[p]);
    } // hash-cpp-5 = efc0ddfe873dcad0f02b137ccb9b432b
};
```

LCA.h

Description: Data structure for computing lowest common ancestors in a tree (with 0 as root). C should be an adjacency list of the tree, either directed or undirected. Can also find the distance between two nodes.
Usage: lca_t lca(undirGraph);
lca.query(firstNode, secondNode);
lca.dist(firstNode, secondNode);
Time: $\mathcal{O}(N \log N + Q)$

46 lines

```
template<class T>
struct RMQ {
  vector<vector<T>> jmp;
  RMQ(const vector<T>& V) {
    int N = V.size(), on = 1, depth = 1;
    while (on < N) on *= 2, depth++;
    jmp.assign(depth, V);
    for(int i = 0; i < depth-1; ++i) for(int j = 0; j < N;
        ↪++j)
      jmp[i+1][j] = min(jmp[i][j],
      jmp[i][min(N - 1, j + (1 << i))]);
  }
  T query(int a, int b) {
    assert(a < b); // or return inf if a == b
    int dep = 31 - __builtin_clz(b - a);
    return min(jmp[dep][a], jmp[dep][b - (1 << dep)]);
  }
};

struct lca_t {
  int n;
  vector<int> depth, order;
  vector<vector<int>> edges;
  vector<pair<int,int>> temp;
  RMQ<pair<int,int>> rmq;
  lca_t(vector<vector<int>>& g) : n(g.size()),
  edges(g), depth(n), order(n), rmq(dfs(0,-1)) {}
  vector<pair<int,int>> dfs(int v, int p) {
    order[v] = temp.size();
    depth[v] = 1 + depth[p];
    temp.push_back({depth[v], v});
    for (int u : edges[v]) {
      if (u == p) continue;
      dfs(u, v);
      temp.push_back({depth[v], v});
    }
    return temp;
  }
  int query(int a, int b) {
    a = order[a]; b = order[b];
    if (a > b) swap(a, b);
    return rmq.query(a, b).second;
  }
  int dist(int a, int b) {
    return depth[a] + depth[b] - 2*depth[query(a, b)];
  }
}; // hash-cpp-all = 4897fe0ab4353cd05392511138d3759f
```

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.height.size());
  vector<int> li = subset, &T = lca.invpreorder;
  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.query(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.query(a, b)], b);
  }
  return ret;
} // hash-cpp-all = 4f28d7f851dd0cb96e0b9e9538bcc079
```

Tree-Isomorphism.h

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

52 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> 75 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>> 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 = 96ccfd04e4ec32ca1a67d9f1044fbe61
```

MatrixTree.h

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

<ModTemplate.h> 16 lines

```
// Need to be tested, has some bug for sure
constexpr int d = 3; // dimension of square matrix
num get(Matrix<num, d> &M) {
  Matrix<num, d> result;
  for (int i = 0; i < n; ++i)
    for (int j = i+1; j < n; ++j) {
      num ed = M.d[i][j];
      result.d[i][i] = result.d[i][i] + ed;
      if (j != n-1) {
        result.d[j][j] = result.d[j][j] + ed;
        result.d[i][j] = result[i][j] - ed;
        result.d[j][i] = result[j][i] - ed;
      }
    }
    return det(result.d);
} // hash-cpp-all = 001a4da570fe37697acab312f4a63adc
```

7.8 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

53 lines

```
vector<int> deg, subtree, order, par, parincycles, idxcycle
  ↪, sz, st, depth, cycles[MAXN];
vector<bool> mark, incycle,
int numcycle;

void bfs() {
    queue<int> q;
    for (int i = 0; i < n; ++i)
      if (!indeg[i]) {
        q.push(i);
        mark[i] = 1;
      }
    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);
        mark[curpar] = 1;
      }
    }
    numcycles = 0;
    for (int i = 0; i < n; ++i)
      if (!mark[i]) find_cycle(i);
    for (int i = order.size()-1; i >= 0; --i) {
      int v = order[i], curpar = par[v];
      parincycle[v] = parincycle[curpar];
      cycle[v] = cycle[curpar];
      incycle[v] = 0;
      idxcycle[v] = -1;
      depth[v] = 1 + depth[curpar];
    }
}
void find_cycle(int u) {
    int idx = ++numcycle, cur = 0, par = u;
    st[idx] = u;
    size[idx] = 0;
    cycles[idx].clear();
    while(!mark[u]) {
      mark[u] = incycle[u] = 1;
```

```
      parincycle[u] = u;
      cycle[u] = idx;
      idxcycle[u] = cur;
      cycles[idx].push_back(u);
      ++size[idx];
      depth[u] = 0;
      ++subtree[u];
      u = par[u];
      ++cur;
    }
} // hash-cpp-all = 6d0efde2516c011a17d627688e936dfd
```

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

60 lines

```
vector<int> deg, subtree, order, par, parincycles, idxcycle
    ↪, sz, st, depth, cycles[MAXN];
vector<bool> mark, incycle,

void bfs() {
    queue<int> q;
    for (int i = 0; i < n; ++i)
      if (deg[i] == 1) {
        q.push(i);
        mark[i] = 1;
      }
    while(!q.empty()) {
      int v = q.front(); q.pop();
      order.push_back(v);
      ++subtree[v];
      int curpar = find_par(v);
      par[v] = curpar;
      deg[curpar]--;
      subtree[curpar] += subtree[v];
      if (deg[curpar] == 1) {
        q.push(curpar);
        mark[curpar] = 1;
      }
    }
    numcycles = 0;
    for (int i = 0; i < n; ++i)
      if (!mark[i]) find_cycle(i);
    for (int i = order.sz()-1; i >= 0; --i) {
      int v = order[i], curpar = par[v];
      parincycle[v] = parincycle[curpar];
      cycle[v] = cycle[curpar];
      incycle[v] = 0;
      idxcycle[v] = -1;
      depth[v] = 1 + depth[curpar];
    }
}
void find_cycle(int u) {
    int idx = ++numcycle, cur = 0, par = u;
    st[idx] = u;
    sz[idx] = 0;
    cycles[idx].clear();
    while(!mark[u]) {
      mark[u] = incycle[u] = 1;
      par[u] = find_par(u);
      if (par[u] == -1) par[u] = par;
      parincycle[u] = u;
      cycle[u] = idx;
      idxcycle[u] = cur;
      cycles[idx].push_back(u);
      ++sz[idx];
      depth[u] = 0;
      ++subtree[u];
      u = par[u];
      ++cur;
    }
}
int find_par(int u) {
    for (int v : graph[u])
      if (!mark[v]) return v;
    return -1;
} // hash-cpp-all = 7202d56d5cb33ca2bff55481531b9c4c
```

7.9 Other

kthShortestPath.h

Description: Find Kth shortest path from s to t.
Time: $\mathcal{O}\left((V+E)lg(V)*k\right)$

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
```

Hamiltonian.h

Description: Find if exist an hamiltonian path
Time: $\mathcal{O}\left(2^n n^2\right)$

17 lines

```
bool hamiltonian(vector<vector<int>> &edges, int n) {
  array<array<bool, MAXN>, MAXN> dp;
  for (int i = 0; i < n; ++i) dp[i][1<<i] = 1;
  for (int i = 0; i < (1 << n); ++i) {
    for (int j = 0; j < n; ++j)
      if (i & (1 << j)) {
        for (int k = 0; k < n; ++k)
          if (i & (1 << k) && edges[k][j] && k != j && dp[k
            ↪][i^(1<<j)]) {
            dp[i][j] = 1;
            break;
          }
      }
  }
  for (int i = 0; i < n; ++i)
    if (dp[i][(1 << n)-1]) return 1;
  return 0;
} // hash-cpp-all = 25ead8823473df3c1c90cc487b54ba8c
```

Boruvka.h

31 lines

```
struct Edge {
  int u, v, w, id;
  Edge() {};
  Edge(int u, int v, int w = 0, int id = 0) : u(u), v(v), w
    ↪(w), id(id) {};
  bool operator<(Edge &o) const { return w < other.w; };
};

vector<Edge> Boruvka(vector<Edge> &edges, int n) {
  vector<Edge> mst, best(n);
  UF dsu(n);
  int f = 1;
  while (f) {
    f = 0;
    for (int i = 0; i < n; ++i) best[i] = Edge(i, i, INF);
    for (Edge e : edges) {
      int pu = dsu.find(e.u), pv = dsu.find(e.v);
      if (pu == pv) continue;
      if (e < best[pu]) best[pu] = e;
      if (e < best[pv]) best[pv] = e;
    }
    for (int i = 0; i < n; ++i) {
      Edge e = best[dsu.find(i)];
      if (e.w != INF) {
        dsu.unite(e.u, e.v);
        mst.push_back(e);
        f = 1;
      }
    }
  }
  return mst;
} // hash-cpp-all = a175a34b938e72edda901cebc98d864f
```

