Universidade Federal de Pernambuco

las4s e pelados

Icaro Copo Papel Nunes, Joao Pou Grangeiro, Pedro Grisi

2026-02-10

Contest (1)

template.cpp — 9 lines

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

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

.bashrc — 2 lines

```
alias c='g++ -Wall -Wconversion -Wfatal-errors -g -std=c++17 \
  -fsanitize=undefined,address'
```

hash.sh — 2 lines

```
# bash hash.sh file.cpp l1 l2
sed -n $2','$3' p' $1 | sed '/^#w/d' | cpp -dD -P -fpreprocessed | tr -d '[:space:]'| md5sum |cut -c-6
```

troubleshoot.txt — 52 lines

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

Wrong answer:
Print your solution! Print debug output, as well.
Are you clearing all 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?
```

Theoretical (2)

2.1 Mathematics

2.1.1 Recurrences

If $a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k}$, and $r_1, \ldots, r_k$ are distinct roots of $x^k - c_1 x^{k-1} - \cdots - c_k$, there are $d_1, \ldots, d_k$ s.t.

$$a_n = d_1 r_1^n + \cdots + d_k r_k^n.$$

Non-distinct roots r become polynomial factors, e.g. $a_n = (d_1 n + d_2) r^n$.

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

2.1.3 Geometry

Triangles

Side lengths: a, b, c

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

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

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

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

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

Length of bisector (divides angles in two): $s_a = \sqrt{bc\left[1 - \left(\frac{a}{b+c}\right)^2\right]}$

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

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

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

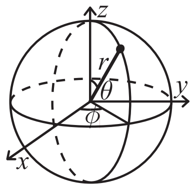
Quadrilaterals

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

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

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

Spherical coordinates



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

Pick's Theorem

The area of a simple polygon whose vertices have integer coordinates is:

$$A = I + \frac{B}{2} - 1$$

where I is the number of interior integer points, and B is the number of integer points in the border of the polygon.

Two Ears Theorem

Every simple polygon with more than 3 vertices has at least two non-overlapping ears (a ear is a vertex whose diagonal induced by its neighbors which lies strictly inside the polygon). Equivalently, every simple polygon can be triangulated.

2.1.4 Derivatives/Integrals

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

Integration by parts:

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

2.1.5 Sums

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

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

$$\sum_{i=0}^{n} ic^i = \frac{nc^{n+2}-(n+1)c^{n+1}+c}{(c-1)^2},\ c\neq 1$$

$$g_k(n) = \sum_{i=1}^{n} i^k = \frac{1}{k+1}\left(n^{k+1} + \sum_{j=1}^{k}\binom{k+1}{j+1}(-1)^{j+1}g_{k-j}(n)\right)$$

2.1.6 Series

$$
\begin{aligned}
e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots,\ (-\infty < x < \infty) \\
\ln(1+x) &= x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots,\ (-1 < x \le 1) \\
\sqrt{1+x} &= 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \ldots,\ (-1 \le x \le 1) \\
\sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots,\ (-\infty < x < \infty) \\
\cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots,\ (-\infty < x < \infty) \\
\sum_{i=0}^{\infty} ic^i &= \frac{c}{(1-c)^2},\quad |c| < 1 \\
(1+x)^n &= \sum_{i=0}^{n}\binom{n}{i}x^i \\
\frac{1}{1-x} &= \sum_{i=0}^{\infty} x^i,\ (-1 < x < 1) \\
\frac{1}{(1-x)^n} &= \sum_{i=0}^{\infty}\binom{n+i-1}{n-1}x^i,\ (-1 < x < 1)
\end{aligned}
$$

2.1.7 Probability theory

Let X be a discrete random variable with probability $p_X(x)$ of assuming the value x. It will then have an expected value (mean) $\mu = \mathbb{E}(X) = \sum_x xp_X(x)$ and variance $\sigma^2 = V(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = \sum_x (x-\mathbb{E}(X))^2 p_X(x)$ where σ is the standard deviation. If X is instead continuous it will have a probability density function $f_X(x)$ and the sums above will instead be integrals with $p_X(x)$ replaced by $f_X(x)$.

Expectation is linear:

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

For independent X and Y,

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

Binomial distribution

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

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

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

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

First success distribution

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

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

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

Poisson distribution

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

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

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

2.2 Combinatorial

2.2.1 Binomial Identities

$$
\begin{aligned}
\tbinom{n-1}{k} - \tbinom{n-1}{k-1} &= \tfrac{n-2k}{k}\tbinom{n}{k} & \tbinom{n}{h}\tbinom{n-h}{k} &= \tbinom{n}{k}\tbinom{n-k}{h} \\
\textstyle\sum_{k=0}^{n} k\tbinom{n}{k} &= n2^{n-1} & \textstyle\sum_{k=0}^{n} k^2\tbinom{n}{k} &= (n+n^2)2^{n-2} \\
\textstyle\sum_{j=0}^{k}\tbinom{m}{j}\tbinom{n-m}{k-j} &= \tbinom{n}{k} & \textstyle\sum_{j=0}^{m}\tbinom{m}{j}^2 &= \tbinom{2m}{m} \\
\textstyle\sum_{m=0}^{n}\tbinom{m}{j}\tbinom{n-m}{k-j} &= \tbinom{n+1}{k+1} & \textstyle\sum_{m=0}^{n}\tbinom{m}{k} &= \tbinom{n+1}{k+1} \\
\textstyle\sum_{r=0}^{m}\tbinom{n+r}{r} &= \tbinom{n+m+1}{m} & \textstyle\sum_{k=0}^{n}\tbinom{n-k}{k} &= \mathrm{Fib}(n+1) \\
\textstyle\sum_{k=0}^{n}\tbinom{r}{k}\tbinom{s}{n-k} &= \tbinom{r+s}{n}
\end{aligned}
$$

2.2.2 Permutations

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

Cycles

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

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

Derangements

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

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

Burnside's lemma

Counts the number of distinct colorings of an object under symmetry.

$$\frac{1}{|G|}\sum_{g\in G} k^{\mathrm{cyc}(g)},$$

where G is the symmetry group, k the number of colors, and $\mathrm{cyc}(g)$ the number of cycles induced by g.

Example: number of ways to color a necklace with n beads using k colors (rotations only):

$$g(n) = \frac{1}{n}\sum_{i=0}^{n-1} k^{(\gcd(n,i))}$$

where rotation i shifts the necklace by i positions.

2.2.3 Partitions and subsets

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

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

2.2.4 Sum of Binomials (FFT)

