ETH zürich

regrETHs

Ema Skottova, Marc Dufay, Theodor Moroianu

Osijek Training Camp

Feb, 2024

Contest (1)

.bashrc
<div align="right">3 lines</div>

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

.vimrc
<div align="right">6 lines</div>

```
set cin aw ai is ts=4 sw=4 tm=50 nu noeb bg=dark ru cul
sy on   |   im jk <esc>   |   im kj <esc>   |   no ; :
" Select region and then type :Hash to hash your selection.
" Useful for verifying that there aren't mistypes.
ca Hash w !cpp -dD -P -fpreprocessed \| tr -d '[:space:]' \
 \| md5sum \| cut -c-6
```

hash.sh
<div align="right">3 lines</div>

```
# Hashes a file, ignoring all whitespace and comments. Use for
# verifying that code was correctly typed.
cpp -dD -P -fpreprocessed | tr -d '[:space:]'| md5sum |cut -c-6
```

troubleshoot.txt
<div align="right">52 lines</div>

```
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 data structures 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 teammate.
Ask the teammate 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 teammate do it.

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

Time limit exceeded:
Do you have any possible infinite loops?
What 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 teammates think about your algorithm?

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

Mathematics (2)

2.1 Equations

$$ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

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

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

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

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

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

2.2 Trigonometry

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

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

$$\sin v + \sin w = 2 \sin \frac{v + w}{2} \cos \frac{v - w}{2}$$

$$\cos v + \cos w = 2 \cos \frac{v + w}{2} \cos \frac{v - w}{2}$$

$$(V + W) \tan(v - w)/2 = (V - W) \tan(v + w)/2$$

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

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

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

2.3 Algebra

Let $H \subseteq \mathbb{Z}_n$ non-empty subset. H is subgroup iff exists d divisor of n such that $H = \hat{d}\mathbb{Z}_n$.

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

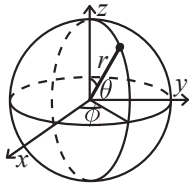Law of tangents: $\dfrac{a + b}{a - b} = \dfrac{\tan \dfrac{\alpha + \beta}{2}}{\tan \dfrac{\alpha - \beta}{2}}$

2.4.2 Quadrilaterals

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

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

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

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

2.5 Derivatives/Integrals

$$\frac{d}{dx}\arcsin x = \frac{1}{\sqrt{1-x^2}} \qquad \frac{d}{dx}\arccos x = -\frac{1}{\sqrt{1-x^2}}$$

$$\frac{d}{dx}\tan x = 1 + \tan^2 x \qquad \frac{d}{dx}\arctan x = \frac{1}{1+x^2}$$

$$\int \tan ax = -\frac{\ln|\cos ax|}{a} \qquad \int x\sin ax = \frac{\sin ax - ax\cos ax}{a^2}$$

$$\int e^{-x^2} = \frac{\sqrt{\pi}}{2}\mathrm{erf}(x) \qquad \int xe^{ax}dx = \frac{e^{ax}}{a^2}(ax-1)$$

Integration by parts

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

Beta Function

$$B(x,y) = n\int_0^1 t^{nx-1}(1-t^n)^{y-1}dt$$

$$B(x,y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$$

$$B(x,y) = B(x,y+1) + B(x+1,y)$$

$$B(x+1,y) = B(x,y)\frac{x}{x+y}$$

$$B(x,y) = B(y,x)$$

$$B(1,x) = \frac{1}{x}$$

$$B(x;a,b) = \int_0^x t^{a-1}(1-t)^{b-1}dt$$

$$B(1;a,b) = B(a,b)$$

$$I_x(a,b) = \frac{B(x;a,b)}{B(a,b)}$$

$$I_0(a,b) = 0 \qquad I_1(a,b) = 1 \qquad I_x(a,1) = x^a$$

$$I_x(1,b) = 1 - (1-x)^b \qquad I_x(a,b) = 1 - I_{1-x}(b,a)$$

$$I_x(a+1,b) = I_x(a,b) - \frac{x^a(1-x)^b}{aB(a,b)}$$

$$I_x(a,b+1) = I_x(a,b) + \frac{x^a(1-x)^b}{bB(a,b)}$$

Gamma Function

$$\Gamma(n) = (n-1)!$$

$$\Gamma(z) = \int_0^\infty x^{z-1}e^{-x}dx$$

$$\Gamma(1/2) = \sqrt{\pi}$$

$$\Gamma(z+1) = z\Gamma(z)$$

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) $\mu = \mathbb{E}(X) = \sum_x x p_X(x)$ and variance $\sigma^2 = V(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = \sum_x (x - \mathbb{E}(X))^2 p_X(x)$ where σ is the standard deviation. If X is instead continuous it will have a probability density function $f_X(x)$ and the sums above will instead be integrals with $p_X(x)$ replaced by $f_X(x)$.

Expectation is linear:

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

For independent X and Y,

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

2.8.1 Discrete distributions
Binomial distribution

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

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

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

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

For n large enough, $\mathrm{Bin}(n,p)$ is approximately $\mathcal{N}(np, np(1-p))$.

Negative binomial $\mathrm{NBin}(r,p)$:

$$F(r+k) = 1 - I_p(k+1,r) = I_{1-p}(r,k+1) = F_{\mathrm{Bin}}(k,k+r,p)$$

First success distribution

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

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

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

Poisson distribution

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

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

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

$$\lambda = rt$$

2.8.2 Continuous distributions

Uniform distribution

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

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

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

Exponential distribution

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

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

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

Normal distribution

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

Data structures (3)

OrderStatisticTree.h
Description: A set (not multiset!) with support for finding the nth element, and finding the index of an element. To get a map, change `null_type`.
Time: $\mathcal{O}(\log N)$
<div align="right">abff78, 16 lines</div>

```cpp
#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
}
```

HashMap.h
Description: Hash map with mostly the same API as unordered_map, but ~3x faster. Uses 1.5x memory. Initial capacity must be a power of 2 (if provided).
<div align="right">260e4d, 8 lines</div>

```cpp
#include <bits/extc++.h>

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

SegTree.h
Description: Very fast and quick segment tree. Only useful for easy invariants. 0-indexed. Range queries are half-open.
<div align="right">39c971, 17 lines</div>

```cpp
struct SegTree {
  vector<ll> T; int n;
  SegTree(int n) : T(2 * n, -INF), n(n) {}

  void Update(int pos, ll val) {
    for (T[pos += n] = val; pos /= 2)
      T[pos] = f(T[pos * 2], T[pos * 2 + 1]);
  }
  ll Query(int b, int e) {
    ll r1 = -INF, r2 = -INF;
    for (b += n, e += n; b < e; b /= 2, e /= 2) {
      if (b % 2) r1 = max(r1, T[b++]);
      if (e % 2) r2 = max(T[--e], r2);
    }
    return max(r1, r2);
  }
}
```

```cpp
};
```

RMQ.h
Description: Disjoint Sparse Table implementation. Returns $\min_{i=l}^{r-1} v[i]$ in constant time. Can be adapted for other things, like sum or product.
Usage: `RMQ rmq(values);`
`rmq.Query(inclusive, exclusive);`
Time: $\mathcal{O}(N \log N + Q)$
<div align="right">08a630, 20 lines</div>

```cpp
struct RMQ {
  vector<int> dp[32];

  RMQ(const vector<int> &v) {
    int n = v.size();
    for (int h = 0, l = 1; l <= n; ++h, l *= 2) {
      dp[h].resize(n + 1, 1e9);
      for (int m = l; m < n + l; m += 2 * l) {
        for (int i = m + 1; i <= min(n, m + l); i++)
          dp[h][i] = min(dp[h][i - 1], v[i - 1]);
        for (int i = min(n, m) - 1; i >= m - l; i--)
          dp[h][i] = min(v[i], dp[h][i + 1]);
      }
    }
  }
  int Query(int l, int r) {
    int h = 31 - __builtin_clz(l ^ r);
    return min(dp[h][l], dp[h][r]);
  }
};
```

BiVec.h
Description: Bi-directional vector (indices in $[-n..n)$).
<div align="right">adce61, 7 lines</div>

```cpp
template<typename T>
struct BiVec {
  vector<T> v;
  BiVec(int n, T x = {}) : v(2 * n, x) {}
  T& operator[](int i) { return v[2 * max(i, ~i) + (i < 0)]; }
  void resize(int n) { v.resize(2 * n); }
};
```

CHT.h
Description: Simple convex hull trick. Requires lines to be inserted in increasing slopes order. Queries have to be increasing in x, but it can be relaxed to any values, by doing binary search on the stack.
Time: $\mathcal{O}(1)$ amortized per operation
<div align="right">ae84bc, 23 lines</div>

```cpp
struct CHT {
  deque<tuple<ll, ll, ll>> stk;
  ll ceil(ll a, ll b) { return a / b + (a % b > 0); }

  void InsertLine(ll a, ll b) {
    ll p = -INF;
    while (stk.size()) {
      auto [pp, ap, bp] = stk.back();
      if (ap == a) p = bp > b ? INF : -INF;
      else p = ceil(bp - b, a - ap);
      if (p > pp) break;
      stk.pop_back();
    }
    if (p != INF) stk.emplace_back(p, a, b);
  }
  ll EvalMax(ll x) {
    // prev(upper_bound(all(stk), {x, INF, INF}))
    while ((int)stk.size() > 1 && get<0>(stk[1]) <= x)
      stk.pop_front();
    auto [_, a, b] = stk.front();
    return a * x + b;
  }
}
```

LineContainer.h

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

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

0757b0, 30 lines

```cpp
bool QUERY;
struct Line {
  mutable ll a, b, p;
  ll Eval(ll x) const { return a * x + b; }
  bool operator<(const Line& o) const {
    return QUERY ? p < o.p : a < o.a;
  }
};
struct LineContainer : multiset<Line> {
  ll div(ll a, ll b) { return a / b - (a % b < 0); }
  void InsertLine(ll a, ll b) {
    auto isect = [&](auto x, auto y) {
      if (y == end()) return x->p = INF, false;
      if (x->a == y->a) x->p = x->b > y->b ? INF : -INF;
      else x->p = div(x->b - y->b, y->a - x->a);
      return x->p >= y->p;
    };
    auto nx = insert({a, b, 0}), it = nx++, pv = it;
    while (isect(it, nx)) nx = erase(nx);
    if (pv != begin() && isect(--pv, it))
      isect(pv, it = erase(it));
    while ((it = pv) != begin() && (--pv)->p >= it->p)
      isect(pv, erase(it));
  }
  ll EvalMax(ll x) {
    assert(!empty());
    QUERY = 1; auto it = lower_bound({0, 0, x}); QUERY = 0;
    return it->Eval(x);
  }
};
```

ConvexTree.h

Description: Container where you can add lines of the form $a \cdot X + b$, and query maximum values at points x. Useful for dynamic programming. To change to minimum, either change the sign of all comparisons, or just add lines of form $(-a) \cdot X + (-b)$ instead and negate the result.

Time: $\mathcal{O}(\log MAX)$ per operation

1969de, 39 lines

```cpp
struct Line { ll a, b; };

struct ConvexTree {
  struct Node { int dp; int l = -1, r = -1; };
  vector<Node> T; vector<Line> L;
  const int MAX = 1e9; // >= abs(x) for all queries.
  int root = -1;

  ll eval(int i, int x) { return L[i].a * x + L[i].b; }
  int update(int node, int b, int e, int upd) {
    if (node == -1) {
      T.push_back({upd});
      return T.size() - 1;
    }
    int& cur = T[node].dp;
    int m = (b + e) / 2;
    int l = T[node].l, r = T[node].r;
    bool bb = eval(upd, b) > eval(cur, b);
    bool bm = eval(upd, m) > eval(cur, m);
    if (bm) swap(cur, upd);
    if (e - b == 1) { /* do nothing */ }
    else if (bb != bm) l = update(l, b, m, upd);
    else r = update(r, m, e, upd);
```

```cpp
    T[node].l = l; T[node].r = r;
    return node;
  }
  ll query(int node, int b, int e, int x) {
    if (node == -1) return -INF; // <= a * x + b for all lines.
    int m = (b + e) / 2;
    return max(eval(T[node].dp, x), (x <= m
      ? query(T[node].l, b, m, x)
      : query(T[node].r, m + 1, e, x)));
  }
  void Update(Line l) {
    L.push_back(l);
    root = update(root, -MAX, MAX, L.size() - 1);
  }
  ll Query(int x) { return query(root, -MAX, MAX, x); }
};
```

KinTour.h

Description: Kinetic tournament data structure. Allows updates of type modify function $f_i(\cdot)$ and query maximum of $f_i(t)$. Queries have to be increasing in t. Modify at will. Useful for temporal sweeps.

Time: Approx $\mathcal{O}(\log^2 N)$ amortized per operation.

c25240, 36 lines

```cpp
struct Line { ll a, b; };

struct KinTour {
  struct Node { int dp = 0; ll ev = INF; };

  int n; ll t = -INF;
  vector<Node> T; vector<Line> L;

  KinTour(int n) : n(n), T(2 * n), L(n, Line{0, 0}) {}

  ll eval(int i) { return L[i].a * t + L[i].b; }
  ll div(ll a, ll b) { return a / b - (a % b < 0); }
  ll beats(int i, int j) {
    // we know i beats j at t; when will j beat i after t?
    if (L[i].a >= L[j].a) return INF;
    return 1 + div(L[i].b - L[j].b, L[j].a - L[i].a);
  }
  void go(int x, int b, int e, int pos) {
    if ((e <= pos || b > pos) && T[x].ev > t) return;
    if (e - b == 1) { T[x].dp = b; return; }
    int m = (b + e) / 2, z = x + 2 * (m - b);
    go(x + 1, b, m, pos); go(z, m, e, pos);
    int i = T[x + 1].dp, j = T[z].dp;
    if (eval(i) < eval(j)) swap(i, j);
    T[x] = {i, min({T[x + 1].ev, T[z].ev, beats(i, j)})};
    assert(T[x].ev > t);
  }
  ll EvalMax(ll tt) {
    assert(t <= tt); t = tt;
    go(1, 0, n, -1);
    return eval(T[1].dp);
  }
  void Update(int i, Line l) {
    L[i] = l; go(1, 0, n, i);
  }
};
```

FenwickTree.h

Description: Adds a value to a (half-open) range and computes the sum on a (half-open) range. Beware of overflows!

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

96578e, 20 lines

```cpp
struct FenwickTree {
  int n;
  vector<ll> T1, T2;
```

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

  void Update(int b, int e, ll val) {
    if (e < n) return Update(e, n, -val), Update(b, n, +val);
    ll c1 = val, c2 = val * (b - 1);
    for (int pos = b + 1; pos <= n; pos += (pos & -pos))
      T1[pos] += c1, T2[pos] += c2;
  }
  ll Query(int b, int e) {
    if (b != 0) return Query(0, e) - Query(0, b);
    ll ans = 0;
    for (int pos = e; pos; pos -= (pos & -pos))
      ans += T1[pos] * (e - 1) - T2[pos];
    return ans;
  }
};
```

FenwickTree2D.h

Description: Computes sums $a[i, j]$ for all $i < x$, $j < y$, and increases single elements $a[x, y]$. Requires that the elements to be updated are known in advance (call FakeUpdate() before Build()).

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

340c18, 33 lines

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

  void FakeUpdate(int x, int y) {
    for (++x; x < (int)ys.size(); x += (x & -x))
      ys[x].push_back(y);
  }
  void Build() {
    for (auto& v : ys) {
      sort(v.begin(), v.end());
      v.erase(unique(v.begin(), v.end()), v.end());
      T.emplace_back(v.size() + 1);
    }
  }
  int ind(int x, int y) {
    auto it = upper_bound(ys[x].begin(), ys[x].end(), y);
    return it - ys[x].begin();
  }
  void Update(int x, int y, int val) {
    for (++x; x < (int)ys.size(); x += (x & -x))
      for (int i = ind(x,y); i < (int)T[x].size(); i += (i & -i))
        T[x][i] = T[x][i] + val;
  }
  int Query(int x, int y) {
    int sum = 0;
    for (; x > 0; x -= (x & -x))
      for (int i = ind(x,y); i > 0; i -= (i & -i))
        sum = sum + T[x][i];
    return sum;
  }
};
```

SkewHeap.h

Description: Easy to implement and fast meldable heap.