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
```

SteinerTree.h

Description: Find the cost of the smallest tree containing all elements of terminal ts for a non-negative undirected graph
Time: $\mathcal{O}\left(3^t n + 2^t n^2 + n^3\right)$

25 lines

```
//TODO: Check what is a terminal...

int Steiner(vector<vector<int>> &g, vector<int> &ts) {
    int n = g.size(), m = ts.size();
    if (m < 2) return 0;
    vector<vector<int>> dp(1<<m, vector<int>(n));
    for(int k = 0; k < n; ++k)
        for(int i = 0; i < n; ++i)
            for(int j = 0; j < n; ++j)
                g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
    for(int i = 0; i < m; ++i)
        for(int j = 0; j < n; ++j)
            dp[1<<i][j] = g[ts[i]][j];
    for (int i = 1; i < (1<<m); ++i) if (((i-1)&i) != 0) {
        for (int j = 0; j < n; ++j) {
            dp[i][j] = INF;
            for (int k = (i-1)&i; k > 0; k = (k-1)&i)
                dp[i][j] = min(dp[i][j], dp[k][j] + dp[i^k
                    ↪][j]);
        }
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                dp[i][j] = min(dp[i][j], dp[i][k] + g[k][j
                    ↪]);
    }
    return dp[(1<<m)-1][ts[0]];
} // hash-cpp-all = 3bb8ba31a1df9c80e44832d553fbf877
```

Pruefer.cpp

Description: Given a tree, construct its pruefer sequence

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
                ↪= i;
        }
        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 $\mathcal{O}(n)$, otherwise $\mathcal{O}(nlog(n))$

15 lines

```
bool EG(vector<int> &deg) {
    sort(deg.begin(), deg.end(), greater<int>());
    int n = deg.size(), p = n+1;
    vector<lint> dp(n);
    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 && dp[p] < k) p--;
        lint sum;
        if (p >= k-1) sum = (p-k+1)*k + dp[n-1] - dp[p];
        else sum = dp[n-1] - dp[k-1];
        if (dp[k-1] > k*(k-1) + sum) return false;
    }
    return dp[n-1] % 2 == 0;
} // hash-cpp-all = d8eb1926923a07a2fdc88d0ab93b1fe0
```

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.

48 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);
  }
}; // hash-cpp-all = b27b0c0eeabb94e7f648f63f003a6867
```

Directed-MST.cpp

Description: Finds the minimum spanning arborescence from the root. (any more notes?)

70 lines

```
#define N 110000
#define M 110000
#define inf 2000000000

struct edg {
    int u, v;
    int cost;
} E[M], E_copy[M];

int In[N], ID[N], vis[N], pre[N];

// edges pointed from root.
int Directed_MST(int root, int NV, int NE) {
  for (int i = 0; i < NE; i++)
    E_copy[i] = E[i];
    int ret = 0;
    int u, v;
    while (true) { // hash-cpp-1
        for (int i = 0; i < NV; ++i) In[i] = inf;
        for (int i = 0; i < NE; ++i) {
            u = E_copy[i].u;
            v = E_copy[i].v;
            if(E_copy[i].cost < In[v] && u != v) {
                In[v] = E_copy[i].cost;
                pre[v] = u;
            }
        }
        for (int i = 0; i < NV; ++i) {
            if(i == root)   continue;
            if(In[i] == inf)    return -1; // no solution
        }

        int cnt = 0;
        for (int i = 0; i < NV; ++i) {
          ID[i] = -1;
```

```
        vis[i] = -1;
    }
    In[root] = 0;

    for (int i = 0; i < NV; ++i) {
        ret += In[i];
        int v = i;
        while(vis[v] != i && ID[v] == -1 && v != root)
          ↪{
            vis[v] = i;
            v = pre[v];
        }
        if(v != root && ID[v] == -1) {
            for(u = pre[v]; u != v; u = pre[u]) {
                ID[u] = cnt;
            }
            ID[v] = cnt++;
        }
    }
    if(cnt == 0)    break;
    for (int i = 0; i < NV; ++i) {
        if(ID[i] == -1) ID[i] = cnt++;
    }
    for (int i = 0; i < NE; ++i) {
        v = E_copy[i].v;
        E_copy[i].u = ID[E_copy[i].u];
        E_copy[i].v = ID[E_copy[i].v];
        if(E_copy[i].u != E_copy[i].v) {
            E_copy[i].cost -= In[v];
        }
    }
    NV = cnt;
    root = ID[root];
  }
  return ret;
} // hash-cpp-1 = 791af8a003d5dd799db879a7c0ef9aec
```

Graph-Dominator-Tree.cpp

Description: Dominator Tree.

107 lines

```
#define N 110000 //max number of vertices

vector<int> succ[N], prod[N], bucket[N], dom_t[N];
int semi[N], anc[N], idom[N], best[N], fa[N], tmp_idom[N];
int dfn[N], redfn[N];
int child[N], size[N];
int timestamp;

void dfs(int now) { // hash-cpp-1
  dfn[now] = ++timestamp;
  redfn[timestamp] = now;
  anc[timestamp] = idom[timestamp] = child[timestamp] =
    ↪size[timestamp] = 0;
  semi[timestamp] = best[timestamp] = timestamp;
  int sz = succ[now].size();
  for(int i = 0; i < sz; ++i) {
    if(dfn[succ[now][i]] == -1) {
      dfs(succ[now][i]);
      fa[dfn[succ[now][i]]] = dfn[now];
    }
    prod[dfn[succ[now][i]]].push_back(dfn[now]);
  }
} // hash-cpp-1 = 6412bfd6a0d21b66ddaa51ea79cbe7bd

void compress(int now) { // hash-cpp-2
  if(anc[anc[now]] != 0) {
    compress(anc[now]);
    if(semi[best[now]] > semi[best[anc[now]]])
      best[now] = best[anc[now]];
    anc[now] = anc[anc[now]];
  }
} // hash-cpp-2 = 1c9444eb3f768b7af8741fafbf3afb5a

inline int eval(int now) { // hash-cpp-3
  if(anc[now] == 0)
    return now;
  else {
    compress(now);
    return semi[best[anc[now]]] >= semi[best[now]] ? best[
      ↪now]
    : best[anc[now]];
  }
} // hash-cpp-3 = 4e235f39666315b46dcd3455d5f866d1

inline void link(int v, int w) { // hash-cpp-4
  int s = w;
  while(semi[best[w]] < semi[best[child[w]]]) {
    if(size[s] + size[child[child[s]]] >= 2*size[child[s]])
      ↪ {
      anc[child[s]] = s;
      child[s] = child[child[s]];
    } else {
      size[child[s]] = size[s];
      s = anc[s] = child[s];
    }
  }
  best[s] = best[w];
  size[v] += size[w];
  if(size[v] < 2*size[w])
    swap(s, child[v]);
  while(s != 0) {
    anc[s] = v;
    s = child[s];
  }
} // hash-cpp-4 = 270548fd021351ae21e97878f367b6f9

// idom[n] and other vertices that cannot be reached from n
  ↪ will be 0
void lengauer_tarjan(int n) { // n is the root's number //
  ↪hash-cpp-5
  memset(dfn, -1, sizeof dfn);
  memset(fa, -1, sizeof fa);
  timestamp = 0;
  dfs(n);
  fa[1] = 0;
  for(int w = timestamp; w > 1; --w) {
    int sz = prod[w].size();
    for(int i = 0; i < sz; ++i) {
      int u = eval(prod[w][i]);
      if(semi[w] > semi[u])
        semi[w] = semi[u];
    }
    bucket[semi[w]].push_back(w);
    //anc[w] = fa[w]; link operation for o(mlogm) version
              link(fa[w], w);
    if(fa[w] == 0)
      continue;
    sz = bucket[fa[w]].size();
    for(int i = 0; i < sz; ++i) {
      int u = eval(bucket[fa[w]][i]);
      if(semi[u] < fa[w])
        idom[bucket[fa[w]][i]] = u;
      else
        idom[bucket[fa[w]][i]] = fa[w];
    }
    bucket[fa[w]].clear();
  }
  for(int w = 2; w <= timestamp; ++w) {
    if(idom[w] != semi[w])
      idom[w] = idom[idom[w]];
  }
  idom[1] = 0;
  for(int i = timestamp; i > 1; --i) {
    if(fa[i] == -1)
      continue;
    dom_t[idom[i]].push_back(i);
  }
  memset(tmp_idom, 0, sizeof tmp_idom);
  for (int i = 1; i <= timestamp; i++)
    tmp_idom[redfn[i]] = redfn[idom[i]];
  memcpy(idom, tmp_idom, sizeof idom);
} // hash-cpp-5 = f49c40461d92222d8d39b28b0de66828
```