Goal: Given freq. array C, compute $Ans[k] = \sum_i C[i]\binom{i}{k}$ for all k. Rewrite: $Ans[k] = \frac{1}{k!}\sum_i (C[i]\cdot i!)\frac{1}{(i-k)!}$.

- Construct P where $P[i] = C[i]\cdot i!$
- Construct Q where $Q[i] = (i!)^{-1}$
- Reverse Q (to handle the $i-k$ subtraction).
- Multiply $R = NTT(P,Q)$.
- Result: $Ans[k] = R[k+|Q|-1]\cdot\frac{1}{k!}$.

2.2.5 General purpose numbers

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:

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

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

Stirling numbers of the first kind

Number of permutations on n items with k cycles.

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

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

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. $\pi(j) > j$.

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

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

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

Stirling numbers of the second kind

Partitions of n distinct elements into exactly k groups.

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

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

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

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,

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

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: $(n-2)!/((d_1-1)!\cdots(d_n-1)!)$

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.

2.3 Number Theory

2.3.1 Bézout's identity

For $a\neq, b\neq 0$, then $d = gcd(a,b)$ is the smallest positive integer for which there are integer solutions to

$$ax + by = d$$

If (x,y) is one solution, then all solutions are given by

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

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

2.3.3 Estimates

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

The number of divisors of n is at most around 100 for $n < 5e4$, 500 for $n < 1e7$, 2000 for $n < 1e10$, 6700 for $n < 1e12$, 200 000 for $n < 1e19$.

2.3.4 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)

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

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

2.3.5 Theorems

Goldbach's conjecture: Every even integer $n > 2$ can be written as $n = a + b$ with a, b prime.

Legendre's conjecture: There is always at least one prime between n^2 and $(n+1)^2$.

Lagrange's four-square theorem: Every positive integer can be written as

$$n = a^2 + b^2 + c^2 + d^2.$$

Zeckendorf's theorem: Every integer $n \ge 1$ has a unique representation as a sum of non-consecutive Fibonacci numbers:

$$n = F_{i_1} + F_{i_2} + \cdots + F_{i_k}, \quad i_j - i_{j+1} \ge 2.$$

Euclid's formula (primitive Pythagorean triples): The Pythagorean triples are uniquely generated by

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

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

Wilson's theorem: n is prime iff

$$(n-1)! \equiv -1 \pmod{n}.$$

Chicken McNugget theorem: For coprime n, m, the largest integer not representable as $an + bm$ (with $a, b \ge 0$) is

$$nm - n - m.$$

There are $\frac{(n-1)(m-1)}{2}$ non-representable integers, and for each pair $(k,\ nm - n - m - k)$ exactly one is representable.

2.4 Graphs

2.4.1 Flows and Matching

Hall's Theorem

In bipartite graphs, there exists a perfect matching covering the entire side X if and only if for every subset $Y \subseteq X$,

$$|Y| \le |N(Y)|,$$

where $N(Y)$ denotes the set of neighbors of Y.

Kőnig's Theorem

In a bipartite graph, the size of a Minimum Vertex Cover is equal to the size of a Maximum Matching. A Minimum Vertex Cover is a minimum set of vertices such that every edge of the graph has at least one endpoint in the set.

As a consequence,

$$n - \text{Maximum Matching} = \text{Maximum Independent Set},$$

where a Maximum Independent Set is the largest set of vertices with no edges between them.

Recovering the Minimum Vertex Cover Given a maximum matching in a bipartite graph (X, Y):

- Construct the residual graph by orienting:
  - non-matching edges from X to Y;
  - matching edges from Y to X.
- Perform a BFS or DFS starting from all free (unmatched) vertices in X.
- Let Z_X be the set of reachable vertices in X, and Z_Y the set of reachable vertices in Y.

The Minimum Vertex Cover is given by:

$$(X \setminus Z_X) \cup Z_Y.$$

Node-Disjoint Path Cover

A node-disjoint path cover is a set of paths such that each vertex belongs to exactly one path.

In a directed acyclic graph (DAG),

$$\text{Minimum Node-Disjoint Path Cover} = n - \text{Maximum Matching}.$$

The construction is as follows: for each vertex u, create a copy u'. Add an edge $u \to v'$ if there exists an edge $u \to v$ in the original graph.

Recovering the Paths

- Vertices that do not appear as destinations in the matching are starting points of paths.
- Each matching edge $u \to v'$ corresponds to an edge $u \to v$ in the original DAG.
- Following these edges reconstructs all paths of the path cover.

General Path Cover

A general path cover is a path cover where a vertex may belong to more than one path.

In a DAG, the construction is similar to the node-disjoint case, but an edge $u \to v'$ exists if there is a path from u to v in the original graph.

Recovering the Cover The vertices can be grouped according to the edges used in the matching to form the path cover.

Dilworth's Theorem

An antichain is a set of vertices such that there is no path between any pair of vertices in the set.

In a directed acyclic graph,

$$\text{Minimum General Path Cover} = \text{Maximum Antichain}.$$

Recovering a Maximum Antichain Given a minimum general path cover, selecting one vertex from each path produces a maximum antichain.

2.4.2 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).

2.4.3 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 $k = 1 \dots n$,

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

2.4.4 Planar Graphs

If G has k connected components, then $n - m + f = k + 1$.

2.5 Optimization tricks

2.5.1 Bit hacks

- `for (int x = m; x; x = (x - 1)&m ) { ... }` loops over all subset masks of `m` (except 0).
- `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.

2.5.2 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 target("bmi,bmi2,popcnt,lzcnt")` improve bit operations.
- `#pragma GCC optimize("unroll-loops")` self explanatory.

2.6 Various

2.6.1 Master Theorem (Simple)

$T(n) = aT(n/b) + O(n^d)$. Compare a vs b^d:

- $a > b^d \implies O(n^{\log_b a})$ (Work at leaves dominates)
- $a = b^d \implies O(n^d \log n)$ (Work is uniform)
- $a < b^d \implies O(n^d)$ (Work at root dominates)

Data structures (3)

Bit.h

Description: *lower_bound* works the same as on vectors
Time: $\mathcal{O}(\log N)$

d41d8c, 23 lines