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

b24fbb, 28 lines

```cpp
struct SkewHeap {
  struct Node { ll key, lazy = 0; int l = -1, r = -1; };
  vector<Node> T;

  void push(int x) {
    if (x == -1 || !T[x].lazy) return;
    for (int y : {T[x].l, T[x].r}) if (y != -1)
      T[y].lazy += T[x].lazy;
```

```
      T[x].key += T[x].lazy, T[x].lazy = 0;
  }
  // Make new node. Returns its index. Indexes go 0, 1, ...
  int New(ll key) {
    T.push_back(Node{key});
    return (int)T.size() - 1;
  }
  // Increment all values in heap p by v
  void Add(int x, ll v) { if (~x) T[x].lazy += v, push(x); }
  // Merge heaps a and b
  int Merge(int a, int b) {
    if (b == -1 || a == -1) return a + b + 1;
    if (T[a].key > T[b].key) swap(a, b);
    int &l = T[a].l, &r = T[a].r;
    push(r); swap(l, r); l = Merge(l, b);
    return a;
  }
  void Pop(int& x) { x = Merge(T[x].l, T[x].r); }
  ll Get(int x) { return T[x].key; }
};
```

Treap.h
Description: A short self-balancing tree. It acts as a sequential container with log-time splits/joins, and is easy to augment with additional data. It can support several operations, including lazy propagation (sample flip operation below). Can be made persistent, by making a copy at split/join functions.
Time: $\mathcal{O}(\log N)$ expected time per operation
WARNING: If making persistent, think twice if you need to ditch priorities.
19a060, 53 lines

```
mt19937 rng(time(0));

struct Treap {
  struct Node {
    int l = 0, r = 0, sz = 0, pri = 0; // required
    ll key = 0; bool flip = 0;         // optional
  };
  vector<Node> T;

  Treap(int n) : T(n + 1) {
    for (int i = 1; i <= n; ++i)
      T[i].pri = rng(), pull(i);
  }
  void pull(int x) {
    if (!x) return;
    int l = T[x].l, r = T[x].r;
    T[x].sz = T[l].sz + T[r].sz + 1;
  }
  void push(int x) {
    if (!x) return;
    int l = T[x].l, r = T[x].r;
    if (T[x].flip) {
      swap(T[x].l, T[x].r);
      T[l].flip ^= 1, T[r].flip ^= 1;
      T[x].flip = 0;
    }
  }
  pair<int, int> Split(int x, auto&& is_lhs) {
    if (!x) return {0, 0};
    push(x); int l, r;
    if (is_lhs(x))
      tie(T[x].r, r) = Split(T[x].r, is_lhs), l = x;
    else
      tie(l, T[x].l) = Split(T[x].l, is_lhs), r = x;
    pull(x); return {l, r};
  }
  pair<int, int> SplitByKey(int x, ll key) {
    return Split(x, [&](int y) { return T[y].key < key; });
```

```
  }
  pair<int, int> SplitByKth(int x, int k) {
    return Split(x, [&](int y) {
      int sz = T[T[y].l].sz + 1;
      return k >= sz ? (k -= sz, 1) : 0;
    });
  }
  int Join(int l, int r) {
    if (!l || !r) return l + r;
    push(l); push(r);
    if (T[l].pri < T[r].pri)
      return T[l].r = Join(T[l].r, r), pull(l), l;
    return T[r].l = Join(l, T[r].l), pull(r), r;
  }
};
```

SplayTree.h
Description: Self-adjusting balanced BST. Nodes are 1-indexed for performance reasons.
Time: $\mathcal{O}(\log N)$ per operation
6b2e59, 47 lines

```
struct SplayTree {
  struct Node {
    int ch[2] = {0, 0}, p = 0;
    ll self = 0, path = 0;                // Path aggregates
    ll sub = 0, vir = 0;                  // Subtree aggregates
    bool flip = 0;                        // Lazy tags
  };
  vector<Node> T;

  SplayTree(int n) : T(n + 1) {
    for (int i = 1; i <= n; ++i) pull(i);
  }
  void push(int x) {
    if (!x || !T[x].flip) return;
    int l = T[x].ch[0], r = T[x].ch[1];
    T[l].flip ^= 1, T[r].flip ^= 1;
    swap(T[x].ch[0], T[x].ch[1]);
    T[x].flip = 0;
  }
  void pull(int x) {
    int l = T[x].ch[0], r = T[x].ch[1]; push(l); push(r);
    T[x].path = T[l].path + T[x].self + T[r].path;
    T[x].sub = T[x].vir + T[l].sub + T[r].sub + T[x].self;
  }
  void set(int x, int d, int y) {
    T[x].ch[d] = y; T[y].p = x; pull(x);
  }
  void splay(int x) {
    auto dir = [&](int x) {
      int p = T[x].p; if (!p) return -1;
      return T[p].ch[0] == x ? 0 : T[p].ch[1] == x ? 1 : -1;
    };
    auto rotate = [&](int x) {
      int y = T[x].p, z = T[y].p, dx = dir(x), dy = dir(y);
      set(y, dx, T[x].ch[!dx]); set(x, !dx, y);
      if (~dy) set(z, dy, x);
      T[x].p = z;
    };
    for (push(x); ~dir(x); ) {
      int y = T[x].p, z = T[y].p;
      push(z); push(y); push(x);
      int dx = dir(x), dy = dir(y);
      if (~dy) rotate(dx != dy ? x : y);
      rotate(x);
    }
  }
};
```

Numerical (4)

4.1 Integrate

Integrate.h
Description: Simple integration of a function over an interval using Simpson's rule. According to cp-algorithms, the error is $-\frac{1}{90}(\frac{b-a}{2})^5 f^{(4)}(\xi)$ for some $\xi \in [a, b]$.
5a2754, 9 lines

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

IntegrateAdaptive.h
Description: Fast integration using an adaptive Simpson's rule.
Usage: double z, y;
double h(double x) { return x*x + y*y + z*z <= 1; }
double g(double y) { ::y = y; return Quad(h, -1, 1); }
double f(double z) { ::z = z; return Quad(g, -1, 1); }
double sphere_vol = Quad(f, -1, 1), pi = sphere_vol * 3 / 4;
e54d01, 18 lines

```
template<typename Func>
double recurse(Func f, double a, double b,
               double eps, double S) {
  auto simpson = [&](double a, double b) {
    return (f(a) + 4 * f((a + b) / 2) + f(b)) * (b - a) / 6; };
  double c = (a + b) / 2;
  double S1 = simpson(a, c), S2 = simpson(c, b);
  double T = S1 + S2;
  if (abs(T - S) < 15 * eps || b - a < 1e-10)
    return T + (T - S) / 15;
  return recurse(f, a, c, eps / 2, S1) +
         recurse(f, c, b, eps / 2, S2);
}

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

4.2 Optimization

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: A = {{1,-1}, {-1,1}, {-1,-2}};
b = {1,1,-4}, c = {-1,-1}, x;
T val = LPSolver(A, b, c).Solve(x);
Time: $\mathcal{O}(NM * \#pivots)$, where a pivot may be e.g. an edge relaxation. $\mathcal{O}(2^n)$ in the general case.
cb2a3b, 62 lines

```
using T = double; // long double, Rational, double + mod<P>...
using vec = vector<T>;
using mat = vector<vec>;

const T EPS = 1e-8, INF = 1/.0;
#define MT make_tuple
#define FOR(i, a, b) for(int i = (a); i < (b); ++i)

double Simplex(mat A, vec b, vec c, vec& x) {
```

```
int m = b.size(), n = c.size();
vector<int> N(n + 1), B(m);
mat D(m + 2, vec(n + 2));

FOR (i, 0, m) FOR (j, 0, n) D[i][j] = A[i][j];
FOR (i, 0, m) B[i] = n + i, D[i][n] = -1, D[i][n + 1] = b[i];
FOR (j, 0, n) N[j] = j, D[m][j] = -c[j];
N[n] = -1, D[m + 1][n] = 1;

auto pivot = [&](int r, int s) {
  vec& a = D[r]; T inv = 1. / a[s];
  FOR (i, 0, m + 2) if (i != r && abs(D[i][s]) > EPS) {
    vec& b = D[i]; T inv2 = b[s] * inv;
    FOR (j, 0, n + 2) b[j] -= a[j] * inv2;
    b[s] = a[s] * inv2;
  }
  FOR (j, 0, n + 2) if (j != s) D[r][j] *= inv;
  FOR (i, 0, m + 2) if (i != r) D[i][s] *= -inv;
  D[r][s] = inv;
  swap(B[r], N[s]);
};
auto simplex = [&](int phase) {
  int x = m + phase - 1;
  while (true) {
    auto T = MT(INF, -1, -1);
    FOR (j, 0, n + 1) if (N[j] != -phase)
      T = min(T, MT(D[x][j], N[j], j));
    if (get<0>(T) > -EPS) return true;
    int s = get<2>(T); T = MT(INF, -1, -1);
    FOR (i, 0, m) {
      if (D[i][s] < EPS) continue;
      T = min(T, MT(D[i][n + 1] / D[i][s], B[i], i));
    }
    int r = get<2>(T);
    if (r == -1) return false;
    pivot(r, s);
  }
};
int r = 0;
FOR (i, 1, m) if (D[i][n + 1] < D[r][n + 1]) r = i;
if (D[r][n + 1] < -EPS) {
  pivot(r, n);
  if (!simplex(2) || D[m + 1][n + 1] < -EPS) return -INF;
  FOR (i, 0, m) if (B[i] == -1) {
    auto T = MT(INF, -1, -1);
    FOR (j, 0, n + 1) T = min(T, MT(D[i][j], N[j], j));
    pivot(i, get<2>(T));
  }
}
bool ok = simplex(1); x.assign(n, 0);
FOR (i, 0, m) if (B[i] < n) x[B[i]] = D[i][n + 1];
return ok ? D[m][n + 1] : INF;
}
```

4.2.1 LP dualization

To dualize a LP, add one dual variable for each primal constraint. Function to optimize becomes $b^T y$. Then, you have one dual constraint for each primal variable, given by the primal constraints in which they appear.

Primal	Dual
max $c^T x$ s.t. $Ax \le b, x \ge 0$	min $b^T y$ s.t. $A^T y \ge c, y \ge 0$
max $c^T x$ s.t. $Ax \le b$	min $b^T y$ s.t. $A^T y = c, y \ge 0$
max $c^T x$ s.t. $Ax = b, x \ge 0$	min $b^T y$ s.t. $A^T y \ge c$

If primal is not unbounded nor infeasible, then neither the dual is, and strong duality holds (i.e., $c^T x^* = b^T y^*$).

4.3 Matrices

RowEchelon.h
Description: Converts matrix to reduced row-echelon form. Each of the first r rows will have exactly one non-zero column piv_i, and all columns to the left of piv_i will be zero. To compute rank, do RowEchelon(M).size().
Time: $\mathcal{O}\left(M^2 N\right)$
604f5c, 22 lines

```
using ld = double;
using mat = vector<vector<ld>>;
const ld EPS = 1e-9;

pair<vector<int>, int> RowEchelon(mat& A) {
  int n = A.size(), m = A[0].size(), sgn = 1;
  vector<int> piv;
  for (int i = 0, rnk = 0; i < m && rnk < n; ++i) {
    for (int j = rnk + 1; j < n; ++j)
      if (abs(A[j][i]) > abs(A[rnk][i]))
        swap(A[j], A[rnk]), sgn = -sgn;
    if (abs(A[rnk][i]) < EPS) continue;
    for (int j = 0; j < n; ++j) {
      ld coef = A[j][i] / A[rnk][i];
      if (j == rnk || abs(coef) < EPS) continue;
      for (int k = 0; k < m; ++k)
        A[j][k] -= coef * A[rnk][k];
    }
    piv.push_back(i); ++rnk;
  }
  return {piv, sgn};
}
```

SolveLinear.h
Description: Solves $Mx = b$. Set A to be the block matrix $[M|b]$. If there are multiple solutions, returns a sol which has all free variables set to 0.
Time: $\mathcal{O}\left(M^2 N\right)$
"RowEchelon.h"
e11534, 9 lines

```
vector<ld> SolveLinear(mat& A) {
  int m = A[0].size() - 1;
  auto piv = RowEchelon(A).first;
  if (!piv.empty() && piv.back() == m) return {};
  vector<ld> sol(m, 0.);
  for (int i = 0; i < (int)piv.size(); ++i)
    sol[piv[i]] = A[i][m] / A[i][i];
  return sol;
}
```

MatrixDeterminant.h
Description: Computes the determinant of $N \times N$ matrix.
Time: $\mathcal{O}\left(N^3\right)$
"RowEchelon.h"
6cb303, 7 lines

```
ld Determinant(mat& A) {
  int n = A.size();
  ld det = RowEchelon(A).second;
  for (int i = 0; i < n; ++i)
    det *= A[i][i];
  return det;
}
```

MatrixInverse.h
Description: Computes the inverse of an $N \times N$ matrix. Returns true if successful. For inverse modulo prime powers, repeatedly set $A^{-1} = A^{-1}(2I - AA^{-1}) \pmod{p^k}$ where A^{-1} starts as the inverse of A modulo p, and k is doubled in each step.
Time: $\mathcal{O}\left(N^3\right)$
"RowEchelon.h"
cc3bf4, 15 lines

```
bool Invert(mat& A) {
  int n = A.size(); // assert(n == A[0].size());
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      A[i].push_back(i == j);
  auto [piv, sgn] = RowEchelon(A);
  if ((int)piv.size() < n) return false;
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      // For adjunct, do A[i][j + n] *= sgn instead.
      A[i][j + n] /= A[i][i];
    A[i].erase(A[i].begin(), A[i].begin() + n);
  }
  return true;
}
```

4.3.1 Matrix determinant lemma

Suppose A is an invertible square matrix and u, v are column vectors. Then:

$$det(A + uv^T) = (1 + v^T A^{-1} u)det(A) = det(A) + v^T adj(A)u$$

Here, uv^T is the outer product of u and v. The first and third expressions are equal even when the square matrix A is not invertible.

4.4 Search

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. Works equally well for maximization with a small change in the code. See TernarySearch.h in the Various chapter for a discrete version.
Usage: double func(double x) { return 4+x+.3*x*x; }
double xmin = GoldenSectionSearch(-1000,1000,func);
Time: $\mathcal{O}\left(\log((b - a)/\epsilon)\right)$
909fb9, 15 lines

```
template<typename Func>
double GoldenSectionSearch(double a, double b, Func f) {
  double r = (sqrt(5) - 1) / 2;
  double x1 = b - r * (b - a), x2 = a + r * (b - a);
  double f1 = f(x1), f2 = f(x2);
  for (int it = 0; it < 64; ++it)
    if (f1 < f2) { // change to > to find maximum
      b = x2; x2 = x1; f2 = f1;
      x1 = b - r * (b - a); f1 = f(x1);
    } else {
      a = x1; x1 = x2; f1 = f2;
      x2 = a + r * (b - a); f2 = f(x2);
    }
  return (a + b) / 2;
}
```

TernarySearch.h
Description: Find the smallest i in $[a, b)$ that maximizes $f(i)$, assuming that $f(a) < \ldots < f(i) \ge \cdots \ge f(b)$. To reverse which of the sides allows non-strict inequalities, change the $<$ marked with (A) to $<=$, and reverse the loop at (B). To minimize f, change it to $>$, also at (B).
Usage: int ind = TernarySearch(0,n,[&](int i){return a[i];});
Time: $\mathcal{O}\left(\log(b - a)\right)$
0f6f0d, 12 lines

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

```
  for (int i = a + 1; i < b; ++i)
    if (f(a) < f(i)) a = i; // (B)
  return a;
}
```

FracBinarySearch.h
Description: Finds the smallest fraction $p/q \in [0, 1]$ such that $f(p/q)$ is true, and $p \leq maxp, q \leq maxq$. You may want to throw an exception from f if it finds an exact solution, in which case $maxp, maxq$ can be removed.
Time: $\mathcal{O}(\log(maxq))$
<div align="right">6eadbb, 24 lines</div>

```
using ll = long long;
struct Frac { ll p, q; };

template<class Func>
Frac FracLowerBound(Func f, ll maxq, ll maxp) {
  Frac lo{0, 1}, hi{1, 1}, mid; // Set hi to 1/0 for (0, inf)
  if (f(lo)) return lo;
  assert(f(hi));
  for (int it = 0, dir = 1; it < 3 || !dir; ++it) {
    // invariant: f(lo) == !dir, f(hi) == dir
    for (ll step = 1, adv = 1, now = 0; ;
         adv ? step *= 2 : step /= 2) {
      now += step;
      mid = {lo.p * now + hi.p, lo.q * now + hi.q};
      if (abs(mid.p) > maxp || mid.q > maxq || f(mid) != dir)
        now -= step, adv = 0;
      if (!step) break;
    }
    if (mid.p != hi.p) it = 0;
    hi = lo; lo = mid; dir = !dir;
  }
  // (lo, hi) are consecutive with f(lo) == 0, f(hi) == 1
  return hi;
}
```

4.5 Fourier transforms

FFT.h
Description: Cooley-Tukey-like fft. For better precision, see "NTT.h".
Time: $\mathcal{O}(N \log N)$
<div align="right">72a787, 14 lines</div>

```
void DFT(vector<complex<double>> &a, bool rev) {
  int n = a.size(); auto b = a;
  for (int step = n / 2; step; step /= 2) {
    for (int i = 0; i < n / 2; i += step) {
      auto wn = polar(1.0, 2.0 * M_PI * (rev ? -i : i) / n);
      for (int j = 0; j < step; ++j) {
        auto u = a[i * 2 + j], v = wn * a[i * 2 + j + step];
        b[i + j] = u + v; b[i + n / 2 + j] = u - v;
      }
    }
    swap(a, b);
  }
  if (rev) for (auto& x : a) x /= n;
}
```

NTT.h
Description: Cooley-Tukey-like ntt. Discrete Fourier modulo specific nice primes of the form $2^a b + 1$, where the convolution result has size at most 2^a. Takes around 100ms on $N = 5 \cdot 10^5$ on library-checker.
Time: $\mathcal{O}(N \log N)$
"../number-theory/ModInt.h"
<div align="right">4fcccc, 22 lines</div>

```
// Good MOD: (119 << 23 + 1), (5 << 25 + 1), (5LL << 55 + 1)
void DFT(vector<ModInt> &a, bool rev) {
  int n = a.size(); auto b = a; // assert(!(n & (n - 1)));
  ModInt g = 1; while (g.pow((MOD - 1) / 2) == 1) g = g + 1;
  if (rev) g = g.inv();
```

```
  for (int step = n / 2; step; step /= 2) {
    ModInt w = g.pow((MOD - 1) / (n / step)), wn = 1;
    for (int i = 0; i < n / 2; i += step) {
      for (int j = 0; j < step; ++j) {
        auto u = a[2 * i + j], v = wn * a[2 * i + j + step];
        b[i + j] = u + v; b[i + n / 2 + j] = u - v;
      }
      wn = wn * w;
    }
    swap(a, b);
  }
  if (rev) {
    auto n1 = ModInt(n).inv();
    for (auto& x : a) x = x * n1;
  }
}
```

FST.h
Description: Transform to a basis with fast convolutions of the form $c[x \oplus y] \mathrel{+}= a[x] \cdot b[y]$, where $\oplus$ is one of AND, OR, XOR. The size of a must be a power of two.
Time: $\mathcal{O}(N \log N)$
<div align="right">eb980e, 13 lines</div>

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

SubsetConv.h
Description: Neat trick to compute $c[x|y] \mathrel{+}= a[x] \cdot b[y]$, where x and y satisfy $x \& y = 0$. Use with "OR" FST transform. The idea is to do FST convolutions and discard the results unless the number of bits of the result equals the sums of the bits of the multiplicants. The idea can probably be adapted to other similar problems.
Time: $\mathcal{O}(N \log^2 N)$ (around 2s for $N = 2^{20}$)
"FST.h"
<div align="right">f85131, 23 lines</div>