Graph-Negative-Cycle.cpp

Description: negative cycle

33 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 = ec5cf9bc61e058959ce8649f1e707b1b
```

TransitiveClosure.h

Description: Given a directed graph adjacency matrix, computes closure, where $closure[i][j] = 1$ if there is a path from i to j in the graph. Closure is computed in $O(N^3/64)$ due to bitset. Also supports adding an edge to the graph and updating the closure accordingly in $O(N^2/64)$.

20 lines

```
template<int sz>
struct TC {
  vector<bitset<sz>> closure;
  TC(vector<vector<int>> adj) : closure(sz) {
    for(int i = 0; i < sz; ++i)
      for(int j = 0; j < sz; ++j)
```

```
      closure[i][j] = adj[i][j];
  for(int i = 0; i < sz; ++i)
    for(int j = 0; j < sz; ++j)
      if (closure[j][i])
        closure[j] |= closure[i];
}
void addEdge(int a, int b) {
  if (closure[a][b]) return;
  closure[a].set(b);
  closure[a] |= closure[b];
  for (int i = 0; i < sz; ++i)
    if (closure[i][a]) closure[i] |= closure[a];
}
}; // hash-cpp-all = eb5414544d683fe95d450ad4d8e805a0
```

Geometry (8)

8.1 Geometric primitives

Point.h

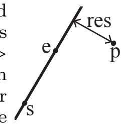
Description: Class to handle points in the plane. T can be e.g. double or long long. (Avoid int.)

26 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)); }
}; // hash-cpp-all = 4d90b59b170ae98f49395e2d118bddd9
```

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.

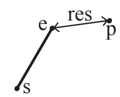


"Point.h" 4 lines

```
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" 6 lines

```
typedef Point<double> P;
double segDist(P& s, P& e, P& p) {
  if (s==e) return (p-s).dist();
  auto d = (e-s).dist2(), t = min(d,max(.0,(p-s).dot(e-s)))
     ↪;
  return ((p-s)*d-(e-s)*t).dist()/d;
} // hash-cpp-all = 5c88f46fb14a05a4f47bbd23b8a9c427
```

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 ⇔ 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

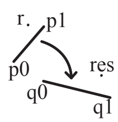
Description: Returns true iff p lies on the line segment from s to e. Use `(segDist(s,e,p)<=epsilon)` instead when using Point<double>.

"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`

37 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 quad() const {
    assert(x || y);
    if (y < 0) return (x >= 0) + 2;
    if (y > 0) return (x <= 0);
    return (x <= 0) * 2;
  }
  Angle t90() const { return {-y, x, t + (quad() == 3)}; }
  Angle t180() const { return {-x, -y, t + (quad() >= 2)}; }
  Angle t360() const { return {x, y, t + 1}; }
};
bool operator<(Angle a, Angle b) {
  // add a.dist2() and b.dist2() to also compare distances
  return make_tuple(a.t, a.quad(), a.y * (ll)b.x) <
         make_tuple(b.t, b.quad(), 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 = 1856c5d371c2f8f342a22615fa92cd54
```

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
```

Complex.h

Description: Exemple of geometry using complex numbers. Just to be used as reference. std::complex has issues with integral data types, be careful, you can't use polar or abs.

83 lines

```
const double E = 1e-9;
typedef double T;
typedef complex<T> pt;
#define x real()
#define y imag()
// example of how to represent a line using complex numbers
struct line {
  pt p, v;
  line(pt a, pt b) {
    p = a;
    v = b - a;
  }
};

pt translate(pt v, pt p) {return p + v;}
//rotate point around origin by a
pt rotate(pt p, T a) { return p * polar(1.0, a); }
//around pivot
pt rotate(pt v, T a, pt pivot) { (a-pivot) * polar(1.0, a) + pivot; }
T dot(pt v, pt w) { return (conj(v)*w).x; }
T cross(pt v, pt w) { return (conj(v)*w).y; }
T cross(pt A, pt B, pt C) {
  return cross(B - A, C - A);
}
pt proj(pt a, pt v) {
  return v * dot(a, v) / dot(v, v);
}

pt closest(pt p, line l) {
  return l.p + proj(p - l.p, l.v);
}

double dist(pt p, line l) {
  return fabs(p - closest(p, l));
}

pt proj(pt p, line l) {
    return
}

pt reflect(pt p, pt v, pt w) {
    pt z = p - v; pt q = w - v;
    return conj(z/q) * q + v;
}

pt intersection(line a, line b) { // undefined if parallel
    T d1 = cross(b.p - a.p, a.v - a.p);
    T d2 = cross(b.v - a.p, a.v - a.p);
    return (d1 * b.v - d2 * b.p)/(d1 - d2);
}

vector<pt> convex_hull(vector<pt> points) {
    if (points.size() <= 1) return points;
  sort(points.begin(), points.end(), [](pt a, pt b) {
    return real(a) == real(b) ? imag(a) < imag(b) : real(a) < real(b); });
  vector<pt> hull(points.size()+1);
  int s = 0, k = 0;
  for (int it = 2; it--; s = --k, reverse(points.begin(), points.end()))
      for (pt p : points) {
          while(k >= s+2 && cross(hull[k-2], hull[k-1], p) <= 0) k--;
          hull[k++] = p;
      }
  return {hull.begin(), hull.begin() + k - (k == 2 && hull[0] == hull[1])};
}
```

```
pt p{4, 3};
// get the absolute value and angle in [-pi, pi]
cout << abs(p) << ' ' << arg(p) << '\n'; // 5 - 0.643501

// make a point in polar form
cout << polar(2.0, -M_PI/2) << '\n'; // (1.41421, -1.41421)
pt v{1, 0};
cout << rotate(v, -M_PI/2) << '\n';
// Projection of v onto Riemann sphere and norm of p
cout << proj(v) << ' ' << norm(p) << '\n';
// Distance between p and v and the squared distance
cout << abs(v-p) << ' ' << norm(v-p) << '\n';
// Angle of elevation of line vp and its slope
cout << arg(p-v) * (180/M_PI) << ' ' << tan(arg(p-v)) << '\n';

// has trigonometric functions aswell (e.g. cos, sin, cosh, sinh, tan, tanh)
// and exp, pow, log
// hash-cpp-all = 2446aedc8bcd593691c082f59fae7479
```

LinearSolver.h

8.2 Circles

CircleIntersection.h

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

"Point.h" 14 lines

```
typedef Point<double> P;
bool circleIntersection(P a, P b, double r1, double r2,
    pair<P, P>* out) {
  P delta = b - a;
  assert(delta.x || delta.y || r1 != r2);
  if (!delta.x && !delta.y) return false;
  double r = r1 + r2, d2 = delta.dist2();
  double p = (d2 + r1*r1 - r2*r2) / (2.0 * d2);
  double h2 = r1*r1 - p*p*d2;
  if (d2 > r*r || h2 < 0) return false;
  P mid = a + delta*p, per = delta.perp() * sqrt(h2 / d2);
  *out = {mid + per, mid - per};
  return true;
} // hash-cpp-all = 828fbb1fff1469ed43b2284c8e07a06c
```

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.

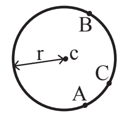
"Point.h" 13 lines

```
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" 9 lines

```
typedef Point<double> P;
double ccRadius(const P& A, const P& B, const P& C) {
  return (B-A).dist()*(C-B).dist()*(A-C).dist()/
      abs((B-A).cross(C-A))/2;
}
P ccCenter(const P& A, const P& B, const P& C) {
  P b = C-A, c = B-A;
  return A + (b*c.dist2()-c*b.dist2()).perp()/b.cross(c)/2;
} // hash-cpp-all = 1caa3aea364671cb961900d4811f0282
```