```
d41  struct Bit {
d41    vector<ll> bit;
d41    Bit(int n) : bit(n + 1) {}
d41    void update(int i, ll v) {
d41      for (i++; i < sz(bit); i += i & -i) bit[i] += v;
d41    }
d41    ll query(int i) {
d41      ll ret = 0;
d41      for (i++; i > 0; i -= i & -i) ret += bit[i];
d41      return ret;
d41    }
d41    int lower_bound(ll v){ // min pos st sum[0, pos] >= v
d41      int pos = 0;
d41      for(int j=(1 << 23); j >= 1; j/=2){
d41        if(pos+j < sz(bit) && bit[pos + j] < v){
d41          pos += j;
d41          v -= bit[pos];
d41        }
d41      }
d41      return pos;
d41    }
d41  };
```

Bit2d.h

Description: Points called on the update function NEED to be on the *pts* vector parameter on build.
Time: $\mathcal{O}((\log N)^2)$

"Bit.h" d41d8c, 37 lines

```
d41  struct Bit2d {
d41    vector<vector<int>> ys;
d41    vector<Bit> bit;
d41    vector<int> cmp_x;
d41    Bit2d(){}
d41    void put(int x, int y) {
d41      for (x++; x < sz(ys); x += x & -x) ys[x].push_back(y);
d41    }
d41    int id(const vector<int> &v, int y) {
d41      return (upper_bound(all(v), y) - v.begin()) - 1;
d41    }
d41    void build(vector<pii> pts) {
d41      sort(all(pts));
d41      for(auto p : pts) cmp_x.push_back(p.first);
d41      cmp_x.erase(unique(all(cmp_x)), cmp_x.end());
d41      ys.resize(cmp_x.size() + 1);
d41      for(auto p : pts) put(id(cmp_x, p.first), p.second);
d41      for(auto &v:ys)sort(all(v)), bit.emplace_back(sz(v));
d41    }
d41    void update(int x, int y, int val){
d41      x = id(cmp_x, x);
d41      for(x++; x < sz(ys); x+= x&-x)
d41        bit[x].update(id(ys[x], y), val);
d41    }
d41    int query(int x, int y){
d41      x = id(cmp_x, x);
d41      int ret = 0;
d41      for(x++; x > 0; x-= x&-x)
d41        ret += bit[x].query(id(ys[x], y));
d41      return ret;
d41    }
d41    int query(int x1, int y1, int x2, int y2){
d41      int a = query(x2, y2)-query(x2, y1-1);
d41      return a-query(x1-1, y2)+query(x1-1, y1-1);
d41    }
d41  };
```

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

d41d8c, 32 lines

```
d41  struct Line {
d41   mutable ll k, m, p;
d41   bool operator<(const Line& o) const { return k < o.k; }
d41   bool operator<(ll x) const { return p < x; }
d41  };
d41  struct LineContainer : multiset<Line, less<>> {
      // (for doubles, use inf = 1/.0, div(a,b) = a/b)
d41   static const ll inf = LLONG_MAX;
d41   ll div(ll a, ll b) { // floored division
d41     return a / b - ((a ^ b) < 0 && a % b); }
d41   bool isect(iterator x, iterator y) {
d41     if (y == end()) return x->p = inf, 0;
d41     if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
d41     else x->p = div(y->m - x->m, x->k - y->k);
d41     return x->p >= y->p;
d41   }
d41   void add(ll k, ll m) {
d41     auto z = insert({k, m, 0}), y = z++, x = y;
d41     while (isect(y, z)) z = erase(z);
d41     if (x != begin() && isect(--x, y))
d41       isect(x, y = erase(y));
d41     while ((y = x) != begin() && (--x)->p >= y->p)
d41       isect(x, erase(y));
d41   }
d41   ll query(ll x) {
d41     assert(!empty());
d41     auto l = *lower_bound(x);
d41     return l.k * x + l.m;
d41   }
d41  };
```

Mo.h

Description: For subtree queries, perform an Euler tour and map each node u to the interval $[tin[u], tin[u] + subtree\_size[u] - 1]$. A subtree query becomes a range query over this interval.
For path queries between nodes U and V, Let U be the closest to the root. If V lies in U's subtree, the path corresponds to the interval $[tin[U], tin[V]]$. Otherwise, the path corresponds to the interval $[min(tout[U], tout[V]), max(tin[U], tin[V])]$.
In both cases, nodes on the U–V path appear exactly once in the interval, while all other nodes appear either 0 or 2 times.
Usage: `queries.push(Query(l, r, index of query))`, intervals are `[l, r]`
Time: $\mathcal{O}\left(N\sqrt(Q)\right)$

d41d8c, 44 lines

```
d41  inline int64_t hilOrd(int x, int y, int pow, int rot) {
d41    if (pow == 0) return 0;
d41    int hpow = 1 << (pow - 1);
d41    int seg = (x < hpow) ? ((y < hpow) ? 0 : 3) : ((y < hpow
     ) ? 1 : 2);
d41    seg = (seg + rot) & 3;
d41    const int rotDelta[4] = { 3, 0, 0, 1 };
d41    int nx = x & (x ^ hpow), ny = y & (y ^ hpow);
d41    int nrot = (rot + rotDelta[seg]) & 3;
d41    int64_t sub = int64_t(1) << (2 * pow - 2);
d41    int64_t ans = seg * sub;
d41    int64_t add = hilOrd(nx, ny, pow - 1, nrot);
d41    ans += (seg == 1 || seg == 2) ? add : (sub - add - 1);
d41    return ans;
d41  }

d41  struct Query {
d41    int l, r, idx;
d41    int64_t ord;
d41    Query(int l, int r, int idx) : l(l), r(r), idx(idx) {
d41      ord = hilOrd(l, r, 21, 0);
d41    }
d41    bool operator < (const Query& other) const {
d41      return ord < other.ord;
d41    }
d41  };

d41  vector<Query> queries;
d41  int ans[ms];
```

```
d41   void put(int x) {} // F
d41   void remove(int x) {} // F
d41   int getAns() {}

d41   void Mo() {
d41     int l = 0, r = -1;
d41     sort(queries.begin(), queries.end());
d41     for (Query q : queries) {
d41       while (l > q.l) put(--l);
d41       while (r < q.r) put(++r);
d41       while (l < q.l) remove(l++);
d41       while (r > q.r) remove(r--);
d41       ans[q.idx] = getAns();
d41     }
d41   }
```

MoUpdate.h

Description: Block size should be around $(2 * N * N)^{\frac{1}{3}}$
Usage: `intervals are [l, r], addQuery(l, r, number of updates happened before this query, index of query), addUpdate(index of updated position, value before update, value after update)`
Time: $\mathcal{O}\left(Q * (2 * N * N)^{\frac{1}{3}} * F\right)$ d41d8c, 55 lines