```
using Poly = vector<int>;

Poly SubsetConv(Poly a, Poly b) {
  int n = a.size(), lg = 31 - __builtin_clz(n);
  assert(n == (1 << lg));
  vector<Poly> p(lg + 1, Poly(n, 0)), q(p);
  for (int i = 0; i < n; ++i) {
    int lev = __builtin_popcount(i);
    p[lev][i] = a[i]; q[lev][i] = b[i];
  }
  for (int i = 0; i <= lg; ++i) DFT(p[i], 0), DFT(q[i], 0);
  for (int i = 0; i <= lg; ++i) {
    fill(b.begin(), b.end(), 0);
    for (int j = 0; j <= i; ++j)
      for (int k = 0; k < n; ++k)
        b[k] = b[k] + p[j][k] * q[i - j][k];
    DFT(b, 1);
    for (int j = 0; j < n; ++j)
      if (__builtin_popcount(j) == i)
        a[j] = b[j];
  }
  return a;
}
```

4.6 Polynomials

PolyInterpolate.h
Description: Given n points $(x[i], y[i])$, computes an $n-1$-degree polynomial p that passes through them: $p(x) = a[0] * x^0 + \ldots + a[n-1] * x^{n-1}$. For numerical precision, pick $x[k] = c * \cos(k/(n-1) * \pi), k = 0 \ldots n-1$.
Time: $\mathcal{O}(n^2)$
<div align="right">cd5077, 16 lines</div>

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

PolyMul.h
Description: Computes $c[x+y] \mathrel{+}= a[x] \cdot b[y]$. To optimize, consider having extra logic for the case where a == b.
Time: $\mathcal{O}(N + M \log(N + M))$
"NTT.h"
<div align="right">f3c7bf, 11 lines</div>

```
vector<ModInt> Mul(vector<ModInt> a, vector<ModInt> b) {
  if (a.empty() || b.empty()) return {};
  int m = a.size() + b.size() - 1, n = m;
  while (n & (n - 1)) ++n;
  a.resize(n, 0); b.resize(n, 0);
  DFT(a, 0); DFT(b, 0);
  for (int i = 0; i < n; ++i)
    a[i] = a[i] * b[i];
  DFT(a, 1); a.resize(m);
  return a;
}
```

PolyInv.h
Description: Computes $\bar{a}$ s.t. $deg(\bar{a}) = n, a\bar{a} = 1 \pmod{X^n}$.
Time: $\mathcal{O}(N \log N)$
"NTT.h"
<div align="right">ab81dc, 14 lines</div>

```
vector<ModInt> Inv(vector<ModInt> a, int n) {
  vector<ModInt> ret(1, a[0].inv()), tmp;
  for (auto& x : a) x = x * (-1);
  for (int step = 2; step < n * 2; step *= 2) {
    tmp = a; tmp.resize(4 * step, 0); ret.resize(4 * step, 0);
    DFT(tmp, 0); DFT(ret, 0);
    for (int i = 0; i < 4 * step; ++i)
      ret[i] = (ret[i] * tmp[i] + 2) * ret[i];
    DFT(ret, 1);
    ret.resize(step);
  }
  ret.resize(n);
  return ret;
}
```

PolyDivRem.h
Description: Fast polynomial division/remainder.
Time: $\mathcal{O}(N \log N)$
"PolyMul.h", "PolyInv.h"
<div align="right">0c818d, 18 lines</div>

```
vector<ModInt> Div(vector<ModInt> a, vector<ModInt> b) {
  int d = (int)a.size() - (int)b.size() + 1;
  if (d <= 0) return {};
```

```
  reverse(a.begin(), a.end()); reverse(b.begin(), b.end());
  a.resize(d); b.resize(d);
  auto ret = Mul(a, Inv(b, d));
  ret.resize(d);
  reverse(ret.begin(), ret.end());
  return ret;
}

vector<ModInt> Rem(vector<ModInt> a, vector<ModInt> b) {
  auto sub = Mul(Div(a, b), b);
  for (int i = 0; i < (int)sub.size(); ++i)
    a[i] = a[i] - sub[i];
  while (a.size() && a.back() == 0) a.pop_back();
  return a;
}
```

PolyRoots.h
Description: Durand-Kerner method of finding all roots of polynomial. If polynomial has real coefficients, it might make more sense to initialize roots with conjugate pairs (and potentially one real root), see (*). It might not converge for all polynomials and all sets of initial roots.
Time: $\mathcal{O}\left(N^2\right)$ per iteration

44d01a, 20 lines

```
using C = complex<double>;

vector<C> FindRoots(vector<C> p) {
  int n = p.size() - 1;
  vector<C> ret(n);
  for (int i = 0; i < n; ++i)
    ret[i] = pow(C{0.456, 0.976}, i); // (*)
  for (int it = 0; it < 1000; ++it) {
    for (int i = 0; i < n; ++i) {
      C up = 0, dw = 1;
      for (int j = n; j >= 0; --j) {
        up = up * ret[i] + p[j];
        if (j != i && j != n)
          dw = dw * (ret[i] - ret[j]);
      }
      ret[i] -= up / dw / p[n];
    }
  }
  return ret;
}
```

4.7 Recurrences

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

"../number-theory/ModInt.h"   85f0eb, 21 lines

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

  auto b = C[0]; int L = 0;
  for (int i = 0, m = 1; i < n; ++i) {
    auto d = s[i];
    for (int j = 1; j <= L; ++j)
      d = d + C[j] * s[i - j];
    if (d == 0) { ++m; continue; }
    auto T = C; auto coef = d / b;
    for (int j = m; j < n; ++j)
      C[j] = C[j] - coef * B[j - m];
    if (2 * L > i) { ++m; continue; }
```

```
    L = i + 1 - L; B = T; b = d; m = 1;
  }
  C.resize(L + 1); C.erase(C.begin());
  for (auto& x : C) x = x * (-1);
  return C;
}
```

LinearRecurrence.h
Description: Generates the k-th term of a n-th order linear recurrence given the first n terms and the recurrence relation. Faster than matrix multiplication. Useful to use along with Berlekamp Massey. Recurrence is $s_i = \sum_{j=0}^{n-1} s_{i-j-1} * trans_j$ where $first = \{s_0, s_1, ..., s_{n-1}\}$
Usage: `LinearRec({0, 1}, {1, 1}, k)` gives k-th Fibonacci number (0-indexed)
Time: $\mathcal{O}\left(N^2 log(K)\right)$ per query

6c6575, 27 lines

```
using Poly = vector<ModInt>;

ModInt LinearRec(Poly first, Poly trans, int k) {
  int n = trans.size(); // assert(n <= (int)first.size());
  Poly r(n + 1, 0), b(r); r[0] = b[1] = 1;
  auto ans = b[0];

  auto combine = [&](Poly a, Poly b) { // a * b mod trans
    Poly res(n * 2 + 1, 0);
    for (int i = 0; i <= n; ++i)
      for (int j = 0; j <= n; ++j)
        res[i + j] = res[i + j] + a[i] * b[j];
    for (int i = 2 * n; i > n; --i)
      for (int j = 0; j < n; ++j)
        res[i - 1 - j] = res[i - 1 - j] + res[i] * trans[j];
    res.resize(n + 1);
    return res;
  };
  // Consider caching the powers for multiple queries
  for (++k; k; k /= 2) {
    if (k % 2) r = combine(r, b);
    b = combine(b, b);
  }
  for (int i = 0; i < n; ++i)
    ans = ans + r[i + 1] * first[i];
  return ans;
}
```

4.7.1 Char. polynomial
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. $a_n = (d_1 n + d_2) r^n$.

Number theory (5)

5.1 Modular arithmetic

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

0214fc, 16 lines

```
const int MOD = 17; // change to something else
struct ModInt {
  int x;
  ModInt(int x = 0) : x(x + (x < 0) * MOD - (x >= MOD) * MOD){}
  ModInt operator+(ModInt o) { return x + o.x; }
  ModInt operator-(ModInt o) { return x - o.x; }
```

```
  ModInt operator*(ModInt o) { return 1LL * x * o.x % MOD; }
  ModInt operator/(ModInt b) { return *this * b.inv(); }
  ModInt inv() { return pow(MOD - 2); }
  ModInt pow(long long e) {
    if (!e) return 1;
    ModInt r = pow(e / 2); r = r * r;
    return e % 2 ? *this * r : r;
  }
  bool operator==(ModInt o) { return x == o.x; }
};
```

ModMulLL.h
Description: Calculate $a \cdot b \mod c$ (or $a^b \mod c$) for $0 \leq a, b \leq c \leq 7.2 \cdot 10^{18}$.
Time: $\mathcal{O}(1)$ for modmul, $\mathcal{O}(\log b)$ for modpow

a34b1e, 12 lines

```
typedef unsigned long long ull;

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

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

fd2931, 14 lines

```
using ll = long long;

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

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

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

"Euclid.h"   9a16de, 17 lines

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

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

```
    if (ret.second == -1) break;
  }
  return ret;
}
```

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.   fa90e3, 21 lines

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

ull SumSq(ull to) { return to / 2 * ((to-1) | 1); }

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

ll ModSum(ull to, ll c, ll k, ll m) {
  c %= m; if (c < 0) c += m;
  k %= m; if (k < 0) k += m;
  return to * c + k * SumSq(to) - m * DivSum(to, c, k, m);
}
```

ModLog.h
Description: Returns the smallest $x > 0$ s.t. $a^x = b \pmod{m}$, or -1 if no such x exists. modLog(a,1,m) can be used to calculate the order of a.
Time: $\mathcal{O}\left(\sqrt{m}\right)$ 4fbb86, 12 lines

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

ModSqrt.h
Description: Tonelli-Shanks algorithm for modular square roots. Finds x s.t. $x^2 = a \pmod{p}$ ($-x$ gives the other solution).
Time: $\mathcal{O}\left(\log^2 p\right)$ worst case, $\mathcal{O}\left(\log p\right)$ for most p
"ModPow.h"   19a793, 24 lines

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

```
  for (;; r = m) {
    ll t = b;
    for (m = 0; m < r && t != 1; ++m)
      t = t * t % p;
    if (m == 0) return x;
    ll gs = modpow(g, 1LL << (r - m - 1), p);
    g = gs * gs % p;
    x = x * gs % p;
    b = b * g % p;
  }
}
```

5.2 Primality

MillerRabin.h
Description: Deterministic Miller-Rabin primality test. Guaranteed to work for numbers up to $7 \cdot 10^{18}$; for larger numbers, use Python and extend A randomly.
Time: 7 times the complexity of $a^b \mod c$.
"ModMulLL.h"   583580, 12 lines

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

Factor.h
Description: Pollard-rho randomized factorization algorithm. Returns prime factors of a number, in arbitrary order (e.g. 2299 -> {11, 19, 11}).
Time: $\mathcal{O}\left(n^{1/4}\right)$, less for numbers with small factors.
"ModMulLL.h", "MillerRabin.h"   b98062, 20 lines

```
ull Pollard(ull n) {
  auto f = [&](ull x) { return modmul(x, x, n) + 1; };
  ull x = 0, y = 0, t = 30, prd = 2, i = 1, q;
  while (t++ % 40 || __gcd(prd, n) == 1) {
    if (x == y) x = ++i, y = f(x);
    q = modmul(prd, max(x, y) - min(x, y), n);
    if (q) prd = q;
    x = f(x), y = f(f(y));
  }
  return __gcd(prd, n);
}

vector<ull> Factor(ull n) {
  if (n == 1) return {};
  if (IsPrime(n)) return {n};
  ull x = Pollard(n);
  auto l = Factor(x), r = Factor(n / x);
  l.insert(l.end(), r.begin(), r.end());
  return l;
}
```

5.3 Divisibility

5.3.1 Bézout's identity

For $a \neq 0$, $b \neq 0$, 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

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

5.4 Pythagorean Triples

The Pythagorean triples are uniquely generated by

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

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

5.5 Primes

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

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

5.6 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 Mobius Function

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

Mobius Inversion:

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

Other useful formulas/forms:

$\sum_{d|n} \mu(d) = [n = 1]$ (very useful)

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

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

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

IntPerm.h

Description: Permutation -> integer conversion. (Not order preserving.) Integer -> permutation can use a lookup table.

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

044568, 6 lines

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

6.1.2 Cycles

Let $g_S(n)$ be the number of n-permutations whose cycle lengths all belong to the set S. Then

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

6.1.3 Derangements

Permutations of a set such that none of the elements appear in their original position.

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

6.1.4 Burnside's lemma

Given a group G of symmetries and a set X, the number of elements of X *up to symmetry* equals

$$\frac{1}{|G|} \sum_{g \in G} |X^g|,$$

where X^g are the elements fixed by g ($g.x = x$).

If $f(n)$ counts "configurations" (of some sort) of length n, we can ignore rotational symmetry using $G = \mathbb{Z}_n$ to get

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

6.2 Partitions and subsets

6.2.1 Partition function

Number of ways of writing n as a sum of positive integers, disregarding the order of the summands.

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

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

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' 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

multinomial.h

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

a0a312, 6 lines

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

6.3 General purpose numbers

6.3.1 Bernoulli numbers

EGF of Bernoulli numbers is $B(t) = \frac{t}{e^t - 1}$ (FFT-able).

$B[0, \ldots] = [1, -\frac{1}{2}, \frac{1}{6}, 0, -\frac{1}{30}, 0, \frac{1}{42}, \ldots]$

Sums of powers:

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

Euler-Maclaurin formula for infinite sums:

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

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

6.3.2 Stirling numbers of the first kind

Number of permutations on n items with k cycles.

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

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

6.3.3 Eulerian numbers

Number of permutations $\pi \in S_n$ in which exactly k elements are greater than the previous element. k j:s s.t. $\pi(j) > \pi(j+1)$, $k+1$ j:s s.t. $\pi(j) \geq j$, k j:s s.t. $\pi(j) > j$.

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

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

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

6.3.4 Stirling numbers of the second kind

Partitions of n distinct elements into exactly k groups.

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

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

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

6.3.5 Bell numbers

Total number of partitions of n distinct elements. $B(n) = 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, \ldots$. For p prime,

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

6.3.6 Labeled unrooted trees

\# on n vertices: n^{n-2}
\# on k existing trees of size n_i: $n_1 n_2 \cdots n_k n^{k-2}$
\# with degrees d_i: $(n-2)!/((d_1 - 1)! \cdots (d_n - 1)!)$

6.3.7 Catalan numbers

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

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

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

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

Graph (7)

7.1 General

SCC.h

Description: Finds strongly connected components in a directed graph. If vertices u, v belong to the same component, we can reach u from v and vice versa.

Usage: SCC(graph, [&](auto& v) { ... }) visits all components in reverse topological order.

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

1deb0c, 22 lines

```
template<class CB>
void SCC(vector<vector<int>>& graph, CB cb) {
  int n = graph.size(), timer = 0;
  vector<int> val(n, 0), stk, cont;

  function<int(int)> dfs = [&](int node) {
    int low; low = val[node] = ++timer;
    int sz = stk.size(); stk.push_back(node);
```

```
    for (auto vec : graph[node])
        if (val[vec] != -1)
            low = min(low, val[vec] ?: dfs(vec));
    if (low == val[node]) {
        cont = {stk.begin() + sz, stk.end()};
        for (auto x : cont) val[x] = -1;
        cb(cont); stk.resize(sz);
    } else val[node] = low;
    return low;
};
for (int i = 0; i < n; ++i)
    if (val[i] != -1)
        dfs(i);
}
```

BCC.h

Description: Finds all biconnected components in an undirected multi-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. HOWEVER, note that we are outputting bridges as BCC's here, because we might be interested in vertex bcc's, not edge bcc's.

To get the articulation points, look for vertices that are in more than 1 BCC.
To get the bridges, look for biconnected components with one edge
Time: $\mathcal{O}(E + V)$ 3c8daa, 30 lines

```
template<typename CB>
void BCC(vector<vector<pair<int, int>>>& graph, CB&& cb) {
    int timer = 0, n = graph.size();
    vector<int> val(n, -1);
    vector<tuple<int, int, int>> stk, cont;

    function<int(int, int)> dfs = [&](int node, int pei) {
        int ret; ret = val[node] = timer++;
        for (auto [vec, ei] : graph[node]) {
            if (ei == pei) continue;
            if (val[vec] != -1) {
                ret = min(ret, val[vec]);
                if (val[vec] < val[node])
                    stk.emplace_back(node, vec, ei);
            } else {
                int sz = stk.size(), low = dfs(vec, ei);
                ret = min(ret, low);
                stk.emplace_back(node, vec, ei);
                if (low >= val[node]) {
                    cont = {stk.begin() + sz, stk.end()};
                    cb(cont); stk.resize(sz);
                }
            }
        }
        return ret;
    };
    for (int i = 0; i < n; ++i)
        if (val[i] == -1)
            dfs(i, -1);
}
```

2SAT.h

Description: Calculates a valid assignment to boolean variables a, b, c,... to a 2-SAT problem, so that an expression of the type $(a \lor b) \land (!a \lor c) \land (d \lor !b) \land \ldots$ becomes true, or reports that it is unsatisfiable. Returns empty vector if no solution. Negated variables are represented by bit-inversions (~x).
Usage: TwoSat sat(4); // number of variables
sat.Either(0, ~3); // Var 0 is true or var 3 is false
sat.SetValue(2); // Var 2 is true
sat.AtMostOne({0,~1,2}); // <= 1 of vars 0, ~1 and 2 are true
sat.Solve(); // Returns solution

Time: $\mathcal{O}(N + E)$, where N is the number of boolean variables, and E is the number of clauses.
<../data-structures/BiVec.h>   0c6cb5, 53 lines