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 101 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 = 024a9290d20aec57c286f84dd8b35701
```

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
```

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" 11 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;
    //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
```

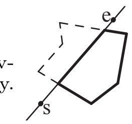
PolygonCenter.h

Description: Returns the center of mass for a polygon.

"Point.h" 10 lines

```
typedef Point<double> P;
Point<double> polygonCenter(vector<P>& v) {
  auto i = v.begin(), end = v.end(), j = end-1;
  Point<double> res{0,0}; double A = 0;
  for (; i != end; j=i++) {
    res = res + (*i + *j) * j->cross(*i);
    A += j->cross(*i);
  }
  return res / A / 3;
} // hash-cpp-all = d210bd2372832f7d074894d904e548ab
```

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" 15 lines

```
typedef Point<double> P;
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.emplace_back();
      lineIntersection(s, e, cur, prev, res.back());
    }
    if (side)
      res.push_back(cur);
  }
  return res;
} // hash-cpp-all = 9494eaafe7195a30491957f5e29de37c
```

ConvexHull.h

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

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

"Point.h" 13 lines

```
typedef Point<lint> P;
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 = 1dda3bbc9ea7ae391330b8cb8a97675a
```

HullDiameter.h

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

12 lines

```
typedef Point<lint> P;
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 = 5d3363d31e941a4a0356469882ea89e1
```

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" 14 lines

```
typedef Point<lint> P;

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 = 13f9135bdca0b3cc782ea80b806ee99e
```

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" 34 lines

```
typedef Point<double> P;
double rat(P a, P b) { return sgn(b.x) ? a.x/b.x : a.y/b.y;
  ↪ }
double polyUnion(vector<vector<P>>& poly) {
  double ret = 0;
  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 = 7792a4559206ac7061afe751d69dcc24
```

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

"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" 70 lines

```
#define eps 1e-8
typedef Point<double> P;

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

"Point.h" 58 lines

```
template<class It>
bool it_less(const It& i, const It& j) { return *i < *j; }
template<class It>
bool y_it_less(const It& i,const It& j) {return i->y < j->y
   ↪;}

template<class It, class IIt> /* IIt = vector<It>::iterator
   ↪ */
double cp_sub(IIt ya, IIt yaend, IIt xa, It &i1, It &i2) {
  typedef typename iterator_traits<It>::value_type P;
  int n = yaend-ya, split = n/2;
  if(n <= 3) { // base case
    double a = (*xa[1]-*xa[0]).dist(), b = 1e50, c = 1e50;
    if(n==3) b=(*xa[2]-*xa[0]).dist(), c=(*xa[2]-*xa[1]).
       ↪dist();
    if(a <= b) { i1 = xa[1];
      if(a <= c) return i2 = xa[0], a;
      else return i2 = xa[2], c;
    } else { i1 = xa[2];
      if(b <= c) return i2 = xa[0], b;
      else return i2 = xa[1], c;
  } }
  vector<It> ly, ry, stripy;
  P splitp = *xa[split];
  double splitx = splitp.x;
  for(IIt i = ya; i != yaend; ++i) { // Divide
    if(*i != xa[split] && (**i-splitp).dist2() < 1e-12)
      return i1 = *i, i2 = xa[split], 0;// nasty special
         ↪case!
    if (**i < splitp) ly.push_back(*i);
    else ry.push_back(*i);
  } // assert((signed)lefty.size() == split)
  It j1, j2; // Conquer
  double a = cp_sub(ly.begin(), ly.end(), xa, i1, i2);
  double b = cp_sub(ry.begin(), ry.end(), xa+split, j1, j2)
     ↪;
  if(b < a) a = b, i1 = j1, i2 = j2;
  double a2 = a*a;
  for(IIt i = ya; i != yaend; ++i) { // Create strip (y-
     ↪sorted)
    double x = (*i)->x;
    if(x >= splitx-a && x <= splitx+a) stripy.push_back(*i)
       ↪;
  }
  for(IIt i = stripy.begin(); i != stripy.end(); ++i) {
    const P &p1 = **i;
    for(IIt j = i+1; j != stripy.end(); ++j) {
      const P &p2 = **j;
      if(p2.y-p1.y > a) break;
      double d2 = (p2-p1).dist2();
      if(d2 < a2) i1 = *i, i2 = *j, a2 = d2;
  } }
  return sqrt(a2);
}

template<class It> // It is random access iterators of
   ↪point<T>
double closestpair(It begin, It end, It &i1, It &i2 ) {
  vector<It> xa, ya;
  assert(end-begin >= 2);
  for (It i = begin; i != end; ++i)
    xa.push_back(i), ya.push_back(i);
  sort(xa.begin(), xa.end(), it_less<It>);
  sort(ya.begin(), ya.end(), y_it_less<It>);
  return cp_sub(ya.begin(), ya.end(), xa.begin(), i1, i2);
} // hash-cpp-all = 42735b8e08701a3b73504ac0690e31df
```

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) x [y1,y2)

Usage: Create vector with lower leftmost and upper rightmost coordinates of each rectangle.//vector<pair<int,int>,pair<int,int>> rectangles;// rectangles.push_back({{1, 3}, {2, 4}});// lint result = rectangle_union_area(rectangles);

63 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[2*i] + tree[2*i+1];
    }
    void push(int v, int lx, int rx) {
        tree[v].first += lazy[v];
        if (lx != rx) {
            lazy[2*v] += lazy[v];
            lazy[2*v+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(2*v, lx, m, a, b, delta);
            update(2*v+1, m+1, rx, a, b, delta);
            tree[v] = (tree[2*v] + tree[2*v+1]);
        }
    }

};

lint rectangle_union_area(vector<pair<pair<int,int>,pair<int,int>>> v) { // area of union of rectangles
  segtree_t L(SZ);
  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++) L.tree[SZ+i].second = y[i+1]-y[i]; // compress coordinates
  L.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];
    L.update(t[2],t[3],t[1]);
    int len = y.back()-y.front()-L.tree[1].second; // L.mn[0].firstshould equal 0
    ans += (lint)(ev[i+1][0]-t[0])*len;
  }
  return ans;
} // hash-cpp-all = 1450ef44b416006a4c7cb39a7d3404ef
```

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

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

29 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;

24 lines

```
int S[N], T[N];

void extKMP(const string &s, const string &t) { // hash-cpp-1
    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;
    } // hash-cpp-1 = 1b7119e667e0c6b48247673c972ecbb7
    int n = s.size(); // hash-cpp-2
    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-2 = 62963ee562740268b77a1234e7c7ae68
}
```

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: $\mathcal{O}(N)$

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

18 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.

68 lines

```
struct Trie {
  int cnt, word;
  map<char, Trie> m;
  Trie() : word(0), cnt(0) { m.clear();}
  void add(const string &s, int i) {
    cnt++;
    if(i ==(int)s.size()) {
      word++;
      return;
    }
    if(!m.count(s[i])) m[s[i]] = Trie();
    m[s[i]].add(s, i + 1);
  }
  bool remove(const string &s, int i) {
    if(i ==(int)s.size()) {
      if(word) {
        cnt--;
        word--;
        return true;
      }
      return false;
    }
    if(!m.count(s[i])) return false;
    if(m[s[i]].remove(s, i + 1) == true) {
      cnt--;
      return true;
    }
    return false;
  }
  bool count(const string &s, int i) {
    if(i ==(int)s.size()) return word;
    if(!m.count(s[i])) return false;
    return m[s[i]].count(s, i + 1);
  }
  bool count_prefix(const string &s, int i) {
    if (word) return true;
    if(i ==(int)s.size()) return false;
    if(!m.count(s[i])) return false;
    return m[s[i]].count_prefix(s, i + 1);
  }

  bool is_prefix(const string &s, int i) {
    if(i ==(int)s.size()) return cnt;
    if(!m.count(s[i])) return false;
    return m[s[i]].is_prefix(s, i + 1);
  }
  void add(const string &s) {
    add(s, 0);
  }
  bool remove(const string &s) {
    return remove(s, 0);
  }
  bool count(const string &s) {
    return count(s, 0);
  }
  // return if trie countains a string that is prefix os s
  // trie has 123, query 12345  returns true
  // trie has 12345, query 123  returns false
  bool count_prefix(const string &s) {
    return count_prefix(s, 0);
  }
  // return if s is prefix of some string countained in trie
  // trie has 12345, query 123  returns true
  // trie has 123, query 12345  returns false
  bool is_prefix(const string &s) {
    return is_prefix(s, 0);
  }
} T; // hash-cpp-all = df13d0b71df32061e3bd54816152e3cf
```

TrieXOR.h

Description: Query max xor with some int in the Trie

30 lines