```
d41   const int B = 2700;
d41   struct MoUpdate {
d41     struct Query {
d41       int l, r, t, idx;
d41       Query(int l, int r, int t, int idx)
d41         : l(l), r(r), t(t), idx(idx) {}
d41       bool operator < (const Query& p) const {
d41         if (l / B != p.l / B) return l < p.l;
d41         if (r / B != p.r / B) return r < p.r;
d41         return t < p.t;
d41       }
d41     };
d41     struct Upd {
d41       int i, old, now;
d41       Upd(int i, int old, int now): i(i),old(old),now(now){}
d41     };

d41     vector<Query> queries;
d41     vector<Upd> updates;

d41     void addQuery(int l, int r, int t, int idx) {
d41       queries.push_back(Query(l, r, t, idx)); }
d41     void addUpdate(int i, int old, int now) {
d41       updates.push_back(Upd(i, old, now)); }

d41     void add(int x) {} // F
d41     void rem(int x) {} // F
d41     int getAns() {}
d41     void update(int novo, int idx, int l, int r) {
d41       if (l <= idx && idx <= r) rem(idx);
d41       arr[idx] = novo;
d41       if (l <= idx && idx <= r) add(idx);
d41     }

d41     void solve() {
d41       int l = 0, r = -1, t = 0;
d41       sort(queries.begin(), queries.end());
d41       for (Query q : queries) {
d41         while (l > q.l) add(--l);
d41         while (r < q.r) add(++r);
d41         while (l < q.l) rem(l++);
d41         while (r > q.r) rem(r--);
d41         while (t < q.t) {
d41           auto u = updates[t++];
d41           update(u.now, u.i, l, r);
d41         }
d41         while (t > q.t) {
d41           auto u = updates[--t];
d41           update(u.old, u.i, l, r);
d41         }
d41         ans[q.idx] = getAns();
d41       }
d41     }
d41   };
```

SegmentTree.h

Description: Zero-indexed max-tree. Bounds are inclusive to the left and inclusive to the right. Can be changed by modifying T, f and unit.
Time: $\mathcal{O}(\log N)$ d41d8c, 21 lines

```
d41   struct Tree {
d41    typedef int T;
d41    static constexpr T unit = INT_MIN;
d41    T f(T a, T b) { return max(a, b); } // (any associative
       fn)
d41    vector<T> s; int n;
d41    Tree(int n = 0, T def = unit) : s(2*n, def), n(n) {}
d41    void update(int pos, T val) {
d41       for (s[pos += n] = val; pos /= 2;)
d41         s[pos] = f(s[pos * 2], s[pos * 2 + 1]);
d41    }
d41    T query(int b, int e) { // query [b, e]
d41       e++;
d41      T ra = unit, rb = unit;
d41      for (b += n, e += n; b < e; b /= 2, e /= 2) {
d41        if (b % 2) ra = f(ra, s[b++]);
d41        if (e % 2) rb = f(s[--e], rb);
d41      }
d41      return f(ra, rb);
d41    }
d41   };
```

OrderStatisticTree.h

Description: A set (not multiset!) with support for finding the n'th element, and finding the index of an element. To get a map, change `null_type`.
Time: $\mathcal{O}(\log N)$ d41d8c, 17 lines

```
d41   #include <bits/extc++.h>
d41   using namespace __gnu_pbds;

d41   template<class T>
d41   using Tree = tree<T, null_type, less<T>, rb_tree_tag,
d41       tree_order_statistics_node_update>;

d41   void example() {
d41    Tree<int> t, t2; t.insert(8);
d41    auto it = t.insert(10).first;
d41    assert(it == t.lower_bound(9));
d41    assert(t.order_of_key(10) == 1);
d41    assert(t.order_of_key(11) == 2);
d41    assert(*t.find_by_order(0) == 8);
d41    t.join(t2); // merge t2 into t
d41   }
```

PersistentSegTree.h

Usage: `SegP(size of the segtree, number of updates)`

`roots = {0}, newRoot = update(roots.back(),...), roots.push(newRoot)` d41d8c, 42 lines

```
d41   struct SegP {
d41     static constexpr ll neut = 0;
d41     struct Node {
d41       ll v; // start with neutral value
d41       int l, r;
d41       Node(ll v=neut, int l=0, int r=0) : v(v), l(l), r(r){}
d41     };
d41     vector<Node> seg;
d41     int n, CNT;
d41     SegP(int _n, int upd): seg(20*(upd+_n)), n(_n), CNT(1){}
d41     ll merge(ll a, ll b) { return a + b; }
d41     int update(int root, int pos, int val, int l, int r) {
d41       int p = CNT++;
d41       seg[p] = seg[root];
d41       if (l == r) {
d41         seg[p].v += val;
d41         return p;
d41       }
d41       int mid = (l + r) / 2;
d41       if (pos <= mid){
d41         seg[p].l = update(seg[p].l, pos, val, l, mid);
d41       }else seg[p].r = update(seg[p].r,pos,val,mid+1,r);

d41       seg[p].v=merge(seg[seg[p].l].v, seg[seg[p].r].v);
d41       return p;
d41     }
d41     int query(int p, int L, int R, int l, int r) {
d41       if (l > R || r < L) return neut;
d41       if (L <= l && r <= R) return seg[p].v;
d41       int mid = (l + r) / 2;
d41       int left = query(seg[p].l, L, R, l, mid);
d41       int right = query(seg[p].r, L, R, mid + 1, r);
d41       return merge(left, right);
d41     }
d41     int update(int root, int pos, int val) {
d41       return update(root, pos, val, 0, n - 1);
d41     }
d41     int query(int root, int L, int R) {
d41       return query(root, L, R, 0, n - 1);
d41     }
d41   };
```

SegBeats.h

Description: In Segment Tree Beats, 'lazy' does NOT mean "updates still missing here". The node already reflects all previous updates. Instead, 'lazy' stores what must be propagated to the children before recursing. Always call 'apply(l,r,p)' before descending. This node layout supports range add, range chmin and range chmax operations. Beats conditions:
break: MIN x: mx1 <= x ; MAX x: mi1 >= x
tag: MIN x: x > mx2 ; MAX x: x < mi2
Time: amortized $\mathcal{O}\left(\log^2 N\right)$, without range add $\mathcal{O}(\log N)$ d41d8c, 47 lines

```
d41   struct node{
d41     ll mx1, mx2, sum, lazy;
d41     ll mi1, mi2;
d41     int cMax, cMin, tam;
d41     node(int x=0) : mx1(x),mx2(-inf),mi1(x),mi2(inf),
d41               cMax(1),cMin(1),tam(1),sum(x),lazy(0){}
d41     node(node a, node b){
d41       sum = a.sum+b.sum, tam = a.tam+b.tam;
d41       lazy = 0;
d41       mx1 = max(a.mx1, b.mx1);
d41       mx2 = max(a.mx2, b.mx2);
d41       if(a.mx1 != b.mx1) mx2 = max(mx2, min(a.mx1, b.mx1));
d41       cMax=(a.mx1==mx1 ? a.cMax:0)+(b.mx1==mx1 ? b.cMax:0);

d41       mi1 = min(a.mi1, b.mi1);
d41       mi2 = min(a.mi2, b.mi2);
d41       if(a.mi1 != b.mi1) mi2=min(mi2, max(a.mi1, b.mi1));
d41       cMin=(a.mi1==mi1 ? a.cMin:0)+(b.mi1==mi1 ? b.cMin:0);
d41     }
d41     void apply_sum(ll x){
```