```
struct TwoSat {
    int n;
    BiVec<vector<int>> graph;

    TwoSat(int n) : n(n), graph(n) {}

    void Implies(int a, int b) {
        graph[a].push_back(b);
        graph[~b].push_back(~a);
    }
    void Either(int a, int b) { Implies(~a, b); }
    void SetValue(int x) { Either(x, x); }
    int AddVar() { graph.resize(++n); return n - 1; }
    void AtMostOne(const vector<int>& vals) {
        if (vals.size() <= 1) return;
        int cur = ~vals[0];
        for (int i = 2; i < (int)vals.size(); ++i) {
            int nxt = AddVar();
            Either(cur, ~vals[i]);
            Either(cur, nxt);
            Either(~vals[i], nxt);
            cur = ~nxt;
        }
        Either(cur, ~vals[1]);
    }
    vector<int> Solve() {
        BiVec<int> vis(n); vector<int> stk, ans(n);
        int cc = 0;

        function<void(int)> dfs1 = [&](int node) {
            if (vis[node]) return;
            vis[node] = -1;
            for (auto vec : graph[node])
                dfs1(vec);
            stk.push_back(node);
        };
        function<void(int)> dfs2 = [&](int node) {
            if (vis[node] != -1) return;
            vis[node] = cc;
            for (auto vec : graph[~node])
                dfs2(~vec);
        };
        for (int i = 0; i < n; ++i) dfs1(i), dfs1(~i);
        for (int i = 2 * n - 1; i >= 0; --i) ++cc, dfs2(stk[i]);

        vector<int> ans(n);
        for (int i = 0; i < n; ++i) {
            if (vis[i] == vis[~i]) return {};
            ans[i] = vis[i] > vis[~i];
        }
        return ans;
    }
};
```

SmallDijkstra.h

Description: Dijkstra optimization on graphs with small costs. Edges of graph must have costs in the range $[0..lim]$. Also works for cases where maximum distance is bounded by something small (just set $lim = maxd + 1$). Particularly useful for linear $0 - 1$ BFS.
Time: $\mathcal{O}(V + E + maxd)$.   be7a29, 20 lines

```
template<typename Graph>
vector<int> Dijkstra(Graph& graph, int src, int lim) {
    vector<vector<int>> qs(lim);
    vector<int> dist(graph.size(), -1);
```

```
    dist[src] = 0; qs[0].push_back(src);
    for (int d = 0, maxd = 0; d <= maxd; ++d) {
        for (auto& q = qs[d % lim]; q.size(); ) {
            int node = q.back(); q.pop_back();
            if (dist[node] != d) continue;
            for (auto [vec, cost] : graph[node]) {
                if (dist[vec] != -1 && dist[vec] <= d + cost) continue;
                dist[vec] = d + cost;
                qs[(d + cost) % lim].push_back(vec);
                maxd = max(maxd, d + cost);
            }
        }
    }
    return dist;
}
```

EulerWalk.h

Description: Eulerian undirected/directed path/cycle algorithm. For a directed/undirected graph. For each path/cycle found, calls a callback. You can check cycle by checking path endpoints.
Time: $\mathcal{O}(V + E)$.   a274cc, 43 lines

```
bool DIR = false;

struct EulerWalk {
    int n;
    vector<vector<pair<int, int>>> graph;
    vector<int> cap, walk, buf, deg;

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

    int AddEdge(int a, int b, int c = 1) {
        int ret = cap.size();
        graph[b].emplace_back(a, ret);
        if (!DIR) graph[a].emplace_back(b, ret);
        cap.push_back(c);
        deg[a] += c; deg[b] -= c;
        if (!DIR) deg[a] %= 2, deg[b] %= 2;
        return ret;
    }

    void dfs(int node) {
        while (graph[node].size()) {
            auto [vec, ei] = graph[node].back();
            if (!cap[ei]) graph[node].pop_back();
            else --cap[ei], dfs(vec);
        }
        walk.push_back(node);
    }

    template<typename CB>
    void Go(CB&& cb) {
        for (int i = 0; i <= n; ++i) {
            if (deg[i] < 0) Add(i, n, -deg[i]);
            if (deg[i] > 0) Add(n, i, +deg[i]);
            assert(deg[i] == 0);
        }
        for (int i = n; i >= 0; --i)
            dfs(i), walk.push_back(n);
        for (int i = 0, j = 0; i < (int)walk.size(); i = j + 1) {
            for (j = i; walk[j] < n; ++j);
            if (j - i > 1) cb({walk.begin() + i, walk.begin() + j});
        }
    }
};
```

MaximumClique.h
Description: Quickly finds a maximum clique of a graph (given as symmetric bitset matrix; self-edges not allowed). Can be used to find a maximum independent set by finding a clique of the complement graph.
Time: Runs in about 1s for n=155 and worst case random graphs (p=.90). Runs faster for sparse graphs.
<div align="right">f7c0bc, 49 lines</div>

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

7.2   Flows, Matchings, Matroids

EZFlow.h
Description: A slow, albeit very easy-to-implement flow algorithm. Mutates graph. Prefer this to DinicFlow when constraints allow.
Usage: while (PushFlow(graph, 0, n - 1)) ++ans;
Time: $\mathcal{O}(EF)$ where F is the maximum flow.
<div align="right">51a2d9, 16 lines</div>

```cpp
bool PushFlow(vector<vector<int>>& graph, int s, int t) {
  vector<bool> vis(graph.size(), false);
  function<bool(int)> dfs = [&](int node) {
    if (node == t) return true;
    vis[node] = true;
```

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

DinicFlow.h
Description: Quick flow algorithm. If it TLEs on ugly inputs, adapt it to use capacity scaling.
Time: $\mathcal{O}(V^2 E)$ or $\mathcal{O}(E\sqrt{E})$ on unit graphs.
<div align="right">78938c, 58 lines</div>

```cpp
using T = int;

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

  vector<Edge> es;
  vector<int> graph, at, dist, q;

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

  int AddEdge(int a, int b, T c, bool dir = true) {
    auto add = [&](int a, int b, T c) {
      es.push_back({a, b, graph[a], c, 0});
      graph[a] = es.size() - 1;
    };
    add(a, b, c); add(b, a, dir ? 0 : c);
    return es.size() - 2;
  }
  bool bfs(int src, int dest) {
    dist.assign(graph.size(), -1); q.clear();
    dist[src] = 0; q.push_back(src);
    for (int i = 0; i < (int)q.size(); ++i) {
      int node = q[i];
      for (int ei = graph[node]; ei >= 0; ei = es[ei].nxt) {
        const auto &e = es[ei];
        if (dist[e.to] == -1 && e.cap > e.flow) {
          dist[e.to] = dist[node] + 1;
          q.push_back(e.to);
        }
      }
    }
    return dist[dest] != -1;
  }
  T dfs(int node, int dest, T need) {
    if (!need || node == dest) return need;
    T ret = 0;
    for (int& ei = at[node]; ei != -1; ei = es[ei].nxt) {
      const auto &e = es[ei];
      if (dist[e.to] != dist[node] + 1) continue;
      if (T now = dfs(e.to, dest, min(need, e.cap - e.flow))) {
        es[ ei ].flow += now;
        es[ei^1].flow -= now;
        ret += now; need -= now;
      }
      if (!need) break;
    }
    return ret;
  }
  T Compute(int src, int dest) {
    T ret = 0;
    while (bfs(src, dest)) {
      at = graph;
```

```cpp
      ret += dfs(src, dest, numeric_limits<T>::max());
    }
    return ret;
  }
  bool SideOfCut(int x) { return dist[x] == -1; }
};
```

GlobalMinCut.h
Description: Find a global minimum cut in an undirected graph, as represented by an adjacency matrix.
Time: $\mathcal{O}(V^3)$, optimizable to $\mathcal{O}(VE \log E)$ (with heap)
<div align="right">4a0096, 24 lines</div>

```cpp
pair<int, vector<bool>> GlobalMinCut(vector<vector<int>> W) {
  int n = W.size(), best = 2e9;
  vector<bool> cut(n), best_cut, vis;

  for (int phase = n - 1; phase > 0; phase--) {
    vector<int> deg = W[0];
    int prev, cur = 0;
    vis = cut;
    for (int i = 0; i < phase; ++i) {
      prev = cur; cur = -1;
      for (int j = 1; j < n; ++j)
        if (!vis[j] && (cur == -1 || deg[j] > deg[cur]))
          cur = j;
      vis[cur] = true;
      for (int j = 0; j < n; ++j) if (!vis[j])
        deg[j] += W[cur][j];
    }
    if (deg[cur] < best) best = deg[cur], best_cut = cut;
    for (int j = 0; j < n; ++j)
      W[prev][j] += W[cur][j], W[j][prev] += W[j][cur];
    cut[cur] = true;
  }
  return {best, best_cut};
}
```

GomoryHu.h
Description: Computes Gomory-Hu tree of an undirected graph.
Time: $\mathcal{O}(V)$ calls of flow algorithm
<div align="right">fc73fe, 19 lines</div>

```cpp
tuple<T, int, int> GomoryHu(Dinic& D) {
  int n = D.graph.size();
  auto ret = make_tuple(numeric_limits<T>::max(), -1, -1);
  vector<int> par(n, 0); par[0] = -1;
  vector<vector<T>> ans(n, vector<T>(n, get<0>(ret)));

  for (int i = 1; i < n; ++i) {
    T now = D.Compute(i, par[i]);
    for (int j = i + 1; j < n; ++j)
      if (D.SideOfCut(j) == 0 && par[j] == par[i])
        par[j] = i;
    ans[i][par[i]] = ans[par[i]][i] = now;
    for (int j = 0; j < i; ++j)
      ans[i][j] = ans[j][i] = min(now, ans[par[i]][j]);
    ret = min(ret, make_tuple(now, i, par[i]));
    for (auto& e : D.es) e.flow = 0;
  }
  return ret;
}
```

DFSMatching.h
Description: This is a simple matching algorithm but should be just fine in most cases. Works very fast for all cases, including non-bipartite graphs. For bipartite graphs, you can safely set rem = 1 ar (*). If getting WA on non-bipartite graphs, increase number of iterations (*) or use a proper algorithm.
Time: $\mathcal{O}(EV)$ where E is the number of edges and V is the number of vertices.
<div align="right">299f5a, 34 lines</div>

```
mt19937 rng(time(0));

vector<int> Match(vector<vector<int>>& graph) {
  int n = graph.size(), rem = 1;
  vector<int> vis(n), mate(n, -1), order(n);
  auto mateup = [&](int a, int b) {
    int c = mate[b]; mate[a] = b; mate[b] = a;
    if (c != -1) mate[c] = -1;
    return c;
  };
  function<bool(int)> dfs = [&](int node) {
    if (vis[node]) return false;
    vis[node] = true;
    shuffle(graph[node].begin(), graph[node].end(), rng);
    for (auto vec : graph[node])
      if (mate[vec] == -1)
        return mateup(node, vec), true;
    for (auto vec : graph[node]) {
      int old = mateup(node, vec);
      if (dfs(old)) return true;
      mateup(old, vec);
    }
    return false;
  };
  iota(order.begin(), order.end(), 0);
  while (rem--) {
    shuffle(order.begin(), order.end(), rng);
    vis.assign(n, false);
    for (auto i : order)
      if (mate[i] == -1 && dfs(i))
        rem = 50; // (*)
  }
  return mate;
}
```

HopcroftKarp.h
Description: Hopcroft-Karp bipartite matching algorithm.
Time: $\mathcal{O}\left(E\sqrt{V}\right)$ where E is the number of edges and V is the number of vertices.   973836, 28 lines

```
vector<int> Match(vector<vector<int>>& graph, int n, int m) {
  vector<int> l(n, -1), r(m, n), q, dist;
  function<bool(int)> dfs = [&](int u) {
    if (u == n) return true;
    int d = dist[u]; dist[u] = -1;
    for (auto v : graph[u])
      if (dist[r[v]] == d + 1 && dfs(r[v]))
        return l[u] = v, r[v] = u, true;
    return false;
  };
  while (true) {
    dist.assign(n + 1, -1); q.clear();
    for (int i = 0; i < n; ++i)
      if (l[i] == -1)
        dist[i] = 0, q.push_back(i);
    for (int i = 0; i < (int)q.size(); ++i) {
      int u = q[i]; if (u == n) break;
      for (auto v : graph[u])
        if (dist[r[v]] == -1)
          dist[r[v]] = 1 + dist[q[i]], q.push_back(r[v]);
    }
    if (dist[n] == -1) break;
    for (int i = 0; i < n; ++i)
      if (l[i] == -1)
        dfs(i);
  }
  return l;
}
```

Blossom.h
Description: Edmonds Blossom general matching algorithm.
Time: $\mathcal{O}(EV)$ where E is the number of edges and V is the number of vertices.   1b2a6f, 52 lines

```
vector<int> Blossom(vector<vector<int>>& graph) {
  int n = graph.size(), timer = -1;
  vector<int> mate(n, -1), label(n), parent(n),
              orig(n), aux(n, -1), q;
  auto lca = [&](int x, int y) {
    for (timer++; ; swap(x, y)) {
      if (x == -1) continue;
      if (aux[x] == timer) return x;
      aux[x] = timer;
      x = (mate[x] == -1 ? -1 : orig[parent[mate[x]]]);
    }
  };
  auto blossom = [&](int v, int w, int a) {
    while (orig[v] != a) {
      parent[v] = w; w = mate[v];
      if (label[w] == 1) label[w] = 0, q.push_back(w);
      orig[v] = orig[w] = a; v = parent[w];
    }
  };
  auto augment = [&](int v) {
    while (v != -1) {
      int pv = parent[v], nv = mate[pv];
      mate[v] = pv; mate[pv] = v; v = nv;
    }
  };
  auto bfs = [&](int root) {
    fill(label.begin(), label.end(), -1);
    iota(orig.begin(), orig.end(), 0);
    q.clear();
    label[root] = 0; q.push_back(root);
    for (int i = 0; i < (int)q.size(); ++i) {
      int v = q[i];
      for (auto x : graph[v]) {
        if (label[x] == -1) {
          label[x] = 1; parent[x] = v;
          if (mate[x] == -1)
            return augment(x), 1;
          label[mate[x]] = 0; q.push_back(mate[x]);
        } else if (label[x] == 0 && orig[v] != orig[x]) {
          int a = lca(orig[v], orig[x]);
          blossom(x, v, a); blossom(v, x, a);
        }
      }
    }
    return 0;
  };
  // Time halves if you start with (any) maximal matching.
  for (int i = 0; i < n; i++)
    if (mate[i] == -1)
      bfs(i);
  return mate;
}
```

MinCostMaxFlow.h
Description: Min-cost max-flow with potentials technique. If costs can be negative, call SetPi before Compute, but note that negative cost cycles are not allowed. To obtain the actual flow, look at positive values only.
Time: $\mathcal{O}(FE \log E)$ 244dce, 70 lines

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

struct MFMC {
  struct Edge { int to, rev; T cap, cost, flow; };
  int n;
  vector<vector<Edge>> graph;
  vector<int> par;
  vector<T> dist, pi;
  vector<Edge> es;
  priority_queue<pair<T, int>> pq;

  MFMC(int n) : n(n), graph(n), par(n), dist(n), pi(n) {}

  void AddEdge(int a, int b, T cap, T cost, T flow = 0) {
    graph[a].push_back({b, (int)graph[b].size(), cap, cost,
        flow});
    graph[b].push_back({a, (int)graph[a].size() - 1, 0, -cost,
        -flow});
  }
  bool relax(int from, const Edge& e) {
    if (dist[from] == INF) return false;
    T now = dist[from] + pi[from] - pi[e.to] + e.cost;
    if (e.flow < e.cap && now < dist[e.to])
      return dist[e.to] = now, par[e.to] = e.rev, true;
    return false;
  }
  bool dijkstra(int s, int t) {
    dist.assign(n, INF); par.assign(n, -1);
    dist[s] = 0; pq.emplace(0, s);
    while (!pq.empty()) {
      auto [d, node] = pq.top(); pq.pop();
      if (dist[node] != -d) continue;
      for (auto& e : graph[node])
        if (relax(node, e))
          pq.emplace(-dist[e.to], e.to);
    }
    for (int i = 0; i < n; ++i)
      pi[i] = min(pi[i] + dist[i], INF);
    return par[t] != -1;
  }
  pair<T, T> Compute(int s, int t) {
    T flow = 0, cost = 0;
    while (dijkstra(s, t)) {
      T now = INF;
      for (int phase : {0, 1}) {
        for (int node = t; node != s; ) {
          auto& e1 = graph[node][par[node]];
          auto& e2 = graph[e1.to][e1.rev];
          if (!phase) now = min(now, e2.cap - e2.flow);
          else e2.flow += now, e1.flow -= now;
          node = e1.to;
        }
      }
      flow += now;
      cost += pi[t];
    }
    return {flow, cost};
  }
  // If some costs can be negative, call this before maxflow:
  void SetPi(int s) { // (otherwise, leave this out)
    dist.assign(n, INF); dist[s] = 0;
    int it = n, ch = 1;
    while (ch-- && it--)
      for (int i = 0; i < n; ++i)
        for (auto& e : graph[i])
          ch |= relax(i, e);
    assert(it >= 0); // negative cost cycle
    pi = dist;
  }
};
```