```
template<int MX, int MXBIT> struct Trie { // hash-cpp-1
    vector<vector<int>> nex;// num is last node in trie
    vector<int> sz;
    int num = 0;
    // change 2 to 26 for lowercase letters
    Trie() {
        nex = vector<vector<int>>(MX, vector<int>(2));
        sz = vector<int>(MX);
    } // hash-cpp-1 = 171b2c3c86583019d3e96ea5c2fcfc4f
    // insert or delete
    void insert(lint x, int a = 1) { // hash-cpp-2
        int cur = 0; sz[cur] += a;
        for(int i = MXBIT-1; i >= 0; --i) {
            int t = (x&(1lint<<i))>>i;
            if (!nex[cur][t]) nex[cur][t] = ++num;
            sz[cur = nex[cur][t]] += a;
        }
    } // hash-cpp-2 = c533ca7f6d0fcdf3a7011207856e065d
    // compute max xor
    lint test(lint x) { // hash-cpp-3
        if (sz[0] == 0) return -INF; // no elements in trie
        int cur = 0;
        for(int i = MXBIT-1; i >= 0; --i) {
            int t = ((x&(1lint<<i))>>i) ^ 1;
            if (!nex[cur][t] || !sz[nex[cur][t]]) t ^= 1;
            cur = nex[cur][t]; if (t) x ^= 1lint<<i;
        }
        return x;
    } // hash-cpp-3 = 3c8060e4c36b53d379b97008c71f1921
};
```

Hashing-codeforces.h

Description: Various self-explanatory methods for string hashing. Use on Codeforces, which lacks 64-bit support and where solutions can be hacked.

<sys/time.h> 50 lines

```
typedef uint64_t ull;
static int C; // initialized below

// Arithmetic mod two primes and 2^32 simultaneously.
// "typedef uint64_t H;" instead if Thue-Morse does not apply.
template<int M, class B>
struct A {
  int x; B b; A(int x=0) : x(x), b(x) {}
  A(int x, B b) : x(x), b(b) {}
  A operator+(A o){int y = x+o.x; return{y - (y>=M)*M, b+o.b};}
```

```
  A operator-(A o){int y = x-o.x; return{y + (y< 0)*M, b-o.
    ↪b};}
  A operator*(A o) { return {(int)(1LL*x*o.x % M), b*o.b};
    ↪}
  explicit operator ull() { return x ^ (ull) b << 21; }
};
typedef A<1000000007, A<1000000009, unsigned>> H;

struct HashInterval {
  vector<H> ha, pw;
  HashInterval(string& str) : ha(sz(str)+1), pw(ha) {
    pw[0] = 1;
    rep(i,0,sz(str))
      ha[i+1] = ha[i] * C + str[i],
      pw[i+1] = pw[i] * C;
  }
  H hashInterval(int a, int b) { // hash [a, b)
    return ha[b] - ha[a] * pw[b - a];
  }
};

vector<H> getHashes(string& str, int length) {
  if (str.size() < length) return {};
  H h = 0, pw = 1;
  for(int i = 0; i < length; ++i)
    h = h * C + str[i], pw = pw * C;
  vector<H> ret = {h};
  for(int i = length; i < str.size(); ++i) {
    ret.push_back(h = h * C + str[i] - pw * str[i-length]);
  }
  return ret;
}

H hashString(string& s) { H h{}; for(auto &c : s) h=h*C+c;
  ↪return h; }

int main() {
  timeval tp;
  gettimeofday(&tp, 0);
  C = (int)tp.tv_usec; // (less than modulo)
  assert((ull)(H(1)*2+1-3) == 0);
  // ...
} // hash-cpp-all = e6c96062e775704f7f7d85fac1232e1c
```

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: $\mathcal{O}(26N)$

50 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
```

SuffixArray.cpp

Description: Builds suffix array for a string. The `lcp` function calculates longest common prefixes for neighbouring strings in suffix array. The returned vector is of size $n+1$, and $ret[0]=0$.
Time: $\mathcal{O}(N\log N)$ where N is the length of the string for creation of the SA. $\mathcal{O}(N)$ for longest common prefixes.

50 lines

```
struct suffix_array_t { // hash-cpp-1
  vector<int> lcp; vector<vector<pair<int, int>>> rmq;
  int n, h; vector<int> sa, invsa;
  bool cmp(int a, int b) { return invsa[a+h] < invsa[b+h];
    ↪}
  void ternary_sort(int a, int b) {
    if (a == b) return;
    int pivot = sa[a+rand()%(b-a)];
    int left = a, right = b;
    for (int i = a; i < b; ++i) if (cmp(sa[i], pivot)) swap
      ↪(sa[i], sa[left++]);
    for (int i = b-1; i >= left; --i) if (cmp(pivot, sa[i])
      ↪) swap(sa[i], sa[--right]);
    ternary_sort(a, left);
    for (int i = left; i < right; ++i) invsa[sa[i]] = right
      ↪-1;
    if (right-left == 1) sa[left] = -1;
    ternary_sort(right, b);
  } // hash-cpp-1 = 3fca933d36bfd1ac53d33525aa3203a2
  suffix_array_t() {} // hash-cpp-2
  suffix_array_t(vector<int> v): n(v.size()), sa(n) {
    v.push_back(INT_MIN);
    invsa = v; iota(sa.begin(), sa.end(), 0);
    h = 0; ternary_sort(0, n);
    for (h = 1; h <= n; h *= 2)
      for (int j = 0, i = j; i != n; i = j)
  if (sa[i] < 0) {
    while (j < n && sa[j] < 0) j += -sa[j];
        sa[i] = -(j-i);
  } // hash-cpp-2 = 045c4939b473f5149c2e552135d12b96
  else { j = invsa[sa[i]]+1; ternary_sort(i, j); } // hash-
    ↪cpp-3
    for (int i = 0; i < n; ++i) sa[invsa[i]] = i;
    lcp.resize(n); int res = 0;
    for (int i = 0; i < n; ++i) {
      if (invsa[i] > 0) while (v[i+res] == v[sa[invsa[i
        ↪]-1]+res]) ++res;
      lcp[invsa[i]] = res; res = max(res-1, 0);
    } // hash-cpp-3 = 90309049bb0fce36d08ad3a8af805d24
    int logn = 0; while ((1<<(logn+1)) <= n) ++logn; //
      ↪hash-cpp-4
    rmq.resize(logn+1, vector<pair<int, int>>(n));
    for (int i = 0; i < n; ++i) rmq[0][i] = make_pair(lcp[i
      ↪], i);
    for (int l = 1; l <= logn; ++l)
      for (int i = 0; i+(1<<l) <= n; ++i)
        rmq[l][i] = min(rmq[l-1][i], rmq[l-1][i+(1<<(l-1))]);
  } // hash-cpp-4 = dc54711f8f7297b8170f572288bf6134
  pair<int, int> rmq_query(int a, int b) { // hash-cpp-5
    int size = b-a+1, l = __lg(size);
    return min(rmq[l][a], rmq[l][b-(1<<l)+1]);
  } // hash-cpp-5 = 6e515b577798ddd26df9f09bf8aa1ae8
  int get_lcp(int a, int b) { // hash-cpp-6
    if (a == b) return n-a;
    int ia = invsa[a], ib = invsa[b];
    return rmq_query(min(ia, ib)+1, max(ia, ib)).first;
  } // hash-cpp-6 = 2ee59379f2812610f89b9c9bee839647
};
```

AhoCorasick.cpp

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

91 lines

```
struct Node {
    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 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
}
// Funcao para adicionar um padrao
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});
}
// Ativar Aho-corasick, ajustando funcoes de falha
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;
        }
    }
}
// Buscar padroes no aho-corasik
void search_all(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 = 1c53345cd6308673461388b1c17b8ddc
```