```
d41       mx1 += x, mx2 += x,mi1 += x, mi2 += x;
d41       sum += tam*x, lazy += x;
d41     }
d41     void apply_min(ll x){
d41       if(x >= mx1) return;
d41       sum -= (mx1 - x)*cMax;
d41       if(mi1 == mx1) mi1 = x;
d41       if(mi2 == mx1) mi2 = x;
d41       mx1 = x;
d41     }
d41     void apply_max(ll x){
d41       if(x <= mi1) return;
d41       sum -= (mi1 - x)*cMin;
d41       if(mx1 == mi1) mx1 = x;
d41       if(mx2 == mi1) mx2 = x;
d41       mi1 = x;
d41     }
d41   };
d41   void apply(int l, int r, int p){
d41     for(int i=2*p+1; i<=2*p+2; i++){
d41       seg[i].apply_sum(st[p].lazy);
d41       seg[i].apply_min(st[p].mx1);
d41       seg[i].apply_max(st[p].mi1);
d41     }
d41     seg[p].lazy = 0;
d41   }
```

RMQ.h

Usage: `RMQ rmq(values);`
`rmq.query(inclusive, inclusive);`
Time: $\mathcal{O}(|V|\log|V| + Q)$ d41d8c, 17 lines

```
d41   struct RMQ {
d41     vector<vector<int>> dp;
d41     RMQ(const vector<int>& a) : dp(1, a) {
d41       for (int i = 1, pw = 1; pw*2 <= sz(a); pw*=2, i++) {
d41         dp.emplace_back(sz(a) - pw*2 + 1);
d41         for (int j = 0; j < sz(dp[i]); j++) {
d41           dp[i][j] = min(dp[i-1][j], dp[i-1][j+pw]);
d41         }
d41       }
d41     }
d41     int query(int l, int r) {
d41       assert(l <= r);
d41       int k = 31 - __builtin_clz(r - l + 1);
d41       return min(dp[k][l], dp[k][r - (1 << k) + 1]);
d41     }
d41   };
```

UnionFind.h

Description: Disjoint-set data structure with bipartite check d41d8c, 22 lines

```
d41   struct Uf{
d41       vector<int> tam, ds, bi, c;
d41       Uf(int n) : tam(n, 1), ds(n), bi(n, 1), c(n){
d41           iota(all(ds), 0);
d41       }
d41       int find(int i){ return (i==ds[i] ? i : find(ds[i]));}
d41       int color(int i){
d41           return (i==ds[i] ? 0 : (c[i]^color(ds[i])));}
d41       void merge(int a, int b){
d41           int ca = color(a), cb = color(b);
d41           a = find(a), b = find(b);
d41           if(a == b){
d41               if(ca == cb) bi[a] = false;
d41               return;
d41           }
d41           if(tam[a] < tam[b]) swap(a, b);
d41           ds[b] = a, tam[a] += tam[b];
d41           bi[a] = (bi[a] && bi[b]);
d41           c[b] = (ca ^ cb ^ 1);
d41       }
d41   };
```

UnionFindRollback.h

Description: Disjoint-set data structure with undo. If undo is not needed, skip st, time() and rollback().
Usage: `int t = uf.time(); ...; uf.rollback(t);`
Time: $\mathcal{O}(\log(N))$ d41d8c, 23 lines

```
d41   struct RollbackUF {
d41    vector<int> e;
d41       vector<pii> st;
d41    RollbackUF(int n) : e(n, -1) {}
d41    int size(int x) { return -e[find(x)]; }
d41    int find(int x) { return e[x] < 0 ? x : find(e[x]); }
d41    int time() { return sz(st); }
d41    void rollback(int t) {
d41      for (int i = time(); i --> t;)
d41        e[st[i].first] = st[i].second;
d41      st.resize(t);
d41    }
d41    bool join(int a, int b) {
d41      a = find(a), b = find(b);
d41      if (a == b) return false;
d41      if (e[a] > e[b]) swap(a, b);
d41      st.push_back({a, e[a]});
d41      st.push_back({b, e[b]});
d41      e[a] += e[b]; e[b] = a;
d41      return true;
d41    }
d41   };
```

Numerical (4)

4.1 Polynomials and recurrences

Polynomial.h

d41d8c, 19 lines

```
d41   struct Poly {
d41    vector<double> a;
d41    double operator()(double x) const {
d41      double val = 0;
d41      for (int i = sz(a); i--;) (val *= x) += a[i];
d41      return val;
d41    }
d41    void diff() {
d41      rep(i,1,sz(a)) a[i-1] = i*a[i];
d41      a.pop_back();
d41    }
d41    void divroot(double x0) {
d41      double b = a.back(), c; a.back() = 0;
d41      for(int i=sz(a)-1; i--;)
d41         c = a[i], a[i] = a[i+1]*x0+b, b=c;
d41      a.pop_back();
d41    }
d41   };
```

PolyRoots.h

Description: Finds the real roots to a polynomial.
Usage: `polyRoots({{2,-3,1}},-1e9,1e9) // solve x^2-3x+2 = 0`
Time: $\mathcal{O}\left(n^2\log(1/\epsilon)\right)$
"Polynomial.h" d41d8c, 24 lines

```
d41   vector<double> polyRoots(Poly p, double xmin, double xmax)
       {
d41    if (sz(p.a) == 2) { return {-p.a[0]/p.a[1]}; }
d41    vector<double> ret;
d41    Poly der = p;
d41    der.diff();
d41    auto dr = polyRoots(der, xmin, xmax);
d41    dr.push_back(xmin-1);
d41    dr.push_back(xmax+1);
d41    sort(all(dr));
d41    rep(i,0,sz(dr)-1) {
d41      double l = dr[i], h = dr[i+1];
d41      bool sign = p(l) > 0;
d41      if (sign ^ (p(h) > 0)) {
d41        rep(it,0,60) { // while (h - l > 1e-8)
d41          double m = (l + h) / 2, f = p(m);
d41          if ((f <= 0) ^ sign) l = m;
d41          else h = m;
d41        }
d41        ret.push_back((l + h) / 2);
d41      }
d41    }
d41    return ret;
d41   }
```

PolyInverse.h

d41d8c, 18 lines

```
d41   vector<ll> get_inverse(vector<ll> a) {
d41     if (a.empty()) return {};
d41     int Y = sz(a) - 1, n = 32 - __builtin_clz(Y);
d41     n = (1 << n);
d41     a.resize(n);
d41     vector<ll> inv = { modpow(a[0], mod - 2) }, f, c;
d41     inv.reserve(n);
d41     for (int tam = 2; tam <= n; tam *= 2) {
d41       while (sz(f) < tam) f.push_back(a[sz(f)]);
d41       c = conv(f, inv);
d41       rep(i, 0, tam) c[i] = (c[i] == 0 ? 0 : mod - c[i]);
d41       c[0] += (c[0] + 2 >= mod ? 2 - mod : 2);
d41       inv = conv(inv, c);
d41       inv.resize(tam);
d41     }
d41     return inv;
d41   }
```

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: $\mathcal{O}\left(N^2\right)$ d41d8c, 21 lines