NetworkSimplex.h

Description: Compute instances of minimum cost circulations very fast (faster than SSP). Handles negative cycles and node demands. For edge lower bounds, adjust supplies accordingly. To handle supplies, modify code at (*) to edges with cost $\infty = 1 + \sum_i c_i$ and capacity sup_i.
Time: Expect $\mathcal{O}\left(\text{poly}(E) \cdot V\right)$ (very fast in practice) 385766, 86 lines

```cpp
const int INF = 1e9; // greater than sum(e.k), INF * sum(sup)
        should fit
using ll = int;

struct NetworkSimplex {
  struct Edge { int a, b, c, k, f = 0; };

  int n;
  vector<int> pei, nxt;
  vector<ll> dual;
  vector<Edge> E;
  vector<set<int>> tree;
  vector<int> stk;

  NetworkSimplex(int n) :
    n(n), pei(n + 1, -1), nxt(n + 1, -1),
    dual(n + 1, 0), tree(n + 1) {}

  int AddEdge(int a, int b, int c, int k) {
    E.push_back({a, b, c, k});
    E.push_back({b, a, 0, -k});
    return E.size() - 2;
  }
  void build(int ei = -1) {
    stk.push_back(ei);
    while (stk.size()) {
      int ei = stk.back(), v = n; stk.pop_back();
      if (ei != -1) {
        dual[E[ei].b] = dual[E[ei].a] + E[ei].k;
        pei[E[ei].b] = (ei ^ 1);
        v = E[ei].b;
      }
      for (auto nei : tree[v])
        if (nei != pei[v])
          stk.push_back(nei);
    }
  }
  long long Compute() {
    for (int i = 0; i < n; ++i) {
      int ei = AddEdge(n, i, 0, 0);
      tree[n].insert(ei);
      tree[i].insert(ei^1);
    }
    build();

    long long answer = 0;
    ll flow, cost; int ein, eout, ptr = 0;
    const int B = n / 2 + 1;
    for (int it = 0; it < E.size() / B + 1; ++it) {
      // Find negative cycle (round–robin).
      cost = 0; ein = -1;
      for (int t = 0; t < B; ++t, (++ptr) %= E.size()) {
        auto& e = E[ptr];
        ll now = dual[e.a] + e.k - dual[e.b];
        if (e.f < e.c && now < cost)
          cost = now, ein = ptr;
      }
      if (ein == -1) continue;

      // Pivot around ein.
      for (int v = E[ein].b; v < n; v = E[pei[v]].b)
        nxt[v] = pei[v];
```

```cpp
      for (int v = E[ein].a; v < n; v = E[pei[v]].b)
        nxt[E[pei[v]].b] = (pei[v]^1);
      nxt[E[ein].a] = -1;

      int flow = E[ein].c - E[ein].f; eout = ein;
      for (int ei = ein; ei != -1; ei = nxt[E[ei].b]) {
        int res = E[ei].c - E[ei].f;
        if (res < flow) flow = res, eout = ei;
      }
      for (int ei = ein; ei != -1; ei = nxt[E[ei].b])
        E[ei].f += flow, E[ei^1].f -= flow;

      if (ein != eout) {
        tree[E[ein].a].insert(ein);
        tree[E[ein].b].insert(ein^1);
        tree[E[eout].a].erase(eout);
        tree[E[eout].b].erase(eout^1);
        build(pei[E[eout].a] == eout ? ein : ein^1);
      }
      answer += 1LL * flow * cost;
      it = -1;
    }
    return answer;
  }
};
```

WeightedMatching.h

Description: Simplified Jonker-Volgenant algorithm for assignment problem. Negate costs for max cost. Incremental from 0 to $n - 1$.
Time: $\mathcal{O}\left(N^3\right)$, fast in practive 601155, 49 lines

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

  ll w, h; int j, l, s, t;
  auto reduce = [&]() {
    if (s == t) {
      l = s; w = dist[idx[t++]];
      for (int k = t; k < m; ++k) {
        j = idx[k]; h = dist[j];
        if (h > w) continue;
        if (h < w) t = s, w = h;
        idx[k] = idx[t]; idx[t++] = j;
      }
      for (int k = s; k < t; ++k) {
        j = idx[k];
        if (R[j] < 0) return 1;
      }
    }
    int q = idx[s++], p = R[q];
    for (int k = t; k < m; ++k) {
      j = idx[k]; h = W[p][j] - W[p][q] + v[q] - v[j] + w;
      if (h < dist[j]) {
        dist[j] = h; prev[j] = p;
        if (h == w) {
          if (R[j] < 0) return 1;
          idx[k] = idx[t]; idx[t++] = j;
        }
      }
    }
    return 0;
  };
  for (int i = 0; i < n; ++i) {
    for (int k = 0; k < m; ++k)
      dist[k] = W[i][k] - v[k], prev[k] = i;
```

```cpp
    s = t = 0;
    while (!reduce());
    for (int k = 0; k < l; ++k) v[idx[k]] += dist[idx[k]] - w;
    for (int k = -1; k != i;)
      R[j] = k = prev[j], swap(j, L[k]);
  }
  ll ret = 0;
  for (int i = 0; i < n; ++i)
    ret += W[i][L[i]]; // (i, L[i]) is a solution
  return ret;
}
```

MatroidIntersection.h

Description: Weighted Matroid intersection algorithm. Given two matroids $M1$ and $M2$ on the same set $[0..n)$, computes the maximal independent set on both matroids. Matroids are functions which should return vector of elements that can be added to an existing solution. For unweighted version, set $w[i] = 0$.
Usage: Color C(colors); Forest F(n, edges);
auto sol = MatroidIntersection(C, F, w);
Time: Generally $\mathcal{O}\left(M^2N\right)$, where M is the solution size. 0345ef, 42 lines

```cpp
template <class M1, class M2>
vector<bool> MatrInter(M1 m1, M2 m2, vector<int> w) {
  int n = w.size();
  vector<bool> sol(n, false);
  while (true) {
    // Build graph.
    vector<vector<int>> graph(n);
    for (int i = 0; i < n; ++i) {
      if (!sol[i]) continue;
      sol[i] = 0;
      for (auto j : m1(sol)) graph[i].push_back(j);
      for (auto j : m2(sol)) graph[j].push_back(i);
      sol[i] = 1;
    }
    // Find augmenting path (Bellman–Ford).
    vector<int> inq(n, 0), parent(n, -2), q;
    vector<long long> dist(n, 1LL * M * M);
    auto push = [&](int v, int p, long long d) {
      if (dist[v] <= d) return;
      dist[v] = d; parent[v] = p;
      if (!inq[v]) inq[v] = 1, q.push_back(v);
    };
    for (auto node : m1(sol))
      push(node, -1, 1LL * w[node] * n);
    for (int i = 0; i < (int)q.size(); ++i) {
      int node = q[i]; inq[node] = 0;
      for (auto vec : graph[node])
        if (vec != node)
          push(vec, node, dist[node] +
              (sol[vec] ? -1LL : 1LL) * w[vec] * n + 1);
    }
    int choose = -1; long long best = 4e18;
    for (auto node : m2(sol))
      if (dist[node] < best)
        best = dist[node], choose = node;
    if (choose == -1) break;
    // Augment.
    for (int node = choose; node != -1; node = parent[node])
      sol[node] = !sol[node];
  }
  return sol;
}
```

7.2.1 Flows with demands

To solve flow with demands where $l(u,v) \le f(u,v) \le h(u,v)$, first find feasible flow by solving max S'-T' flow on network:

- $c'(S', v) = \sum_{u \in V} l(u, v)$ for all $v \in V$
- $c'(v, T') = \sum_{w \in V} l(v, w)$ for all $v \in V$
- $c'(u, v) = h(u, v) - l(u, v)$ for all $(u, v) \in E$
- $c'(T, S) = \infty$

Assert that flow equals $\sum l(u, v)$. Next, solve max flow on "residual" graph given flow function (take each edge $(u, v) \in E$ and set $c(u, v) = h(u, v) - l(u, v), f(u, v) = f'(u, v)$). For minimum flow, find max $T - S$ flow on the same network.

7.3 Trees

BinaryLifting.h
Description: Calculate skew-binary links.
Time: construction $\mathcal{O}(N)$, queries $\mathcal{O}(\log N)$

<div align="right">59f2fb, 39 lines</div>

```cpp
struct Lift {
  struct Data { int par, link, dep; };
  vector<Data> T;

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

  void Add(int node, int par) {
    if (par == -1) T[node] = Data{-1, node, 0};
    else {
      int link = par, a1 = T[par].link, a2 = T[a1].link;
      if (2 * T[a1].dep == T[a2].dep + T[par].dep)
        link = a2;
      T[node] = Data{par, link, T[par].dep + 1};
    }
  }

  int Kth(int node, int k) {
    int seek = T[node].dep - k;
    if (seek < 0) return -1;
    while (T[node].dep > seek)
      node = (T[T[node].link].dep >= seek)
        ? T[node].link : T[node].par;
    return node;
  }

  int LCA(int a, int b) {
    if (T[a].dep < T[b].dep) swap(a, b);
    while (T[a].dep > T[b].dep)
      a = (T[T[a].link].dep >= T[b].dep)
        ? T[a].link : T[a].par;
    while (a != b) {
      if (T[a].dep == 0) return -1;
      if (T[a].link != T[b].link)
        a = T[a].link, b = T[b].link;
      else a = T[a].par, b = T[b].par;
    }
    return a;
  }
};
```

LCA.h
Description: Lowest common ancestor. Finds the lowest common ancestor in a rooted tree.
Usage: LCA lca(graph);
lc = lca.Query(u, v);
Time: $\mathcal{O}(N \log N + Q)$

"../data-structures/RMQ.h" <div align="right">ad70bf, 24 lines</div>

```cpp
struct LCA {
  int n, timer = 0;
  vector<int> enter, pv, pt;
  RMQ rmq;
```

```cpp
  LCA(vector<vector<int>>& graph, int root = 0) :
    n(graph.size()), enter(n, -1),
    rmq((dfs(graph, root), pt)) {}

  void dfs(auto& graph, int node) {
    enter[node] = timer++;
    for (auto vec : graph[node]) {
      if (enter[vec] != -1) continue;
      pv.push_back(node), pt.push_back(enter[node]);
      dfs(graph, vec);
    }
  }

  int Query(int a, int b) {
    if (a == b) return a;
    a = enter[a], b = enter[b];
    return pv[rmq.Query(min(a, b), max(a, b))];
  }
  // Distance is depth[a] + depth[b] - 2 depth[Query(a, b)]
};
```

CompressTree.h
Description: Given a rooted tree/forest and a subset v of nodes, compute the minimal subtree that contains all the nodes by adding all (at most $|v|-1$) pairwise LCA's and compressing edges. Mutates v to contain nodes of the reduced tree, while at the same time populating a *link* array that stores the new parents. The root(s) point to -1.
Time: $\mathcal{O}(|v| * (\log |v| + LCA\_Q))$

"LCA.h" <div align="right">e154f2, 12 lines</div>

```cpp
void CompressTree(LCA& L, vector<int>& v, vector<int>& link) {
  for (int it = 0; it < 2; ++it) {
    for (int i = (int)v.size() - 1; i; --i) {
      link[v[i]] = L.Query(v[i - 1], v[i]);
      if (link[v[i]] != -1) v.push_back(link[v[i]]);
    }
    sort(v.begin(), v.end(), [&](int a, int b) {
      return L.enter[a] < L.enter[b];
    });
    v.erase(unique(v.begin(), v.end()), v.end());
  }
}
```

Centroid.h
Description: Computes centroid labels and the centroid tree for a given tree. Centroid labels have the property that for any two vertices u, v with $clab[u] = clab[v]$ we are guaranteed that there is a vertex w on the path from u to v that has a strictly higher label. In order to simulate tree D&C, copy the dfs function from CentrTree and adapt it accordingly. To optimize, convert inline functions to global ones.
Time: construction $\mathcal{O}(N)$, D&C $\mathcal{O}(N \log N)$

<div align="right">d88af1, 33 lines</div>

```cpp
vector<int> CentrLabel(vector<vector<int>>& graph) {
  vector<int> clab(graph.size(), -1);
  function<int(int)> dfs = [&](int node) {
    int one = 0, two = 0;
    int& l = clab[node]; l = 0;
    for (auto vec : graph[node]) {
      if (clab[vec] != -1) continue;
      int son = dfs(vec);
      two |= (one & son); one |= son;
    }
    while (two) ++l, one /= 2, two /= 2;
    while (one % 2) ++l, one /= 2;
    return (one | 1) << l;
  };
  dfs(0);
  return clab;
}
```

```cpp
vector<int> CentrTree(vector<vector<int>>& graph) {
  int n = graph.size(), root;
  vector<int> clab = CentrLabel(graph), cpar(n, -1);
  function<void(int, int)> dfs = [&](int node, int par) {
    for (auto vec : graph[node]) {
      if (vec == par || clab[vec] >= clab[root]) continue;
      if (cpar[vec] == -1 || clab[cpar[vec]] > clab[root])
        cpar[vec] = root;
      dfs(vec, node);
    }
  };
  for (root = 0; root < n; ++root)
    dfs(root, -1);
  return cpar;
}
```

HLD.h
Description: Decomposes a tree into vertex disjoint heavy paths and light edges such that the path from any leaf to the root contains at most $log(n)$ light edges. If values are on edges, modify (*) to exclude the highest vertex.
WARNING: Reorders edges of graph. Ranges are inclusive.

<div align="right">b03d6e, 46 lines</div>

```cpp
struct HeavyLight {
  int n, timer;
  vector<int> jump, sub, depth, enter, parent;

  HeavyLight(vector<vector<int>>& graph, int root = 0) :
    n(graph.size()), jump(n), sub(n),
    depth(n), enter(n), parent(n) {
    for (auto _ : {0, 1})
      timer = 0, dfs(graph, root, -1, 0, -1);
  }

  int dfs(auto& graph, int node, int par, int dep, int jmp) {
    if (jmp == -1) jmp = node;
    parent[node] = par; depth[node] = dep; jump[node] = jmp;
    enter[node] = timer++;
    int heavy = 0, ret = 1;
    for (auto& vec : graph[node]) {
      if (vec == par) continue;
      int now = dfs(graph, vec, node, dep + 1, jmp);
      if (heavy < now) heavy = now, swap(vec, graph[node][0]);
      ret += now; jmp = -1;
    }
    return sub[node] = ret;
  }
  // Returns the label in the HL linearization
  int Get(int node) { return enter[node]; }
  // Runs a callback for all ranges [l, r] in the path
  // a -> b. Some ranges might have l > r; if combining
  // function is commutative just swap them in callback.
  template<typename Callback>
  void QueryPath(int a, int b, Callback&& cb) {
    if (jump[a] == jump[b]) {
      cb(enter[a], enter[b]); // (*)
    } else if (depth[jump[a]] > depth[jump[b]]) {
      cb(enter[a], enter[jump[a]]);
      QueryPath(parent[jump[a]], b, cb);
    } else {
      QueryPath(a, parent[jump[b]], cb);
      cb(enter[jump[b]], enter[b]);
    }
  }
  // Range [l, r] corresponding to nodes in the subtree.
  pair<int, int> QuerySubtree(int node) {
    return {enter[node], enter[node] + sub[node] - 1};
  }
};
```

LinkCut.h
Description: Mother of all tree data structures. Keeps an dynamic unordered forest of trees. Can do all sorts of aggregates (path/subtree aggregates) as well as vertex updates. For edge weights, create extra nodes for each edge and Link(a, e), Link(b, e).
Time: $\mathcal{O}(\log N)$ per operation
`"SplayTree.h"`   d848b4, 45 lines

```cpp
struct LinkCut : SplayTree {
  LinkCut(int n) : SplayTree(n) {}

  int access(int x) {
    int u = x, v = 0;
    for (; u; v = u, u = T[u].p) {
      splay(u);
      int& ov = T[u].ch[1];
      T[u].vir += T[ov].sub;
      T[u].vir -= T[v].sub;
      ov = v; pull(u);
    }
    return splay(x), v;
  }
  void reroot(int x) {
    access(x); T[x].flip ^= 1; push(x);
  }
  void Link(int u, int v) {
    reroot(u); access(v);
    T[v].vir += T[u].sub;
    T[u].p = v; pull(v);
  }
  void Cut(int u, int v) {
    reroot(u); access(v);
    T[v].ch[0] = T[u].p = 0; pull(v);
  }
  // Rooted tree LCA. Returns 0 if u and v arent connected.
  int LCA(int u, int v) {
    if (u == v) return u;
    access(u); int ret = access(v);
    return T[u].p ? ret : 0;
  }
  // Query subtree of u where v is outside the subtree.
  long long Subtree(int u, int v) {
    reroot(v); access(u); return T[u].vir + T[u].self;
  }
  // Query path [u..v].
  long long Path(int u, int v) {
    reroot(u); access(v); return T[v].path;
  }
  // Update vertex with value v.
  void Update(int u, long long v) {
    access(u); T[u].self = v; pull(u);
  }
};
```

7.4 Misc

DominatorTree.h
Description: Computes the dominator tree *dom* of a given directed graph *G*. A node *v* dominates *u* if all paths from *src* to *v* go through *u*. All nodes that aren't reachable from *src* will have $dom(u) = -1$. *src* will also have $dom(src) = -1$.
Time: $\mathcal{O}(M \log N)$ (often faster)
4230f7, 45 lines