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
```

PalindromicTree.h

Description: Used infrequently. Palindromic tree computes number of occurrences of each palindrome within string. `ans[i][0]` stores min even number x such that the prefix $s[1..i]$ can be split into exactly x palindromes, `ans[i][1]` does the same for odd x.
Time: $\mathcal{O}(N\sum)$ for addChar, $\mathcal{O}(N \log N)$ for updAns 48 lines

```
template<int SZ> struct PalTree {
    static const int sigma = 26;
    int s[SZ], len[SZ], link[SZ], to[SZ][sigma], oc[SZ];
    int slink[SZ], diff[SZ];
    array<int,2> ans[SZ], seriesAns[SZ];
    int n, last, sz;
    PalTree() {
        s[n++] = -1; link[0] = 1; len[1] = -1; sz = 2;
        ans[0] = {0,MOD};
    }
    int getLink(int v) {
        while (s[n-len[v]-2] != s[n-1]) v = link[v];
        return v;
    }
    void updAns() { // serial path has O(log n) vertices
        ans[n-1] = {MOD,MOD};
        for (int v = last; len[v] > 0; v = slink[v]) {
            seriesAns[v] = ans[n-1-(len[slink[v]]+diff[v])];
            if (diff[v] == diff[link[v]])
                for(int i = 0; i < 2; ++i)
                    seriesAns[v][i] = min(seriesAns[v][i], seriesAns[link[v]][i]);
            // previous oc of link[v] coincides with start of last oc of v
            for(int i = 0; i < 2; ++i)
                ans[n-1][i] = min(ans[n-1][i],seriesAns[v][i^1]+1);
        }
    }
    void addChar(int c) {
        s[n++] = c;
        last = getLink(last);
        if (!to[last][c]) {
            len[sz] = len[last]+2;
            link[sz] = to[getLink(link[last])][c];
            diff[sz] = len[sz]-len[link[sz]];
            if (diff[sz] == diff[link[sz]]) slink[sz] = slink[link[sz]];
            else slink[sz] = link[sz];
            // slink[v] = max suffix u of v such that diff[v]\neq diff[u]
            to[last][c] = sz++;
        }
        last = to[last][c]; oc[last] ++;
        updAns();
    }
    void numOc() { // # occurrences of each palindrome
        vector<pair<int,int>> v;
        for(int i = 2; i < sz; ++i) v.push_back({len[i],i});
        sort(v.rbegin(), v.rend());
        for(auto& a : v) oc[link[a.second]] += oc[a.second];
    }
}; // hash-cpp-all = afab6add7b90a5ae6e99231a523dc26c
```

ReverseBurrowsWheeler.h

Description: Reverse of Burrows-Wheeler
Time: $\mathcal{O}(nlog(n))$ 16 lines

```
string RBW(string &s) {
    vector<pair<char,int>> v;
    vector<int> nex(s.size());
    for (int i = 0; i < s.size(); ++i)
        v.push_back({s[i], i});
    sort(v.begin(), v.end());
    for (int i = 0; i < s.size(); ++i)
        nex[i] = v[i].second;
    int cur = nex[0];
    string result;
    while(cur) {
```

```
        result += v[cur].first;
        cur = nex[cur];
    }
    return result;
} // hash-cpp-all = fd5d9fc744ee311a9d51a7e90afa38ad
```

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 $f(a) < \dots < f(i) \geq \dots \geq f(b)$. 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: $\mathcal{O}(\log(b-a))$

11 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+1;i++) if (f(a) < f(i)) a = i; // (B)
  return a;
} // hash-cpp-all = 0b750a57790807d99a432f12841f1af2
```

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

19 lines

```
int n, inv;
vector<int> v, result;

void merge_sort(int lx, int rx, vector<int> &v) {
    if (lx == rx) return;
    int m = lx + (rx - lx)/2;
    merge_sort(lx, m, v);
    merge_sort(m+1, rx, v);
    int i = lx, j = m+1, k = lx;
    while(i <= m || j <= rx) {
        if (i <= m && (j > rx || v[i] < v[j])) {
            result[k++] = v[i++];
            inv += (j - k);
        }
        else result[k++] = v[j++];
    }
    for (int i = lx; i <= rx; ++i)
        v[i] = result[i];
} // hash-cpp-all = 34a7b0c31ffe6abe903916da641d98b3
```

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

13 lines

```
int64_t isqrt(int64_t n) {
    int64_t left = 0;
    int64_t right = 10000000;
    while (right - left > 1) {
        int64_t middle = (left + right) / 2;
        if (middle * middle <= n) {
            left = middle;
        } else {
            right = middle;
        }
    }
    return left;
} // hash-cpp-all = fc5f42aa60261c39ccc263bfba494ef1
```

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> 7 lines

```
FT<int> bit(maxv+10);
int inv = 0;
for (int i = n-1; i >= 0; --i) {
    inv += bit.query(v[i]); // careful with the interval
    bit.update(v[i], 1);    // [0, x) or [0, x] ?
}
// hash-cpp-all = 3582f611430853173f9f3cf4efb5d3ff
```

Histogram.h

Description: Maximum rectangle in histogram

16 lines

```
lint histogram(lint vet[], int n) {
  stack<lint> s;
  lint ans = 0, tp, cur;
  int i = 0;
  while(i < n || !s.empty()) {
    if (i < n && (s.empty() || vet[s.top()] <= vet[i])) s.push(i++);
    else {
      tp = s.top();
      s.pop();
      cur = vet[tp] * (s.empty() ? i : i - s.top() - 1);
      if (ans < cur) ans = cur;
    }
  }
  return ans;
}
// hash-cpp-all = a9d3f9be854b498aa88dfb2dc149ea9c
```

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
```

MagicSquare.h

17 lines

```
int mat[MAXN][MAXN], n; //0-indexed

void magicsquare() {
  int i=n-1, j=n/2;
  memset(&mat, 0, sizeof mat);
  for(int k = 1; k <= n*n; k++) {
    mat[i][j] = k;
    if (mat[(i+1)%n][(j+1)%n] > 0) {
      i = (i-1+n)%n;
    }
    else {
      i = (i+1)%n;
      j = (j+1)%n;
    }
  }
}
// hash-cpp-all = e5f5fb9897d7b39b4fa47e4070ee0704
```

FindPattern.h

17 lines

```
bool test(vector<int> &v, int init, int size, int lim){
  for(int i = init; i < lim; ++i)
    if(v[init + ((i-init)%size)] != v[i])
      return false;
  return true;
}

void identifyPattern(vector<int> &v, int lim){
  for(int init = 0; init < lim; ++init){
    for(int size = 1; size < 500; ++size){
      if(test(v, init, size, lim)){
        cout << init << " " << size << endl;
        break;
      }
    }
  }
} // hash-cpp-all = 45155504b29b390a722aa33cc2ae5a24
```

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

43 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;
        //cout<<row<<" "<<col<<endl;
        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: $\mathcal{O}(n)$

34 lines

```
lint a, b, c;

lint f(lint x) {
  return (a * x + (x % b)) % c;
}

//mu -> first ocurrence
//lambda -> cycle length
lint mu, lambda;

void Floyd(lint x0) {
    //hare -> fast pointer
    //tortoise -> slow pointer
    lint hare, tortoise;
    tortoise = f(x0);
    hare = f(f(x0));
    while(hare != tortoise) {
        tortoise = f(tortoise);
        hare = f(f(hare));
    }
    hare = x0;
    mu = 0;
    while(tortoise != hare) {
        tortoise = f(tortoise);
        hare = f(hare);
        mu++;
    }
    hare = f(tortoise);
    lambda = 1;
    while(t != h) {
        hare = f(hare);
        lambda++;
    }
} // hash-cpp-all = eb059fec84c1516c7f9a827c6c36ee4c
```

SlidingWindow.h

Description: Given an array v and an integer K, the problem boils down to computing for each index $i : min(v[i], v[i-1], ..., v[i-K+1])$. if $mx == true$, returns the maximum.
Time: $\mathcal{O}(N)$

18 lines

```
vector<int> sliding_window_minmax(vector<int> &v, int K,
   ↪bool mx) {
  deque< pair<int, int>> window;
  vector<int> ans;
  for (int i = 0; i < v.size(); i++) {
    if (mx) {
      while (!window.empty() && window.back().first <= v[i
         ↪])
        window.pop_back();
    } else {
      while (!window.empty() && window.back().first >= v[i
         ↪])
        window.pop_back();
    }
    window.push_back(make_pair(v[i], i));
    while(window.front().second <= i - K)
      window.pop_front();
    ans.push_back(window.front().first);
  }
  return ans;
} // hash-cpp-all = 71875466ea0431246dff646437250de6
```

Scanline.h

Description: Scanline (Merge all overalapping intervals into a single interval)
Usage: O(N)

11 lines