```
d41   vector<ll> berlekampMassey(vector<ll> s) {
d41    int n = sz(s), L = 0, m = 0;
d41    vector<ll> C(n), B(n), T;
d41    C[0] = B[0] = 1;

d41    ll b = 1;
d41    rep(i,0,n) { ++m;
d41      ll d = s[i] % mod;
d41      rep(j,1,L+1) d = (d + C[j] * s[i - j]) % mod;
d41      if (!d) continue;
d41      T = C; ll coef = d * modpow(b, mod-2) % mod;
d41      rep(j,m,n) C[j] = (C[j] - coef * B[j - m]) % mod;
d41      if (2 * L > i) continue;
d41      L = i + 1 - L; B = T; b = d; m = 0;
d41    }

d41    C.resize(L + 1); C.erase(C.begin());
d41    for (ll& x : C) x = (mod - x) % mod;
d41    return C;
d41   }
```

LinearRecurrence.h

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

d41d8c, 27 lines

```
d41 using Poly = vector<ll>;
d41 ll linearRec(Poly S, Poly tr, ll k) {
d41   int n = sz(tr);

d41   auto combine = [&](Poly a, Poly b) {
d41     Poly res(n * 2 + 1);
d41     rep(i,0,n+1) rep(j,0,n+1)
d41       res[i + j] = (res[i + j] + a[i] * b[j]) % mod;
d41     for (int i = 2 * n; i > n; --i) rep(j,0,n)
d41       res[i-1-j] = (res[i-1-j] + res[i] * tr[j]) % mod;
d41     res.resize(n + 1);
d41     return res;
d41   };

d41   Poly pol(n + 1), e(pol);
d41   pol[0] = e[1] = 1;

d41   for (++k; k; k /= 2) {
d41     if (k % 2) pol = combine(pol, e);
d41     e = combine(e, e);
d41   }

d41   ll res = 0;
d41   rep(i,0,n) res = (res + pol[i + 1] * S[i]) % mod;
d41   return res;
d41 }
```

4.2 Matrices

SolveLinear.h

Description: If inv = 1, finds the inverse of the matrix eq and returns it as a flat vector
Time: $\mathcal{O}(\min(n,m)\, nm)$

d41d8c, 52 lines

```
d41 struct Gauss {
d41   const double eps = 1e-9;
d41   vector<vector<double>> eq;
d41   void addEquation(const vector<double>& e) {
d41     eq.push_back(e); }
d41   pair<int, vector<double>> solve(int inv=0) {
d41     int n = sz(eq), m = sz(eq[0]) - 1 + inv;
d41     if(inv){
d41       rep(i, 0, n) eq[i].resize(2*n), eq[i][n+i] = 1;
d41     }
d41     vector<int> where(m, -1);
d41     for (int col = 0, row = 0; col < m && row < n; col++)
    {
d41       int sel = row;
d41       rep(i, row, n) {
d41         if (abs(eq[i][col]) > abs(eq[sel][col])) sel = i;
d41       }
d41       if (abs(eq[sel][col]) < eps) continue;
d41       rep(i, col, sz(eq[0])) swap(eq[sel][i], eq[row][i]);
d41       where[col] = row;
d41       rep(i, 0, n) if (i != row) {
d41         double c = eq[i][col] / eq[row][col];
d41         rep(j, col, sz(eq[0])) eq[i][j] -= eq[row][j] * c;
d41       }
d41       ++row;
d41     }
d41     if(inv){
d41       vector<double> res;
d41       rep(i, 0, n) {
d41         if (where[i] == -1) return {0, {}}; // Singular
d41         rep(j, n, 2*n)
d41           res.push_back(eq[where[i]][j] / eq[where[i]][i])
    ;
d41       }
d41       return {1, res};
d41     }

d41     vector<double> ans(m, 0);
d41     rep(i, 0, m) {
d41       if (where[i] != -1)
d41         ans[i] = eq[where[i]][m] / eq[where[i]][i];
d41     }
d41     rep(i, 0, n) {
d41       double sum = 0;
d41       rep(j, 0, m) {
d41         sum = sum + ans[j] * eq[i][j];
d41       }
d41       if(abs(sum - eq[i][m]) > eps)return {0, {}};
d41     }
d41     rep(i, 0, m) if (where[i] == -1) return {2, ans};
d41     return {1, ans};
d41   }
d41 };
```

SolveLinearBinary.h

Time: $\mathcal{O}\left(\frac{\min(n,m)\, nm}{64}\right)$

d41d8c, 32 lines

```
d41 pair<int, bitset<M>> gauss(vector<bitset<M>> eq) {
d41     int n = eq.size(), m = M - 1;
d41     vector<int> where(m, -1);
d41     for(int col = 0, row = 0; col < m && row < n; col++){
d41         rep(i,row,n)
d41             if (eq[i][col]) {
d41                 swap(eq[i], eq[row]);
d41                 break;
d41             }
d41         if (!eq[row][col]) continue;
d41         where[col] = row;

d41         rep(i, 0, n) {
d41             if (i != row && eq[i][col]) eq[i] ^= eq[row];
d41         }
d41         ++row;
d41     }
d41     bitset<M> ans;
d41     rep(i,0,m) {
d41         if (where[i] != -1) ans[i] = eq[where[i]][m];
d41     }
d41     rep(i,0,n) {
d41         int sum = (ans & eq[i]).count();
d41         sum %= 2;
d41         if (sum != eq[i][m]) return pair(0, bitset<M>());
d41     }
d41     rep(i,0,m) {
d41         if (where[i] == -1) return pair(INF, ans);
d41     }
d41     return pair(1, ans);
d41 }
```

XorGauss.h

d41d8c, 30 lines

```
d41 struct XorGauss {
d41     int N;
d41     vector<ll> basis, who, mask;
d41     XorGauss(int N) : N(N), basis(N), who(N), mask(N) {}
        // if(ans & (1ll << j)) who[j] was used to form x
d41     bool belong(ll x){
d41         ll ans = 0;
d41         for(int i=N-1; i>=0; i--){
d41             if((x ^ basis[i]) < x){
d41                 ans ^= mask[i];
d41                 x ^= basis[i];
d41             }
d41         }
d41         return (x == 0);
d41     }
d41     void add(ll v, int idx) {
d41         ll msk = 0;
d41         for (int i = N - 1; i >= 0; i--) {
d41             if (!(v & (1ll << i))) continue;
d41             if (basis[i] == 0) {
d41                 basis[i] = v, who[i] = idx;
d41                 mask[i] = (msk | (1ll << i));
d41                 return;
d41             }
d41             msk ^= mask[i];
d41             v ^= basis[i];
d41         }
d41     }
d41 };
```

4.3 Fourier transforms

FastFourierTransform.h

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

d41d8c, 44 lines