```cpp
vector<int> DomTree(vector<vector<int>>& graph, int src) {
  int n = graph.size();
  vector<vector<int>> tree(n), trans(n), buck(n);
  vector<int> semi(n), par(n), dom(n), label(n),
    atob(n, -1), btoa(n, -1), link(n, -1);

  function<int(int, int)> find = [&](int u, int d) {
    if (link[u] == -1)
```

```cpp
      return d ? -1 : u;
    int v = find(link[u], d + 1);
    if (v < 0) return u;
    if (semi[label[link[u]]] < semi[label[u]])
      label[u] = label[link[u]];
    link[u] = v;
    return d ? v : label[u];
  };
  int t = 0;
  function<void(int)> dfs = [&](int u) {
    atob[u] = t; btoa[t] = u;
    label[t] = semi[t] = t; t++;
    for (auto v : graph[u]) {
      if (atob[v] == -1)
        dfs(v), par[atob[v]] = atob[u];
      trans[atob[v]].push_back(atob[u]);
    }
  };
  dfs(src);
  for (int u = t - 1; u >= 0; --u) {
    for (auto v : trans[u])
      semi[u] = min(semi[u], semi[find(v, 0)]);
    if (u) buck[semi[u]].push_back(u);
    for (auto w : buck[u]) {
      int v = find(w, 0);
      dom[w] = semi[v] == semi[w] ? semi[w] : v;
    }
    if (u) link[u] = par[u];
  }
  vector<int> ret(n, -1);
  for (int u = 1; u < t; ++u) {
    if (dom[u] != semi[u])
      dom[u] = dom[dom[u]];
    ret[btoa[u]] = btoa[dom[u]];
  }
  return ret;
}
```

DirectedMST.h
Description: Finds a minimum spanning tree/arborescence of a directed graph, given a root node.
where $i > 0$.
Time: $\mathcal{O}(M \log N)$
WARNING: Input graph must be strongly connected and have no loops; if this is not the case for the problem, consider adding "fake" edges $(i, 0, \infty)$ **and** $(0, i, \infty)$.
`"../data-structures/DSU.h"`, `"../data-structures/SkewHeap.h"`   af69b8, 28 lines

```cpp
struct Edge { int a, b; ll c; };

pair<ll, vector<int>> DMST(int n, int src, vector<Edge> es) {
  // Compress graph - O(M logN)
  SkewHeap H; DSU D(2 * n); int x = 0;
  vector<int> par(2 * n, -1), ins(par), vis(par);
  for (auto e : es) ins[e.b] = H.Merge(ins[e.b], H.New(e.c));
  auto go = [&](int x) { return D.Find(es[ins[x]].a); };
  for (int i = n; ins[x] != -1; ++i) {
    for (; vis[x] == -1; x = go(x)) vis[x] = 0;
    for (; x != i; x = go(x)) {
      int rem = ins[x]; ll w = H.Get(rem); H.Pop(rem);
      H.Add(rem, -w); ins[i] = H.Merge(ins[i], rem);
      par[x] = i; D.link[x] = i;
    }
    for (; ins[x] != -1 && go(x) == x; H.Pop(ins[x]));
  }
  // Expand graph - O(N)
  ll cost = 0; vector<int> ans;
  for (int i = src; i != -1; i = par[i]) vis[i] = 1;
  for (int i = x; i >= 0; --i) {
    if (vis[i]) continue;
```

```cpp
    cost += es[ins[i]].c; ans.push_back(ins[i]);
    for (int j = es[ins[i]].b; j != -1 && !vis[j]; j = par[j])
      vis[j] = 1;
  }
  return {cost, ans};
}
```

BottleneckTree.h
Description: Data structure for dynamic online incremental MST. AddEdge returns the cost of the exchanged edge (maximum on path).
Time: $\mathcal{O}(???)$ (probably $\mathcal{O}(\log n)$ amortized per added edge). 6db648, 29 lines

```cpp
struct BottleneckTree {
  vector<int> link;
  vector<ll> cost;

  BottleneckTree(int n) : link(n, 0), cost(n, INF) {
    link[0] = -1; cost[0]++;
  }
  ll AddEdge(int x, int y, ll c) {
    compress(x); reroot(x); compress(y); reroot(y);
    assert(link[x] == y);
    if (cost[x] > c) swap(cost[x], c);
    return c;
  }
  void compress(int x) {
    if (link[x] == -1) return;
    compress(link[x]);
    reroot(link[x], cost[x]);
    if (cost[link[x]] <= cost[x])
      link[x] = link[link[x]];
  }
  void reroot(int x, ll c = INF + 1) {
    int p = link[x];
    while (p != -1 && cost[p] <= c) {
      swap(cost[p], cost[x]);
      int g = link[p]; link[p] = x; p = g;
    }
    link[x] = p;
  }
};
```

EdgeColoring.h
Description: Given a simple, undirected graph with max degree D, computes a $(D + 1)$-coloring of the edges such that no neighboring edges share a color. (D-coloring is NP-hard, but can be done for bipartite graphs by repeated matchings of max-degree nodes.)
Time: $\mathcal{O}(NM)$
e210e2, 31 lines

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

```
      free[right] = e;
    }
    adj[u][d] = fan[i];
    adj[fan[i]][d] = u;
    for (int y : {fan[0], u, end})
      for (int& z = free[y] = 0; adj[y][z] != -1; z++);
  }
  rep(i,0,sz(eds))
    for (tie(u, v) = eds[i]; adj[u][ret[i]] != v;) ++ret[i];
  return ret;
}
```

7.5 Math

7.5.1 Number of Spanning Trees

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

7.5.2 Erdős–Gallai theorem

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

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

Geometry (8)

Point.h
Description: Point declaration, and basic operations.   f894c8, 48 lines

```
using Point = complex<double>;

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

int sgn(double d) {
  return (d > EPS) - (d < -EPS);
}
double dot(Point a, Point b) {
  return a.real() * b.real() + a.imag() * b.imag();
}
double cross(Point a, Point b) {
  return a.real() * b.imag() - a.imag() * b.real();
}
double det(Point a, Point b, Point c) {
  return cross(b - a, c - a);
}
double dist(Point a, Point b) {
  return abs(b - a);
}
Point perp(Point a) { // +90deg
  return {-a.imag(), a.real()};
}
Point rotate_ccw(Point a, double theta) {
  return a * polar(1.0, theta);
}
int half(Point p) { return p < 0; }

namespace std {
```

```
  bool operator<(Point a, Point b) {
    return make_pair(a.real(), a.imag()) <
      make_pair(b.real(), b.imag());
  }
}
// abs() is norm (length) of vector
// norm() is square of abs()
// arg() is angle of vector
// det() is twice the signed area of the triangle abc
// and is > 0 iff c is to the left as viewed from a towards b.
// polar(r, theta) gets a vector from abs() and arg()

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

Line.h
Description: Line operations.
WARNING: Line equation is kept as $ax + by = c$ **(unlike the usual** $ax + by + c = 0$**)**   70c6db, 24 lines

```
using T = int;
using T2 = long long;
using T4 = __int128_t;
const T2 INF = 4e18;

struct Line { T a, b; T2 c; };

bool half(Line m) { return m.a < 0 || m.a == 0 && m.b < 0; };
void normalize(Line& m) {
  T2 g = __gcd(__(T2)__gcd(abs(m.a), abs(m.b)), abs(m.c));
  if (half(m)) g *= -1;
  m.a /= g, m.b /= g, m.c /= g;
}
// Sorts halfplanes in clockwise order.
// To sort lines, normalize first (gcd logic not needed).
bool operator<(Line m, Line n) {
  return make_pair(half(m), (T2)m.b * n.a) <
        make_pair(half(n), (T2)m.a * n.b);
}
Line LineFromPoints(T x1, T y1, T x2, T y2) {
  T a = y1 - y2, b = x2 - x1;
  T2 c = (T2)a * x1 + (T2)b * y1;
  return {a, b, c}; // halfplane points to the left of vec.
}
```

8.1 Geometric primitives

LineInter.h
Description:

Returns the intersection between non-parallel lines. Does
products up to $O(X^3)$. Both vector variant and line variant are present.
"Point.h", "Line.h"   86b737, 12 lines

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

tuple<T4, T4, T2> LineIntersection(Line m, Line n) {
  T2 d = (T2)m.a * n.b - (T2)m.b * n.a; // assert(d);
  T4 x = (T4)m.c * n.b - (T4)m.b * n.c;
  T4 y = (T4)m.a * n.c - (T4)m.c * n.a;
```

```
  return {x, y, d}; // (x/d, y/d) is intersection.
}
```

SegInter.h
Description:
If a unique intersection point between the line segments going
from a to b and from c to d 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. Does
products up to $O(X^3)$.
"Point.h", "OnSegment.h"   204caa, 13 lines

```
vector<Point> SegInter(Point a, Point b, Point c, Point d) {
  auto oa = det(c, d, a), ob = det(c, d, b),
       oc = det(a, b, c), od = det(a, 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<Point> 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()};
}
```

LineDist.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. Does products up to $O(X^3)$.

WARNING: If $a = b$ **nan is returned, although don't rely on that.**
"Point.h"   32cf2c, 7 lines

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

SegDist.h
Description:
Returns the shortest distance between point p and the line
segment from point s to e.
Usage: Point a{0, 0}, b{2, 2}, p{1, 1};
bool onSegment = SegmentDistance(p, a, b) < EPS;
"Point.h"   e35a3d, 12 lines

```
double SegmentDistance(Point p, Point a, Point b) {
  if (sgn(abs(a - b))) return abs(p - a);
  double d = norm(b - a), t = clamp(dot(p - a, b - a), 0., d);
  return abs((p - a) * d - (b - a) * t) / d;
}
// Projects point p on segment [a, b]
Point ProjectPointOnSegment(Point p, Point a, Point b) {
  double d = norm(b - a);
  if (sgn(d) == 0) return a;
  double r = dot(p - a, b - a) / d;
  return (r < 0) ? a : (r > 1) ? b : (a + (b - a) * r);
}
```

OnSeg.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"   88cc3d, 3 lines
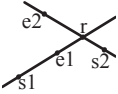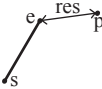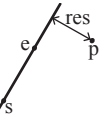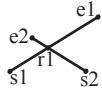
```cpp
bool OnSegment(P s, P e, P p) {
  return sgn(det(p, s, e)) == 0 && sgn(dot(s - p, e - p)) <= 0;
}
```

LinTrans.h
Description:
Apply the affine transformation (translation, rotation and scaling) which maps $(p, q) \rightarrow (fp, fq)$ to point r.

"Point.h"   1d3397, 3 lines

```cpp
Point LinTrans(Point p, Point q, Point fp, Point fq, Point r) {
    return fp + (r - p) * (fq - fp) / (q - p);
}
```

8.2   Polygons

PolyCenter.h
Description: Computes the center of mass of a polygon.

"Point.h"   e2a1f0, 10 lines

```cpp
Point PolygonCenter(vector<Point> P) {
  int n = P.size();
  Point O = 0.; ld A = 0.;
  for (int i = 0, j = n - 1; i < n; j = i++) {
    O += (P[i] + P[j]) * cross(P[j], P[i]);
    A += cross(P[j], P[i]);
  }
  O = O / 3. / A;
  return O;
}
```

InsidePoly.h
Description: Returns true if p lies within the polygon described by the points between iterators begin and end. Returns 0 if on polygon, 1 if inside polygon and -1 if outside. Answer is calculated by counting the number of intersections between the polygon and a line going from p to infinity in the positive x-direction. The algorithm uses products in intermediate steps so watch out for overflow. If points within epsilon from an edge should be considered as on the edge replace the line "if (onSegment..." with the comment bellow it (this will cause overflow for int and long long).
Time: $\mathcal{O}(n)$

"Point.h", "OnSegment.h"   151734, 10 lines

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

ConvexHull.h
Description:
Returns the convex hull of a set of points. Sorts hull in counter-clockwise order. Points on the edge of the hull between two other points are considered part of the hull. To change that, change the sign under (*) to make it non-strict.
Time: $\mathcal{O}(N \log N)$

"Point.h"   c32055, 19 lines

```cpp
vector<Point> HullHalf(vector<Point>& P, int z) {
  Poly ret; // z = +1 if lower, −1 if upper −^
  for (auto p : P) {
    while ((int)ret.size() >= 2 &&          // (*)
        z * sgn(det(ret.end()[-2], ret.end()[-1], p)) < 0)
      ret.pop_back();
    ret.push_back(p);
```

```cpp
  }
  return ret;
}

vector<Point> Hull(vector<Point> P) {
  sort(P.begin(), P.end());
  P.erase(unique(P.begin(), P.end()), P.end());
  if (P.size() <= 1) return P;
  auto l = HullHalf(P, +1), u = HullHalf(P, -1);
  l.insert(l.end(), u.rbegin() + 1, u.rend() - 1);
  return l;
}
```

Minkowski.h
Description: Computes the set $P + Q = \{p + q \mid p \in P, q \in Q\}$, where P, Q are convex sets (described by polygons).
Time: $\mathcal{O}(N + M)$

"Point.h"   e92994, 15 lines

```cpp
vector<Point> MinkowskiSum(vector<Point> P, vector<Point> Q) {
  int n = P.size(), m = Q.size();
  vector<Point> R = {P[0] + Q[0]};
  for (int i = 1, j = 1; i < n || j < m; ) {
    if (i < n && (j == m ||
        cross(P[i] - P[i - 1], Q[j] - Q[j - 1]) > 0)) {
      R.push_back(R.back() + P[i] - P[i - 1]);
      ++i;
    } else {
      R.push_back(R.back() + Q[j] - Q[j - 1]);
      ++j;
    }
  }
  return R;
}
```

ExpandPoly.h
Description: Expands the edges of a polygon with some delta d. To shrink, call with $d < 0$.
Time: $\mathcal{O}(N)$
WARNING: Resulting polygon might self-intersect for big values of d.

"Point.h"   3a9bf0, 15 lines

```cpp
vector<Point> ExpandPoly(vector<Point> P, double d) {
  int n = P.size();
  vector<Point> ret(n);
  for (int i = 0; i < n; ++i) {
    Point prv = P[i == 0 ? n - 1 : i - 1],
          nxt = P[i == n - 1 ? 0 : i + 1],
          cur = P[i];
    Point d1 = cur - prv, d2 = nxt - cur;
    d1 = perp(d1) / abs(d1), d2 = perp(d2) / abs(d2);
    ret[i] = LineIntersection(
        prv - d1 * d, cur - d1 * d,
        cur - d2 * d, nxt - d2 * d);
  }
  return ret;
}
```
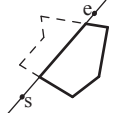
PolyCut.h
Description:
Returns a vector with the vertices of a polygon with everything to the left of the line going from s to e cut away.
Usage: vector<Point> p = ...;
p = PolygonCut(p, Point(0, 0), Point(1, 0));
WARNING: Result might contain degenerations when applied to a concave polygon.

"Point.h", "LineIntersection.h"   001a0d, 10 lines

```cpp
vector<Point> PolygonCut(vector<Point>& P, Point s, Point e) {
```

```cpp
  int n = P.size(); vector<Point> res;
  for (int i = 0, j = n - 1; i < n; j = i++) {
    int z1 = sgn(det(s, e, P[i])), z2 = sgn(det(s, e, P[j]));
    if (z1 * z2 == -1)
      res.push_back(LineIntersection(s, e, P[i], P[j]));
    if (z1 <= 0) res.push_back(P[i]);
  }
  return res;
}
```

Voronoi.h
Description: Determines the voronoi cell of a point with a list of other points. to see if the cell is unbounded, check for points with very high coordinates.
Time: $\mathcal{O}(N^2)$ technically, but expect $\mathcal{O}(NM)$, where M is size of output.

"Point.h", "PolygonCut.h"   e0536d, 17 lines

```cpp
const double INF = 1e9;

// To the right of mediator is region closer to b
pair<Point, Point> Mediator(Point a, Point b) {
  Point m = (a + b) * .5;
  return make_pair(m, m + perp(b - a));
}
vector<Point> VoronoiCell(Point p, vector<Point> P) {
  vector<Point> ret = {{-INF, -INF}, {+INF, -INF},
                       {+INF, +INF}, {-INF, +INF}};
  for (auto q : P) {
    if (p == q) continue;
    auto [a, b] = Mediator(p, q);
    ret = PolygonCut(ret, b, a);
  }
  return ret;
}
```

8.3 Circles

Circle.h
Description: Circle
"Point.h"   1ee1ed, 1 lines

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

LineCircInter.h
Description: Computes the intersection(s) between a line pq and a circle. Can be 0(non-intersecting), 1(tangent), or 2 points
"Circle.h", "LineDistance.h"   93c2d9, 10 lines

```
void LineCircleIntersect(Circle c, Point p, Point q,
                         vector<Point>& inter) {
  Point mid = ProjectPointOnLine(c.c, p, q);
  double d2 = norm(mid - c.c), dist = c.r * c.r - d2;
  if (sgn(dist) < 0) return;
  Point dir = (q - p) * sqrt(dist) / abs(q - p);
  inter.push_back(mid - dir);
  if (sgn(dist) != 0)
    inter.push_back(mid + dir);
}
```

CircCircInter.h
Description: Computes the intersection(s) between two circles. Can be 0(non-intersecting), 1(tangent), or 2 points.
"Circle.h"   fd1b8e, 15 lines

```
void CircleCircleIntersect(Circle c, Circle d,
                           vector<Point>& inter) {
  Point a = c.c, b = d.c, delta = b - a;
  double r1 = c.r, r2 = d.r;
  if (sgn(abs(delta)) == 0) return;
  double r = r1 + r2, d2 = norm(delta);
  double p = (d2 + r1 * r1 - r2 * r2) / (2.0 * d2);
  double h2 = r1 * r1 - p * p * d2;
  if (sgn(d2 - r * r) > 0 || sgn(h2) < 0) return;
  Point mid = a + delta * p,
        per = perp(delta) * sqrt(abs(h2) / d2);
  inter.push_back(mid - per);
  if (sgn(abs(per)) != 0)
    inter.push_back(mid + per);
}
```

CircTangents.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", "Circle.h"   cec47e, 12 lines