```
void scanline(vector<pair<int,int>> p, vector<pair<int,int
   ↪>> &intervals) {
  int f = p[0].first, l = p[0].second;
  for (int i = 0; i < m; ++i) {
    if (p[i].first <= l) l = max(l, p[i].second);
    else {
      intervals.push_back({f, l});
      f = p[i].first, l = p[i].second;
    }
  }
  intervals.push_back({f, l});
} // hash-cpp-all = d8de9398e495a1a7dafe79a1326213b0
```


SlidingWindow.h

Description: Given an array v and an integer K, the problem boils down to computing for each index $i : min(v[i], v[i-1], ..., v[i-K+1])$. if $mx == true$, returns the maximum.
Time: $\mathcal{O}(N)$

18 lines

```
vector<int> sliding_window_minmax(vector<int> &v, int K,
   ↪bool mx) {
  deque< pair<int, int>> window;
  vector<int> ans;
  for (int i = 0; i < v.size(); i++) {
    if (mx) {
      while (!window.empty() && window.back().first <= v[i
         ↪])
        window.pop_back();
    } else {
      while (!window.empty() && window.back().first >= v[i
         ↪])
        window.pop_back();
    }
    window.push_back(make_pair(v[i], i));
    while(window.front().second <= i - K)
      window.pop_front();
    ans.push_back(window.front().first);
  }
  return ans;
} // hash-cpp-all = 71875466ea0431246dff646437250de6
```

10.3 Dynamic programming

DivideAndConquerDP.h

Description: Optimizes dp of the form (or similar) $dyn[i][j] = min_{k<i}(dyn[k][j-1] + f(k+1,i))$. 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 $\mathcal{O}(n^2 m)$ to $\mathcal{O}(nm\log(n))$ 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> dyn;
    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[l...r][j], knowing that opt[l...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;
        dyn[mid][j] = INF;
        for (int k = low_opt; k <= high_opt && k < mid; ++k)
            if (dyn[k][j-1] + f(k + 1, mid) < dyn[mid][j]){
                dyn[mid][j] = dyn[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.dyn[i][0] = INF;

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

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

  cout << DP.dyn[n][m] << endl;
}
// hash-cpp-all = f9d57965a870cfc0ac239c3c0789fb25
```

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 $\mathcal{O}(N^3)$ to $\mathcal{O}(N^2)$ 30 lines

```
array<array<lint, 1123>, 1123> dyn;
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) {
            dyn[i][i - 1] = 0
            opt[i][i - 1] = i;
        }
        for (int i = n; i > 0; --i)
            for (int j = i; j <= n; ++j) {
                dyn[i][j] = LLONG_MAX; // INF
                for (int k = max(i, opt[i][j - 1]); k <= j && k <= opt[i + 1][j]; ++k)
                    if (dyn[i][k - 1] + dyn[k + 1][j] + f(i, j) < dyn[i][j]) {
                        dyn[i][j] = dyn[i][k - 1] + dyn[k + 1][j] + f(i, j);
                        opt[i][j] = k;
                    }
            }
        cout << dyn[1][n] << '\n';
    }
} // hash-cpp-all = 0bd5b9607c21b45ba61ecb55cde1ecae
```

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: $\mathcal{O}(N)$ 8 lines

```
int coin(vector<int> &c, int k) {
    vector<int> dp(k+1, 0); dp[0] = 1;
    for (int i = 0; i < c.size(); ++i)
        for (int j = c[i]; j <= k; ++j)
            dp[j] += dp[j-c[i]];
    return dp[k];
}
// hash-cpp-all = c38f010ad4252350bcc4fc8967fd1159
```

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

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+1; i++) for(int j = 1; j < b+1; 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 = b096b75c43618ce1ea19738b94be83fb
```

Knapsack.h

Description: Knapsack 01 problem, returns a vector that hold the quantity of items chosen and its values.
Time: $\mathcal{O}(N \log N)$

16 lines

```
vector<int> Knapsack(int limit, vector<int> &v, vector<int>
  ↪ &w) {
    vector<vector<int>> dyn(v.size()+1);
    dyn[0].resize(limit+1);
    for (int i = 0; i < v.size(); ++i) {
        dyn[i+1] = dyn[i];
        for (int j = 0; j <= limit - w[i]; ++j)
            dyn[i+1][w[i]+j] = max(dyn[i+1][w[i]+j], dyn[i
              ↪][j] + v[i]);
    }
    vector<int> result;
    for (int i = v.size()-1; i >= 0; --i)
        if (dyn[i][limit] != dyn[i+1][limit]) {
            limit -= w[i];
            result.push_back(i);
        }
    return result;
} // hash-cpp-all = 09847e2c75f917d2ae747f1d67edd253
```

01Knapsack.h

Description: Bottom-up is faster in practice
Time: $\mathcal{O}(N \log N)$

35 lines

```
// 1-indexed bottom-up, faster in practice
int Knapsack(int limit, vector<int> &v, vector<int> &w) {
    vector<int> dyn(10*v.size(), -1); int n = w.size();
    dyn[0] = 0;
    for (int i = 0; i < n; ++i)
        for (int j = limit; j >= w[i]; --j)
            if (dyn[j - w[i]] >= 0)
                dyn[j] = max(dyn[j], dyn[j - w[i]] + v[i]);
    int result = 0;
    for (int i = 0; i <= limit; ++i)
        result = max(result, dyn[i]);
    return result;
}
// top-down
int n, c; // total of items and cost
array<int, MAXN> w, v; // weight, value
array<array<int, MAXN>, MAXN> dyn; // filled -1

int get(int idx, int cap) {
    if (cap < 0) return -INT_MAX;
    if (idx == n) return 0;
    if (dyn[idx][cap] != -1) return dyn[idx][cap];
    return dyn[idx][cap] = max(get(idx+1, cap), v[idx] +
      ↪get(idx+1, cap - w[idx]));
}

void recover(int idx, int cap) {
    if (idx == n) return;
    int grab = v[idx] + get(idx+1, cap - w[idx]);
    int change = get(idx+1, cap);
    if (grab >= change) {
        items.push_back(idx);
        recover(idx+1, cap - w[idx]);
    }
    else recover(idx+1, cap);
} // hash-cpp-all = dd79ee9b0bde249ce084503065d827bc
```

LargeKnapsack.h

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

9 lines

```
const int max_value = (int)1e5+10;
int knapsack2(vector<lint> &v, vector<lint> &w, int n, int
  ↪total) {
    vector<lint> dp(max_value, 2e18); dp[0] = 0;
    for (int i = 0; i < n; ++i)
        for (int j = max_value - v[i] - 1; j >= 0; --j)
            dp[j + v[i]] = min(dp[j + v[i]], dp[j] + w[i]);
    for (int i = max_value-1; i >= 0; --i)
        if (dp[i] <= total) return i;
} // hash-cpp-all = e49b98b8006fe6f48e59ccc119f9c8b1
```

KnapsackUnbounded.h

Description: Knapsack problem but repetitions are allowed.
Time: $\mathcal{O}(N \log N)$

10 lines

```
int unbounded_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 = 390c2286ce88b58740d71dc5ba395434
```

KnapsackBounded.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 weighs $w[i]$ and costs $c[i]$. 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> dyn; // the minimum cost dyn[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++) dyn[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(dyn[j]);
      dyn[j] = mq.getMin();
    }
  }
  cout << "Minimum value i can pay putting a total weight "
    ↪ << m << " is " << dyn[m] << '\n';
```

```
  for (int i = 0; i <= m; i++) cout << dyn[i] << " " << i
    << '\n';
  cout << "\n";
} // hash-cpp-all = cac0faadab0e006a19e0104670f4b9ef
```

KnapsackBitset.h

Time: $\mathcal{O}(N \log N)$ 12 lines

```
bitset<MAX> 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 = a6f378c86ddc023e5dd53ac1236f7093
```

TSP.h

Description: Solve the Travelling Salesman Problem.