```
d41 typedef complex<double> C;

d41 void fft(vector<C>& a) {
d41   int n = a.size(), L = 31 - __builtin_clz(n);
d41   static vector<complex<long double>> R(2, 1); // 10%
    faster if double
d41   static vector<C> rt(2, 1);
d41   for (static int k = 2; k < n; k *= 2) {
d41     R.resize(n);
d41     rt.resize(n);
d41     auto x = polar(1.0L, acos(-1.0L) / k);
d41     rep(i,k,2*k) rt[i] = R[i] = i&1 ? R[i/2] * x : R[i/2];
d41   }
d41   vector<ll> rev(n);
d41   rep(i,0,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
d41   rep(i,0,n) if (i < rev[i]) swap(a[i], a[rev[i]]);

d41   for (int k = 1; k < n; k *= 2) {
d41     for (int i = 0; i < n; i += 2 * k) {
d41       for (int j = 0; j < k; j++) {
d41         auto x = (double*)&rt[j + k];
d41         auto y = (double*)&a[i + j + k];
d41         C z(x[0]*y[0] - x[1]*y[1], x[0]*y[1] + x[1]*y[0]);
d41         a[i + j + k] = a[i + j] - z;
d41         a[i + j] += z;
d41       }
d41     }
d41   }
d41 }

d41 vector<ll> conv(const vector<ll>& a, const vector<ll>& b){
d41   if (a.empty() || b.empty()) return {};
d41   vector<ll> res(sz(a) + sz(b) - 1);
d41   int L = 32 - __builtin_clz(sz(res)), n = 1 << L;
d41   vector<C> in(n), out(n);
d41   copy(all(a), in.begin());
```

```
d41    rep(i,0,sz(b)) in[i].imag(b[i]);
d41    fft(in);
d41    for (C& x : in) x *= x;
d41    rep(i,0,n) out[i] = in[-i & (n - 1)] - conj(in[i]);
d41    fft(out);
d41    rep(i,0,sz(res)) res[i]=round(imag(out[i]) / (4 * n));
d41    return res;
d41  }
```

FastFourierTransformMod.h

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

"FastFourierTransform.h" d41d8c, 23 lines

```
d41  typedef vector<ll> vl;
d41  template<int M> vl convMod(const vl &a, const vl &b) {
d41   if (a.empty() || b.empty()) return {};
d41   vl res(sz(a) + sz(b) - 1);
d41   int B=32-__builtin_clz(sz(res)), n=1<<B,cut=int(sqrt(M));
d41   vector<C> L(n), R(n), outs(n), outl(n);
d41   rep(i,0,sz(a)) L[i] =C((int)a[i] / cut, (int)a[i] % cut);
d41   rep(i,0,sz(b)) R[i] =C((int)b[i] / cut, (int)b[i] % cut);
d41   fft(L), fft(R);
d41   rep(i,0,n) {
d41     int j = -i & (n - 1);
d41     outl[j] = (L[i] + conj(L[j])) * R[i] / (2.0 * n);
d41     outs[j] = (L[i] - conj(L[j])) * R[i] / (2.0 * n) / 1i;
d41   }
d41   fft(outl), fft(outs);
d41   rep(i,0,sz(res)) {
d41     ll av = ll(real(outl[i])+.5), cv =ll(imag(outs[i])+.5);
d41     ll bv = ll(imag(outl[i])+.5) + ll(real(outs[i])+.5);
d41     res[i] = ((av % M * cut + bv) % M * cut + cv) % M;
d41   }
d41   return res;
d41  }
```

NumberTheoreticTransform.h

Description: ntt(a) computes $\hat{f}(k) = \sum_x a[x] g^{xk}$ for all k, where $g = \text{root}^{(mod-1)/N}$. N must be a power of 2. Useful for convolution modulo specific nice primes of the form $2^a b + 1$, where the convolution result has size at most 2^a. For arbitrary modulo, see FFTMod. `conv(a, b) = c`, where $c[x] = \sum a[i]b[x-i]$. For manual convolution: NTT the inputs, multiply pointwise, divide by n, reverse(start+1, end), NTT back. Inputs must be in [0, mod).
Time: $\mathcal{O}(N \log N)$

d41d8c, 34 lines

```
d41  const int mod = 998244353, root = 62;
d41  typedef vector<ll> vl;
d41  void ntt(vl &a) {
d41   int n = sz(a), L = 31 - __builtin_clz(n);
d41   static vl rt(2, 1);
d41   for (static int k = 2, s = 2; k < n; k *= 2, s++) {
d41     rt.resize(n);
d41     ll z[] = {1, modpow(root, mod >> s)};
d41     rep(i,k,2*k) rt[i] = rt[i / 2] * z[i & 1] % mod;
d41   }
d41   vector<int> rev(n);
d41   rep(i,0,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
d41   rep(i,0,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
d41   for (int k = 1; k < n; k *= 2)
d41     for (int i = 0; i < n; i += 2 * k) rep(j,0,k) {
d41       ll z = rt[j+k] * a[i+j+k] % mod, &ai = a[i+j];
d41       a[i + j + k] = ai - z + (z > ai ? mod : 0);
d41       ai += (ai + z >= mod ? z - mod : z);
d41     }
d41  }
d41  vl conv(const vl &a, const vl &b) {
d41   if (a.empty() || b.empty()) return {};
d41   int s = sz(a) + sz(b) - 1, B = 32 - __builtin_clz(s),
d41       n = 1 << B;
d41   int inv = modpow(n, mod - 2);
d41   vl L(a), R(b), out(n);
d41   L.resize(n), R.resize(n);
d41   ntt(L), ntt(R);
d41   rep(i,0,n)
d41     out[-i & (n - 1)] = (ll)L[i] * R[i] % mod * inv % mod;
d41   ntt(out);
d41   return {out.begin(), out.begin() + s};
d41  }
```

FWHT.h

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

d41d8c, 20 lines

```
d41  void FST(vector<ll>& a, bool inv) {
d41    for (int n = sz(a), step = 1; step < n; step *= 2) {
d41      for (int i = 0; i < n; i += 2 * step) {
d41        for (int j = i; j < i + step; j++) {
d41          ll& u = a[j], &v = a[j + step];
d41          tie(u, v) =
d41          inv ? pair(v - u, u) : pair(v, u + v); // AND
d41          inv ? pair(v, u - v) : pair(u + v, u); // OR
d41          pair(u + v, u - v);  // XOR
d41        }
d41      }
d41    }
d41    if(inv) for(ll& x : a) x /= sz(a); // XOR only
d41  }
d41  vector<ll> conv(vector<ll> a, vector<ll> b) {
d41    FST(a, 0); FST(b, 0);
d41    for (int i = 0; i < sz(a); i++) a[i]*=b[i];
d41    FST(a, 1); return a;
d41  }
```

Number theory (5)

5.1 Modular arithmetic

ModInverse.h

Description: Pre-computation of modular inverses. Assumes LIM $\leq$ mod and that mod is a prime.

d41d8c, 5 lines