```
int Tangents(Circle c1, Circle c2,
             vector<pair<Point, Point>>& out) {
  Point d = c2.c - c1.c;
  double dr = c1.r - c2.r, d2 = norm(d), h2 = d2 - dr * dr;
  if (sgn(d2) == 0 || sgn(h2) == -1) return 0;
  for (double z : {-1, 1}) {
    Point v = (d * dr + perp(d) * sqrt(h2) * z) / d2;
    out.emplace_back(c1 + v * c1.r, c2 + v * c2.r);
    if (sgn(h2) == 0) return 1;
  }
  return 2;
}
```

CircumCirc.h

Description:
The circumcircle of a triangle is the circle intersecting all three vertices. CircumRadius returns the radius of the circle going through points a, b and c and CircumCenter returns the center of the same circle.
"Circle.h"   057e5e, 20 lines

```
double CircumRadius(Point a, Point b, Point c) {
  return dist(a, b) * dist(b, c) * dist(c, a) /
    abs(det(a, b, c)) / 2.;
}
Point CircumCenter(Point a, Point b, Point c) {
  c = c - a; b = b - a;
  return a + perp(c*norm(b) - b*norm(c)) / cross(c, b) / 2.;
}
Circle CircumCircle(Point a, Point b, Point c) {
  Point p = CircumCenter(a, b, c);
  return {p, abs(p - a)};
}
// +1 if inside circle, 0 if on circle, -1 if outside.
int InsideCircum(Point p, Point a, Point b, Point c) {
  // (can be ll if coords are < 2e4)
  __int128_t p2 = norm(p), A = norm(a) - p2,
    B = norm(b) - p2, C = norm(c) - p2;
  return sgn(det(p, a, b) * C + det(p, b, c) * A
    + det(p, c, a) * B);
}
```

MEC.h
Description: Computes the minimum circle that encloses a set of points.
Time: expected $\mathcal{O}(N)$
   d6eaa5, 20 lines

```
Circle MEC(vector<Point>& pts) {
  assert(pts.size());
  shuffle(pts.begin(), pts.end(), rng);
  int n = pts.size();
  Point c = pts[0]; double r = 0;
  for (int i = 0; i < n; ++i) {
    if (abs(c - pts[i]) < r * (1 + EPS)) continue;
    c = pts[i]; r = 0.;
    for (int j = 0; j < i; ++j) {
      if (abs(c - pts[j]) < r * (1 + EPS)) continue;
      c = (pts[i] + pts[j]) * 0.5; r = abs(c - pts[i]);
      for (int k = 0; k < j; ++k) {
        if (abs(c - pts[k]) < r * (1 + EPS)) continue;
        c = CircumCenter(pts[i], pts[j], pts[k]);
        r = abs(c - pts[i]);
      }
    }
  }
  return {c, r};
}
```

8.4 Misc. Point Set Problems

PointLocation.h
Description: Given a list of points P, a list of segments E, and a list of query points Q, computes for each point the segment that is closest from each query point looking down (towards y = -INF). Returns -1 if no such segment is found, -2 if query point coincides with input point, or the index of the segment otherwise. Products are up to $O(X^2)$.
Time: $\mathcal{O}((N + Q) \log N)$
WARNING: Segments have to be non-intersecting and non-degenerate.
"Point.h"   18033c, 41 lines

```
vector<int> PointLocation(vector<Point>& P,
    vector<array<int, 2>>& E, vector<Point>& Q) {
  int n = P.size(), m = E.size(), q = Q.size();
  // Make sweepline events.
  vector<tuple<Point, int, int>> evs;
```

```
  for (int i = 0; i < m; ++i) {
    auto& [a, b] = E[i];
    if (P[b] < P[a]) swap(a, b);
    evs.emplace_back(P[a], 3, i),
    evs.emplace_back(P[b], 0, i);
  }
  for (int i = 0; i < n; ++i) evs.emplace_back(P[i], 1, i);
  for (int i = 0; i < q; ++i) evs.emplace_back(Q[i], 2, i);
  // Solve.
  sort(evs.begin(), evs.end());
  auto cmp = [&](int i, int j) {
    auto a = P[E[i][0]], b = P[E[i][1]],
         p = P[E[j][0]], q = P[E[j][1]];
    return det(a + a, b + b, p + q) > det(p + p, q + q, a + b);
  };
  set<int, decltype(cmp)> s(cmp);
  int last = -1;
  vector<int> ans(q);
  P.emplace_back(); E.emplace_back();
  for (auto [_, t, i] : evs) {
    if (t == 0) s.erase(i);
    if (t == 1) last = i;
    if (t == 2) {
      if (last != -1 && P[last] == Q[i]) ans[i] = -2;
      else {
        P[n] = Q[i]; E[m] = {n, n};
        auto it = s.lower_bound(m);
        if (it != s.end()) ans[i] = *it;
        else ans[i] = -1;
      }
    }
    if (t == 3) s.insert(i);
  }
  P.pop_back(); E.pop_back();
  return ans;
}
```

DCEL.h
Description: Computes the faces of a planar graph. Returns a permutation nxt where $nxt[i]$ is the next edge in the same face as edge i. Faces are trigonometric ordered. It doesn't compute a DCEL per se, but any other information can be recovered (origin is $E[i][0]$, prv can be derived from nxt, twin is $i^{\wedge}1$). To make it a little faster, short-circuit the tuple comparison at (*).
Time: $\mathcal{O}(M \log M)$
WARNING: E **has to contain both an edge and its reverse; moreover, the reverse of edge** i **has to be** $i^{\wedge}1$.   5a89d3, 20 lines

```
vector<int> DCEL(vector<Point>& P, vector<array<int, 2>>& E) {
  int n = P.size(), m = E.size();
  vector<int> ord(m);
  iota(ord.begin(), ord.end(), 0);
  stable_sort(ord.begin(), ord.end(), [&](int a, int b) {
    Point p = P[E[a][1]] - P[E[a][0]],
          q = P[E[b][1]] - P[E[b][0]];
    return make_tuple(E[a][0], half(p), 0) // (*)
      > make_tuple(E[b][0], half(q), cross(p, q));
  });
  int l = 0, r = 0;
  vector<int> nxt(m);
  for (l = 0; l < m; l = r) {
    for (r = l + 1; r < m && E[ord[r]][0] == E[ord[l]][0]; ++r)
      nxt[ord[r - 1]] = ord[r];
    nxt[ord[r - 1]] = ord[l];
  }
  for (int i = 0; i < m; i += 2) swap(nxt[i], nxt[i ^ 1]);
  return nxt;
}
```

ClosestPair.h

Description: Finds the closest pair of points. Returns the minimum **squared distance**, along with the points. Need to include proper operator< in namespace std (see Point.h).
Usage: auto [d, p, q] = ClosestPair(pts);
Time: $\mathcal{O}(N \log N)$

`"Point.h"` cc6b36, 18 lines

```cpp
tuple<long long, Point, Point> ClosestPair(vector<Point> v) {
  assert((int)v.size() > 1);
  sort(v.begin(), v.end(), [&](Point a, Point b) {
    return a.imag() < b.imag();
  });
  set<Point> s; int j = 0;
  tuple<long long, Point, Point> ret{4e18, {}, {}};
  for (auto p : v) {
    long long d = 1 + sqrt(get<0>(ret));
    while (v[j].imag() <= p.imag() - d) s.erase(v[j++]);
    auto lo = s.lower_bound(p - d),
         hi = s.upper_bound(p + d);
    for (auto it = lo; it != hi; ++it)
      ret = min(ret, {norm(*it - p), *it, p});
    s.insert(p);
  }
  return ret;
}
```

DynamicHull.h

Description: Dynamic convex hull tree. Useful for onion peeling. Can be made persistent by replacing the condition at (*) with true. Hull is trigonometric ordered and non-strict. To make it strict, play with the inequalities. To allow insertions as well, wrap it in a segment tree of integers and combine with the given function. Works in around 1.5 seconds for N = 200000. Does products up to $O(X^3)$.
Time: $\mathcal{O}(\log^2 N)$ per operation
WARNING: P has to be sorted (either increasing or decreasing) c6aa78, 74 lines

```cpp
struct DynHull {
  struct Node { int bl, br, l, r, lc, rc; };
  vector<Node> T = {{-1, -1, -1, -1, 0, 0}};
  vector<Point> P;

  DynHull(vector<Point> P) : P(P) {}

  bool leaf(int x) { return T[x].l == T[x].r; }
  int combine(int lc, int rc, int ret = -1) {
    if (!lc || !rc) return lc + rc;
    if (ret == -1 || ret == lc || ret == rc) // (*)
      ret = T.size(), T.push_back({});

    T[ret] = {-1, -1, T[lc].l, T[rc].r, lc, rc};
    while (!leaf(lc) || !leaf(rc)) {
      int a = T[lc].bl, b = T[lc].br,
          c = T[rc].bl, d = T[rc].br;
      if (a != b && det(P[a], P[b], P[c]) > 0) {
        lc = T[lc].lc;
      } else if (c != d && det(P[b], P[c], P[d]) > 0) {
        rc = T[rc].rc;
      } else if (a == b) {
        rc = T[rc].lc;
      } else if (c == d) {
        lc = T[lc].rc;
      } else {
        auto s1 = det(P[a], P[b], P[c]),
             s2 = det(P[a], P[b], P[d]);
        assert(s1 >= s2);
        auto xc = P[c].real(), xd = P[d].real(),
             xm = P[T[rc].l].real(), xa = P[a].real();
        if ((s1 * xd - s2 * xc < (s1 - s2) * xm) ^ (xa < xm)) {
          rc = T[rc].lc;
```

```cpp
      } else {
        lc = T[lc].rc;
      }
    }
  }
  T[ret].bl = T[lc].l; T[ret].br = T[rc].l;
  return ret;
}
// Build the hull from points P[l..r]
int Build(int l, int r) {
  if (l == r) {
    T.push_back({l, l, l, l, 0, 0});
    return T.size() - 1;
  }
  int m = (l + r) / 2;
  return combine(Build(l, m), Build(m + 1, r));
}
// Maximize dot product with p [set p = {x, 1} for CHT]
// UNTESTED: USE WITH CAUTION
int Maximize(int x, Point p) {
  assert(x);
  if (leaf(x)) return T[x].l; // can also return dot here
  return (dot(P[T[x].br], p) > dot(P[T[x].bl], p)
    ? Maximize(T[x].rc, p)
    : Maximize(T[x].lc, p));
}
// Erase P[i] from hull (if it exists)
int Erase(int x, int i) {
  if (!x || T[x].r < i || T[x].l > i) return x;
  return leaf(x) ? 0 : combine(
    Erase(T[x].lc, i), Erase(T[x].rc, i), x);
}
// Calls callback on all points of the hull.
template<typename Callback>
void Hull(int x, Callback&& cb, int l = 0, int r = 1e9) {
  if (!x || l > r) return;
  if (leaf(x)) { cb(T[x].l); return; }
  Hull(T[x].lc, cb, max(l, T[x].l), min(r, T[x].bl));
  Hull(T[x].rc, cb, max(l, T[x].br), min(r, T[x].r));
}
};
```

HalfplaneSet.h

Description: Data structure that dynamically keeps track of the intersection of halfplanes. Use is straightforward. Area should be able to be kept dynamically with some modifications. Does products up to $O(X^4)$.
Usage: HalfplaneSet hs;
hs.Cut({0, 1, 2});
double best = hs.Maximize({1, 2});
Time: $\mathcal{O}(\log N)$ amortized per operation
`"Line.h"`, `"LineIntersection.h"` 294bc8, 44 lines

```cpp
struct HalfplaneSet : multiset<Line> {
  HalfplaneSet() {
    insert({+1, 0, INF}); insert({0, +1, INF});
    insert({-1, 0, INF}); insert({0, -1, INF});
  };

  auto adv(auto it, int z) { // z = {-1, +1}
    return (z == -1
      ? --(it == begin() ? end() : it)
      : (++it == end() ? begin() : it));
  }
  bool chk(auto it) {
    Line l = *it, pl = *adv(it, -1), nl = *adv(it, +1);
    auto [x, y, d] = LineIntersection(pl, nl);
    T4 sat = l.a * x + l.b * y - (T4)l.c * d;
    if (d < 0 && sat < 0) return clear(), 0; // unsat
    if ((d > 0 && sat <= 0) || (d == 0 && sat < 0))
      return erase(it), 1;
```

```cpp
    return 0;
  }
  void Cut(Line l) { // add ax + by <= c
    if (empty()) return;
    auto it = insert(l);
    if (chk(it)) return;
    for (int z : {-1, +1})
      while (size() && chk(adv(it, z)));
  }
  double Maximize(T a, T b) { // max ax + by
    if (empty()) return -1/0.;
    auto it = lower_bound({a, b});
    if (it == end()) it = begin();
    auto [x, y, d] = LineIntersection(*adv(it, -1), *it);
    return (1.0 * a * x + 1.0 * b * y) / d;
  }
  double Area() {
    double total = 0.;
    for (auto it = begin(); it != end(); ++it) {
      auto [x1, y1, d1] = LineIntersection(*adv(it, -1), *it);
      auto [x2, y2, d2] = LineIntersection(*it, *adv(it, +1));
      total += (1.0 * x1 * y2 - 1.0 * x2 * y1) / d1 / d2;
    }
    return total * 0.5;
  }
};
```

RotatePlane.h

Description: Rotates plane, keeping points in order of y-coordinate. Initially plane is vertical and slightly rotated ccw. If no 3 points are collinear, the logic at (*) can be replaced with a simple swap in permutation.
Time: $\mathcal{O}(N^2 \log N)$
`"Point.h"` 529fe5, 42 lines

```cpp
void RotatePlane(vector<Point> P) {
  int n = P.size();
  // Make swap events.
  vector<array<int, 2>> evs;
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      if (P[i] < P[j])
        evs.push_back({i, j});
  int m = evs.size();
  auto cmp = [&](auto a, auto b) {
    return cross(P[a[1]] - P[a[0]], P[b[1]] - P[b[0]]) > 0; };
  stable_sort(evs.begin(), evs.end(), cmp);
  // ord is order of points, pos is the inverse permutation.
  vector<int> ord(n), pos(n);
  iota(ord.begin(), ord.end(), 0);
  sort(ord.begin(), ord.end(), [&](int a, int b) {
    return P[a] < P[b];
  });
  for (int i = 0; i < n; ++i) pos[ord[i]] = i;
  // Do radial sweep.
  vector<bool> mark(n, false);
  for (int half : {0, 1}) // you might not need it
  for (int i = 0, j = 0; i < m; i = j) { // (*)
    for (j = i; j < m && !cmp(evs[i], evs[j]); ++j);
    for (int k = i; k < j; ++k) {
      auto [a, b] = evs[k];
      mark[min(pos[a], pos[b])] = true;
    }
    for (int k = i; k < j; ++k) {
      auto [a, b] = evs[k];
      a = pos[a]; b = pos[b];
      if (a > b) swap(a, b);
      if (b != a + 1 || !mark[a]) continue;
      while (a > 0 && mark[a - 1]) --a;
      while (b + 1 < n && mark[b]) ++b;
      // ord[a], ord[a+1], ..., ord[b] are the sweepline here.
```

Column 1

```
      reverse(ord.begin() + a, ord.begin() + b + 1);
      for (int l = a; l <= b; ++l) pos[ord[l]] = l;
      for (int l = a; l < b; ++l) mark[l] = false;
    }
  }
}
```

KDTree.h
Description: KD-tree (2d, can be trivially extended to kd). Not sure how it works for duplicate points. Takes about 0.7s for 10^6 **random** queries.
<div align="right">0de3c1, 40 lines</div>

```cpp
using Point = array<int, 2>;

struct KDTree {
  int n;
  vector<Point> P;

  KDTree(vector<Point> P) : n(P.size()), P(P) {
    build(0, n, 0);
  }
  ll sqr(ll x) { return x * x; }
  void build(int b, int e, int spl) {
    if (e <= b) return;
    int m = (b + e) / 2;
    nth_element(P.begin() + b, P.begin() + m, P.begin() + e,
      [&](auto& p, auto& q) { return p[spl] < q[spl]; });
    build(b, m, !spl);
    build(m + 1, e, !spl);
  }
  Point Nearest(Point p, bool exclude_me = false) {
    ll best = 5e18; Point ans;
    auto go = [&](auto& self, int b, int e, int spl) {
      if (e <= b) return;
      int m = (b + e) / 2;
      auto& q = P[m];
      ll now = sqr(q[0] - p[0]) + sqr(q[1] - p[1]);
      if (now >= exclude_me && now < best) best = now, ans = q;
      if (p[spl] < q[spl]) {
        self(self, b, m, !spl);
        if (q[spl] - p[spl] < 1e-5 + sqrt(best))
          self(self, m + 1, e, !spl);
      } else {
        self(self, m + 1, e, !spl);
        if (p[spl] - q[spl] < 1e-5 + sqrt(best))
          self(self, b, m, !spl);
      }
    };
    go(go, 0, n, 0); // y-combinator is considerably faster.
    return ans;
  }
};
```

Delaunay.h
Description: Computes the Delaunay triangulation of a set of points. Triangles are in counter-clockwise order. Answer is unique if the set of points is in general position. Products are up to $O(X^4)$.
Time: $\mathcal{O}\left(N^2\right)$
<div align="right">"ConvexHull3D.h"   b88573, 11 lines</div>

```cpp
vector<array<int, 3>> Delaunay(vector<Point> P) {
  int n = P.size();
  vector<D3::Point> Q(n);
  for (int i = 0; i < n; ++i)
    Q[i] = {P[i].real(), P[i].imag(), norm(P[i])};
  vector<array<int, 3>> ret;
  for (auto [a, b, c] : D3::Hull3D(Q))
    if (D3::det(Q[a], Q[b], Q[c], {0, 0, 1}) < 0)
      ret.push_back({a, b, c});
  return ret;
}
```

Column 2

FastDelaunay.h
Description: Fast Delaunay triangulation. Each circumcircle contains none of the input points. 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 'InsideCircle'. Returns triangles in counter-clockwise order.
Time: $\mathcal{O}\left(n \log n\right)$
<div align="right">"Point.h", "CircumCircle.h"   d3f747, 92 lines</div>