Time: $\mathcal{O}\left(N^2 * 2^N\right)$ 18 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> 7 lines

```
num tot[30];
num distinct(const string &S) {
    num ans = 1; // tot[i] stands for number of distinct
      strings ending with character 'a'+i
    for(auto &c : S) tie(ans, tot[c-'a']) = make_pair(2*ans
      -tot[c-'a'],ans);
    return ans-1;
}
// hash-cpp-all = 7ec0c8d69757e755ccf5d3d3338a8f92
```

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
```

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
```

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
```

Unrolling.h

6 lines

```
#define F {...; ++i;}
int i = from;
while (i&3 && i < to) F // for alignment, if needed
while (i + 4 <= to) { F F F F }
while (i < to) F
// hash-cpp-all = 69ac737ad5a50f5688d5720fb6fce39f
```

SIMD.h

Description: Cheat sheet of SSE/AVX intrinsics, for doing arithmetic on several numbers at once. Can provide a constant factor improvement of about 4, orthogonal to loop unrolling. Operations follow the pattern "_mm(256)?_name_(si(128|256)|epi(8|16|32|64)|pd|ps)". Not all are described here; grep for _mm_ in /usr/lib/gcc/*/4.9/include/ for more. If AVX is unsupported, try 128-bit operations, "emmintrin.h" and #define __SSE__ and __MMX__ before including it. For aligned memory use `_mm_malloc(size, 32)` or `int buf[N] alignas(32)`, but prefer loadu/storeu.

43 lines

```
#pragma GCC target ("avx2") // or sse4.1
#include "immintrin.h"

typedef __m256i mi;
#define L(x) _mm256_loadu_si256((mi*)&(x))

// High-level/specific methods:
// load(u)?_si256, store(u)?_si256, setzero_si256,
    _mm_malloc
// blendv_(epi8|ps|pd) (z?y:x), movemask_epi8 (hibits of
    bytes)
// i32gather_epi32(addr, x, 4): map addr[] over 32-b parts
    of x
// sad_epu8: sum of absolute differences of u8, outputs 4
    xi64
// maddubs_epi16: dot product of unsigned i7's, outputs 16
    xi15
// madd_epi16: dot product of signed i16's, outputs 8xi32
// extractf128_si256(, i) (256->128), cvtsi128_si32 (128->
    lo32)
// permute2f128_si256(x,x,1) swaps 128-bit lanes
// shuffle_epi32(x, 3*64+2*16+1*4+0) == x for each lane
// shuffle_epi8(x, y) takes a vector instead of an imm

// Methods that work with most data types (append e.g.
    _epi32):
// set1, blend (i8?x:y), add, adds (sat.), mullo, sub, and/
    or,
// andnot, abs, min, max, sign(1,x), cmp(gt|eq), unpack(lo|
    hi)

int sumi32(mi m) { union {int v[8]; mi m;} u; u.m = m;
  int ret = 0; rep(i,0,8) ret += u.v[i]; return ret; }
mi zero() { return _mm256_setzero_si256(); }
mi one() { return _mm256_set1_epi32(-1); }
bool all_zero(mi m) { return _mm256_testz_si256(m, m); }
bool all_one(mi m) { return _mm256_testc_si256(m, one()); }

ll example_filteredDotProduct(int n, short* a, short* b) {
  int i = 0; ll r = 0;
  mi zero = _mm256_setzero_si256(), acc = zero;
  while (i + 16 <= n) {
    mi va = L(a[i]), vb = L(b[i]); i += 16;
    va = _mm256_and_si256(_mm256_cmpgt_epi16(vb, va), va);
    mi vp = _mm256_madd_epi16(va, vb);
    acc = _mm256_add_epi64(_mm256_unpacklo_epi32(vp, zero),
      _mm256_add_epi64(acc, _mm256_unpackhi_epi32(vp, zero)
        ));
  }
  union {ll v[4]; mi m;} u; u.m = acc; for(int i=0;i<4;i++)
    r += u.v[i];
  for (;i<n;++i) if (a[i] < b[i]) r += a[i]*b[i]; // <-
    equiv
  return r;
} // hash-cpp-all = f6fcb50f92027098053182262274f061
```

Hashmap.h

Description: Faster/better hash maps, taken from CF

14 lines

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
gp_hash_table<int, int> table;

struct custom_hash {
  size_t operator()(uint64_t x) const {
    x += 48;
    x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
    x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
    return x ^ (x >> 31);
  }
};
gp_hash_table<int, int, custom_hash> safe_table;
// hash-cpp-all = e62eb2668aee2263b6d72043f3652fb2
```

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
```

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

18 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)`.

Techniques (A)

techniques.txt 159 lines

```
Recursion
Divide and conquer
  Finding interesting points in N log N
Algorithm analysis
  Master theorem
  Amortized time complexity
Greedy algorithm
  Scheduling
  Max contiguous subvector sum
  Invariants
  Huffman encoding
Graph theory
  Dynamic graphs (extra book-keeping)
  Breadth first search
  Depth first search
  * Normal trees / DFS trees
  Dijkstra's algorithm
  MST: Prim's algorithm
  Bellman-Ford
  Konig's theorem and vertex cover
  Min-cost max flow
  Lovasz toggle
  Matrix tree theorem
  Maximal matching, general graphs
  Hopcroft-Karp
  Hall's marriage theorem
  Graphical sequences
  Floyd-Warshall
  Euler cycles
  Flow networks
  * Augmenting paths
  * Edmonds-Karp
  Bipartite matching
  Min. path cover
  Topological sorting
  Strongly connected components
  2-SAT
  Cut vertices, cut-edges och biconnected components
  Edge coloring
  * Trees
  Vertex coloring
  * Bipartite graphs (=> trees)
  * 3^n (special case of set cover)
  Diameter and centroid
  K'th shortest path
  Shortest cycle
Dynamic programming
  Knapsack
  Coin change
  Longest common subsequence
  Longest increasing subsequence
  Number of paths in a dag
  Shortest path in a dag
  Dynprog over intervals
  Dynprog over subsets
  Dynprog over probabilities
  Dynprog over trees
  3^n set cover
  Divide and conquer
  Knuth optimization
  Convex hull optimizations
  RMQ (sparse table a.k.a 2^k-jumps)
  Bitonic cycle
  Log partitioning (loop over most restricted)
Combinatorics
  Computation of binomial coefficients
  Pigeon-hole principle
  Inclusion/exclusion
  Catalan number
  Pick's theorem
Number theory
  Integer parts
  Divisibility
  Euclidean algorithm
  Modular arithmetic
  * Modular multiplication
  * Modular inverses
  * Modular exponentiation by squaring
  Chinese remainder theorem
  Fermat's little theorem
  Euler's theorem
  Phi function
  Frobenius number
  Quadratic reciprocity
  Pollard-Rho
  Miller-Rabin
  Hensel lifting
  Vieta root jumping
Game theory
  Combinatorial games
  Game trees
  Mini-max
  Nim
  Games on graphs
  Games on graphs with loops
  Grundy numbers
  Bipartite games without repetition
  General games without repetition
  Alpha-beta pruning
Probability theory
Optimization
  Binary search
  Ternary search
  Unimodality and convex functions
  Binary search on derivative
Numerical methods
  Numeric integration
  Newton's method
  Root-finding with binary/ternary search
  Golden section search
Matrices
  Gaussian elimination
  Exponentiation by squaring
Sorting
  Radix sort
Geometry
  Coordinates and vectors
  * Cross product
  * Scalar product
  Convex hull
  Polygon cut
  Closest pair
  Coordinate-compression
  Quadtrees
  KD-trees
  All segment-segment intersection
Sweeping
  Discretization (convert to events and sweep)
  Angle sweeping
  Line sweeping
  Discrete second derivatives
Strings
  Longest common substring
  Palindrome subsequences
  Knuth-Morris-Pratt
  Tries
  Rolling polynomial hashes
  Suffix array
  Suffix tree
  Aho-Corasick
  Manacher's algorithm
  Letter position lists
Combinatorial search
  Meet in the middle
  Brute-force with pruning
  Best-first (A*)
  Bidirectional search
  Iterative deepening DFS / A*
Data structures
  LCA (2^k-jumps in trees in general)
  Pull/push-technique on trees
  Heavy-light decomposition
  Centroid decomposition
  Lazy propagation
  Self-balancing trees
  Convex hull trick (wcipeg.com/wiki/Convex_hull_trick)
  Monotone queues / monotone stacks / sliding queues
  Sliding queue using 2 stacks
  Persistent segment tree
```