```cpp
using Q = struct Quad*;
struct Quad {
  Q rot, o; int p = 0; bool mark;
  int& F() { return r()->p; }
  Q& r() { return rot->rot; }
  Q prev() { return rot->o->rot; }
  Q next() { return r()->prev(); }
} *H;

struct Delaunay {
  vector<Point> P; int n;

  Delaunay(vector<Point>& _P) : P(_P), n(_P.size()) {
    P.push_back(P.back() + Point{1, 0}); // any point not in P
  }
  Q add(int orig, int dest) {
    Q r = H ?: new Quad{new Quad{new Quad{new Quad{0}}}};
    H = r->o; r->r()->r() = r;
    for (int i = 0; i < 4; ++i)
      r = r->rot, r->p = n, r->o = i & 1 ? r : r->r();
    r->p = orig; r->F() = dest;
    return r;
  }
  void splice(Q a, Q b) {
    swap(a->o->rot->o, b->o->rot->o);
    swap(a->o, b->o);
  }
  Q connect(Q a, Q b) {
    Q q = add(a->F(), b->p);
    splice(q, a->next());
    splice(q->r(), b);
    return q;
  }
#define H(e) P[e->F()], P[e->p]
#define valid(e) (det(P[e->F()], H(base)) > 0)
  pair<Q, Q> divide(int b, int e) {
    if (e - b <= 3) {
      Q qa = add(b, b + 1), qb = add(b + 1, e - 1);
      if (e - b == 2) return {qa, qa->r()};
      splice(qa->r(), qb);
      auto side = det(P[b], P[b + 1], P[b + 2]);
      Q qc = side ? connect(qb, qa) : 0;
      return {side < 0 ? qc->r() : qa,
              side < 0 ? qc : qb->r()};
    }
    Q A, B, ra, rb;
    int m = (b + e) / 2;
    tie(ra, A) = divide(b, m);
    tie(B, rb) = divide(m, e);
    while ((det(P[B->p], H(A)) < 0 && (A = A->next())) ||
           (det(P[A->p], 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 (InsideCircum(P[e->dir->F()], H(base), P[e->F()]) > \
           0) { \
      Q t = e->dir; \
      splice(e, e->prev()); \
      splice(e->r(), e->r()->prev()); \
```

Column 3

```cpp
      e->o = H; H = e; e = t; \
    }
    while (true) {
      DEL(LC, base->r(), o);  DEL(RC, base, prev());
      if (!valid(LC) && !valid(RC)) break;
      if (!valid(LC) || (valid(RC) &&
          InsideCircum(H(RC), H(LC)) > 0))
        base = connect(RC, base->r());
      else
        base = connect(base->r(), LC->r());
    }
    return {ra, rb};
  }

  vector<array<int, 3>> Triangulate() {
    if (P.size() < 2) return {};
    Q e = divide(0, n).first;
    while (det(P[e->o->F()], P[e->F()], P[e->p]) < 0) e = e->o;
    vector<Q> q = {e};
    vector<array<int, 3>> ret;
    for (int qi = 0; qi < (int)q.size(); ++qi) {
      if ((e = q[qi])->mark) continue;
      ret.emplace_back();
      for (int i = 0; i < 3; ++i) {
        e->mark = 1; ret.back()[i] = e->p;
        q.push_back(e->r()); e = e->next();
      }
    }
    ret.erase(ret.begin());
    return ret;
  }
};
```

8.5 3D

Point3D.h
Description: Class to handle points in 3D space. T can be e.g. double or long long.
<div align="right">f92aff, 29 lines</div>

```cpp
using T = double;
using Face = array<int, 3>;

struct Point {
  T x = 0, y = 0, z = 0;
  Point operator+(Point o) { return { x+o.x, y+o.y, z+o.z }; }
  Point operator-(Point o) { return { x-o.x, y-o.y, z-o.z }; }
  Point operator*(T d) { return { x*d, y*d, z*d }; }
  Point operator/(T d) { return { x/d, y/d, z/d }; }
};
// Point orth to the span of (a, b) by right hand rule.
// abs() is equal to 2 * area of triangle.
Point cross(Point a, Point b) {
  return {a.y * b.z - a.z * b.y,
          a.z * b.x - a.x * b.z,
          a.x * b.y - a.y * b.x};
}
T dot(Point a, Point b) { return a.x*b.x + a.y*b.y + a.z*b.z; }
T norm(Point p) { return dot(p, p); }
T abs(Point p) { return sqrt(norm(p)); }
// 6 * signed volume of tetrahedron.
T det(Point a, Point b, Point c, Point d) {
  return dot(a - d, cross(b - d, c - d));
}
//returns point rotated 'theta' radians ccw around axis a
Point rotate_ccw(Point p, Point a, double theta) {
  double s = sin(theta), c = cos(theta); Point u = a / abs(a);
  return u * dot(p, u) * (1-c) + p * c - cross(p, u) * s;
}
```

ConvexHull3D.h

Description: Incrementally computes all faces of the 3-dimension hull of a point set. **Ideally, no four points must be coplanar**, or else random results will be returned. All faces will point outwards. To optimize, cache cross products. For integer coordinates, products of up to X^3 are made.
Time: $\mathcal{O}\left(N^2\right)$

"Point3D.h"   ac26a3, 24 lines

```cpp
vector<Face> Hull3D(vector<Point> P) {
  int n = P.size(); assert(n >= 3);
  vector<vector<bool>> dead(n, vector<bool>(n));
  vector<Face> ret = {{0, 1, 2}, {2, 1, 0}}, nret;
  for (int i = 3; i < n; ++i) {
    nret.clear();
    for (auto f : ret) {
      auto [a, b, c] = f;
      if (det(P[a], P[b], P[c], P[i]) > 0) // consider sgn()
        dead[a][b] = dead[b][c] = dead[c][a] = true;
      else nret.push_back(f);
    }
    ret.clear();
    for (auto f : nret) {
      ret.push_back(f);
      for (int j = 0, k = 2; j < 3; k = j++) {
        int a = f[k], b = f[j];
        if (dead[b][a]) ret.push_back({b, a, i});
        dead[b][a] = false;
      }
    }
  }
  return ret;
}
```

PolyVolume3D.h

Description: Computes the (signed) volume of a polyhedron. Faces should point outwards.

79f037, 5 lines

```cpp
T Volume(vector<Point> v, vector<Face> faces) {
  double ret = 0; Point O{0, 0, 0}; // origin
  for (auto [a, b, c] : faces) ret += det(v[c], v[b], v[a], O);
  return ret / 6;
}
```

8.6 Math

8.6.1 Pick's Theorem

If a polygon has vertices on integer coordinates, then:

$$A = I + B/2 - 1$$

where A is the polygon area, B is the number of boundary points (on vertices + edges), and I is the number of interior points (strict).

8.6.2 Euler's Formula

If a finite, planar graph is drawn in the plane without any edge intersections, and V is the number of vertices, E is the number of edges and F is the number of faces (regions bounded by edges, including the outer, infinitely large region), then:

$$V - E + F = 2$$

$$E \leq 3V - 6$$

A more general formula for K connected components is:

$$V - E + F = 1 + K$$

Strings (9)

KMP.h

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

b0d18a, 20 lines

```cpp
vector<int> ComputePi(string s) {
  int n = s.size();
  vector<int> pi(n + 1, -1);
  for (int i = 0; i < n; ++i) {
    int j = pi[i];
    while (j != -1 && s[j] != s[i]) j = pi[j];
    pi[i + 1] = j + 1;
  }
  return pi;
}

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

ZFunction.h

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

9b1f7a, 10 lines

```cpp
vector<int> ZFunction(string s) {
  int n = s.size();
  vector<int> z(n);
  for (int i = 1, l = 0, r = 0; i < n; ++i) {
    if (i <= r) z[i] = min (r - i + 1, z[i - l]);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
    if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
  }
  return z;
}
```

MinRotation.h

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

8a84f6, 11 lines

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

AhoCorasick.h

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

80641a, 34 lines

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

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

  // Adds a word to trie and returns the end node
  int Add(const string &s) {
    int node = 0;
    for (auto c : s) {
      auto &nxt = T[node].leg[c];
      if (nxt == 0) nxt = nodes++;
      node = nxt;
    }
    return node;
  }
  // Advances from a node with a character (like an automaton)
  int Go(int node, char c) {
    while (node != -1 && !T[node].leg.count(c))
      node = T[node].link;
    return (node == -1 ? 0 : T[node].leg[c]);
  }
  // Builds links
  void Build() {
    T[0].link = -1, q.push_back(0);
    for (int i = 0; i < (int)q.size(); ++i) {
      int node = q[i];
      for (auto [c, vec] : T[node].leg)
        T[vec].link = Go(T[node].link, c), q.push_back(vec);
    }
  }
};
```

SuffixAutomaton.h

Description: Builds an automaton of all the suffixes of a given string (online from left to right). To support multiple strings/trie, you can add characters in BFS order of trie and it should work.
Usage: last = 0; for (auto c : s) last = SA.Add(last, c);
Time: $\mathcal{O}(\log \Sigma)$ amortized per character added

fb042f, 28 lines

```cpp
struct SuffixAutomaton {
  struct Node {
    int len, link;
    map<char, int> leg; // Can use array<int, 26> instead.
  };
  vector<Node> T = {{0, -1, {}}};
  // Adds another character to the automaton.
  int Add(int last, char c) {
    int node = last, cur = T.size();
    T.push_back({T[last].len + 1, 0, {}});
    while (node != -1 && !T[node].leg[c])
      T[node].leg[c] = cur, node = T[node].link;
    if (node != -1) {
      int old = T[node].leg[c], len = T[node].len + 1;
      assert(T[cur].len >= T[old].len);
      if (T[old].len == len) {
        T[cur].link = old;
      } else {
        int clone = T.size();
        T.push_back({len, T[old].link, T[old].leg});
        T[old].link = T[cur].link = clone;
        while (node != -1 && T[node].leg[c] == old)
          T[node].leg[c] = clone, node = T[node].link;
```

```
        }
    }
    return cur;
  }
};
```

Manacher.h
Description: Given a string s, computes the length of the longest palindromes centered in each position (for parity == 1) or between each pair of adjacent positions (for parity == 0).
Usage: Manacher("abacaba", 1) => {0, 1, 0, 3, 0, 1, 0}
Manacher("aabbaa", 0) => {1, 0, 3, 0, 1}
Time: $\mathcal{O}(N)$
9c6d91, 13 lines

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

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

PalindromicTree.h
Description: A trie-like structure for keeping track of palindromes of a string s. It has two roots, 0 (for even palindromes) and 1 (for odd palindromes). Each node stores the length of the palindrome, the count and a link to the longest "aligned" subpalindrome. Can be made online from left to right
Time: $\mathcal{O}(N)$
50613b, 45 lines

```cpp
struct Node {
  map<char, int> leg;
  int link, len, cnt = 0;
};

vector<Node> PalTree(string str) {
  vector<Node> T(str.size() + 2);
  T[1].link = T[1].len = 0;
  T[0].link = T[0].len = -1;
  int last = 0, nodes = 2;

  for (int i = 0; i < (int)str.size(); ++i) {
    char now = str[i];
    int node = last;
    while (now != str[i - T[node].len - 1])
      node = T[node].link;
    if (T[node].leg.count(now)) {
      node = T[node].leg[now];
      T[node].cnt += 1;
      last = node;
      continue;
    }
    int cur = nodes++;
    T[cur].len = T[node].len + 2;
    T[node].leg[now] = cur;
    int link = T[node].link;
    while (link != -1) {
      if (now == str[i - T[link].len - 1] &&
          T[link].leg.count(now)) {
        link = T[link].leg[now];
        break;
      }
      link = T[link].link;
```

```cpp
    }
    if (link <= 0) link = 1;
    T[cur].link = link;
    T[cur].cnt = 1;
    last = cur;
  }
  for (int node = nodes - 1; node > 0; --node)
    T[T[node].link].cnt += T[node].cnt;

  T.resize(nodes);
  return T;
}
```

Various (10)

AlphaBeta.h
Description: Uses the alpha-beta pruning method to find score values for states in games (minimax). Works faster if better states are explored first.
a5c6d7, 8 lines

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

Hashing.h
Description: Arithmetic mod $2^{64} - 1$. 2x slower than mod 2^{64} and more code, but works on evil test data (e.g. Thue-Morse, where ABBA... and BAAB... of length 2^{10} hash the same mod 2^{64}). "typedef ull H;" instead if you think test data is random, or work mod $10^9 + 7$ if the Birthday paradox is not a problem.
654f55, 11 lines

```cpp
struct H {
  typedef uint64_t ull;
  ull x; H(ull x=0) : x(x) {}
#define OP(O,A,B) H operator O(H o) { ull r = x; asm \
  (A "addq %%rdx, %0\n adcq $0,%0" : "+a"(r) : B); return r; }
  OP(+,,"d"(o.x)) OP(*,"mul %1\n", "r"(o.x) : "rdx")
  H operator-(H o) { return *this + ~o.x; }
  ull get() const { return x + !~x; }
  bool operator==(H o) const { return get() == o.get(); }
  bool operator<(H o) const { return get() < o.get(); }
};
```

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: $\mathcal{O}(\log N)$
49551a, 28 lines

```cpp
struct IntervalContainer : map<int, int> {
  iterator AddInterval(int l, int r) {
    if (l == r) return end();
    auto it = lower_bound(l);
    while (it != end() && it->first <= r) {
      r = max(r, it->second);
      it = erase(it);
    }
    while (it != begin() && (--it)->second >= l) {
      l = min(l, it->first), r = max(r, it->second);
      it = erase(it);
    }
    return emplace(l, r).first;
  }
```

```cpp
  iterator FindInterval(int x) {
    auto it = upper_bound(x);
    if (it == begin() || (--it)->second <= x)
      return end();
    return it;
  }
  void RemoveInterval(int l, int r) {
    if (l == r) return;
    auto it = AddInterval(l, r);
    auto [l2, r2] = *it; erase(it);
    if (l != l2) emplace(l2, l);
    if (r != r2) emplace(r, r2);
  }
};
```

ConstantIntervals.h
Description: Split a monoene function on [b, e) into a minimal set of half-open intervals on which it has the same value. Runs a callback cb for each such interval.
Usage: ConstantIntervals(0, v.size(), [&](int x){return v[x];}, [&](int lo, int hi, T val){...});
Time: $\mathcal{O}\left(k \log \frac{n}{k}\right)$
577a4c, 20 lines

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

10.2 Dynamic programming

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

10.3 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 failures 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**).

10.4 Optimization tricks

`__builtin_ia32_ldmxcsr(40896);` disables denormals (which make floats 20x slower near their minimum value).

10.4.1 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.

10.4.2 Pragmas

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

- `#pragma GCC target ("avx2")` can double performance of vectorized code, but causes crashes on old machines.

- `#pragma GCC optimize ("trapv")` kills the program on integer overflows (but is really slow).

Montgomery.h
Description: Faster ModInt multiplication using Montgomery space. Copy other operators from "ModInt.h" if needed.
WARNING: Using the ModInt wrapper might make the operations slower. Refactor at your own risk.
                                            128ce9, 40 lines

```
// Set T2 such that (T2)mod * mod does not overflow.
using T = uint32_t;
using T2 = uint64_t;
const int BITS = 8 * sizeof(T); // = 32
const T MOD = 998244353, INV = 3296722945, R2 = 932051910;
//            970592641         905457281         670047489
//            1000000007        2068349879        582344008
//       3006703054056749 11535267960557300389  886200325609766
//   1000000000000000003 14584756025394899627 527506668952824130

T reduce(T2 x) {
  T q = (T)x * INV, a = (x - (T2)q * MOD) >> BITS;
  if (a >= MOD) a += MOD; // not a mistake
  return a;
}
T mul(T a, T b) { return reduce((T2)a * b); }
// Only required for unknown modulo. Call before everything.
void init(T mod) {
  MOD = mod; INV = 1; R2 = -mod % mod;
  for (int i = 1; i < BITS; i *= 2)
    INV *= 2 - MOD * INV;
  for (int i = 0; i < 4; i++)
    if ((R2 *= 2) >= mod)
      R2 -= mod;
  for (int i = 4; i < BITS; i *= 2)
    R2 = mul(R2, R2);
}

struct ModInt {
  T x;
  ModInt(T x) : x(mul(x, R2)) {}
```

```
  ModInt ctr(T x) { // unsafe internal constructor
    if (x >= MOD) x -= MOD;
    ModInt ret(*this); ret.x = x; return ret;
  }
  ModInt operator+(ModInt oth) { return ctr(x + oth.x); }
  ModInt operator-(ModInt oth) { return ctr(x + MOD - oth.x); }
  ModInt operator*(ModInt oth) { return ctr(mul(x, oth.x)); }
  T Get() { return reduce(x); }
};
```

FastInput.h
Description: Read an integer from stdin. Usage requires your program to pipe in input from file.
Usage: ./a.out < input.txt
Time: About 5x as fast as cin/scanf.   7b3c70, 17 lines

```
inline char gc() { // like getchar()
  static char buf[1 << 16];
  static size_t bc, be;
  if (bc >= be) {
    buf[0] = 0, bc = 0;
    be = fread(buf, 1, sizeof(buf), stdin);
  }
  return buf[bc++]; // returns 0 on EOF
}

int readInt() {
  int a, c;
  while ((a = gc()) < 40);
  if (a == '-') return -readInt();
  while ((c = gc()) >= 48) a = a * 10 + c - 480;
  return a - 48;
}
```