```
d41  const ll mod = 1000000007, LIM = 200000;
d41  inv[1] = 1;
d41  for(int i=2; i<LIM; i++)
d41      inv[i] = mod - (mod / i) * inv[mod % i] % mod;
```

ModMulLL.h

Description: Calculate $a \cdot b \bmod c$ (or $a^b \bmod c$) 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`

d41d8c, 12 lines

```
d41  typedef unsigned long long ull;
d41  ull modmul(ull a, ull b, ull M) {
d41   ll ret = a * b - M * ull(1.L / M * a * b);
d41   return ret + M * (ret < 0) - M * (ret >= (ll)M);
d41  }
d41  ull modpow(ull b, ull e, ull mod) {
d41   ull ans = 1;
d41   for (; e; b = modmul(b, b, mod), e /= 2)
d41     if (e & 1) ans = modmul(ans, b, mod);
d41   return ans;
d41  }
```

ModPow.h

d41d8c, 9 lines

```
d41  const ll mod = 1000000007; // faster if const

d41  ll modpow(ll b, ll e) {
d41   ll ans = 1;
d41   for (; e; b = b * b % mod, e /= 2)
d41     if (e & 1) ans = ans * b % mod;
d41   return ans;
d41  }
```

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: $\mathcal{O}(\log^2 p)$ worst case, $\mathcal{O}(\log p)$ for most p

"ModPow.h" d41d8c, 25 lines

```
d41  ll sqrt(ll a, ll p) {
d41   a %= p; if (a < 0) a += p;
d41   if (a == 0) return 0;
d41   assert(modpow(a, (p-1)/2, p) == 1); // else no solution
d41   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
d41   ll s = p - 1, n = 2;
d41   int r = 0, m;
d41   while (s % 2 == 0)
d41     ++r, s /= 2;
d41   while (modpow(n, (p - 1) / 2, p) != p - 1) ++n;
d41   ll x = modpow(a, (s + 1) / 2, p);
d41   ll b = modpow(a, s, p), g = modpow(n, s, p);
d41   for (;; r = m) {
d41     ll t = b;
d41     for (m = 0; m < r && t != 1; ++m)
d41       t = t * t % p;
d41     if (m == 0) return x;
d41     ll gs = modpow(g, 1LL << (r - m - 1), p);
d41     g = gs * gs % p;
d41     x = x * gs % p;
d41     b = b * g % p;
d41   }
d41  }
```

DiscreteLog.h

Description: Returns the smallest x such that $a^x \bmod m = b \bmod m$. If no such x exists, returns -1.
Time: O(sqrt(m) * log(sqrt(m)))

d41d8c, 32 lines

```
d41  int solve(int a, int b, int m) {
d41      a %= m, b %= m;
d41      if (a == 0) return (b ? -1 : 1);
         // caso gcd(a, m) > 1
d41      int k = 1, add = 0, g;
d41      while ((g = gcd(a, m)) > 1) {
d41          if (b == k) return add;
d41          if (b % g) return -1;
d41          b /= g, m /= g, ++add;
d41          k = (k * 1ll * a / g) % m;
d41      }

d41      int sq = sqrt(m) + 1;
d41      int big = 1;
d41      for (int i = 0; i < sq; i++) big = (1ll * big * a) % m
    ;

d41      vector<pii> vals;
d41      for (int q = 0, cur = b; q <= sq; q++) {
d41          vals.push_back({ cur, q });
d41          cur = (1ll * cur * a) % m;
d41      }
d41      sort(all(vals));
```

```cpp
    for (int p = 1, cur = k; p <= sq; p++) {
        cur = (1ll * cur * big) % m;
        auto it = lower_bound(all(vals), pair(cur, INF));
        if (it != vals.begin() && (--it)->first == cur) {
            return sq * p - it->second + add;
        }
    }
    return -1;
}
```

DiscreteRoot.h

Description: Returns x such that $x^k \bmod m = a \bmod m$. If no such x exists, returns -1.

Time: O(sqrt(m) * log(sqrt(m)))

"PrimitiveRoot.h", "DiscreteLog.h" d41d8c, 11 lines

```cpp
    // Discrete Root

ll discreteRoot(ll k, ll a, ll m) {
    ll g = primitiveRoot(m);
    ll y = discreteLog(fexp(g, k, m), a, m);
    if (y == -1) return y;
    return fexp(g, y, m);
}
```

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" d41d8c, 13 lines

```cpp
bool isPrime(ull n) {
 if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
 ull A[] = {2, 325, 9375, 28178, 450775, 9780504, 17952650
22};
 ull s = __builtin_ctzll(n-1), d = n >> s;
 for (ull 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" d41d8c, 19 lines

```cpp
ull pollard(ull n) {
 ull x = 0, y = 0, t = 30, prd = 2, i = 1, q;
 auto f = [&](ull x) { return modmul(x, x, n) + i; };
 while (t++ % 40 || gcd(prd, n) == 1) {
   if (x == y) x = ++i, y = f(x);
   if ((q = modmul(prd, max(x,y) - min(x,y), n))) prd = q;
   x = f(x), y = f(f(y));
 }
 return gcd(prd, n);
}
vector<ull> factor(ull n) {
 if (n == 1) return {};
 if (isPrime(n)) return {n};
 ull x = pollard(n);
 auto l = factor(x), r = factor(n / x);
 l.insert(l.end(), all(r));
 return l;
}
```

PrimitiveRoot.h

d41d8c, 15 lines

```cpp
    //is n primitive root of p ?
bool test(ll x, ll p) {
    ll m = p - 1;
    for (ll i = 2; i * i <= m; ++i) if (!(m % i)) {
        if (modpow(x, i, p) == 1) return false;
        if (modpow(x, m / i, p) == 1) return false;
    }
    return true;
}
    //find the smallest primitive root for p
ll search(ll p) {
    for (ll i = 2; i < p; i++) if (test(i, p)) return i;
    return -1;
}
```

5.3 Divisibility

Euclid.h

Description: Find x, y such that $Ax + By = \gcd(A, B)$. If $\gcd(A, B) = 1$, then $x = A^{-1} \pmod B$ and $y = B^{-1} \pmod A$.

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

d41d8c, 6 lines

```cpp
ll euclid(ll a, ll b, ll &x, ll &y) {
 if (!b) return x = 1, y = 0, a;
 ll d = euclid(b, a % b, y, x);
 return y -= a/b * x, d;
}
```

CRT.h

d41d8c, 25 lines

```cpp
ll modinverse(ll a, ll b, ll s0 = 1, ll s1 = 0) {
  return !b ? s0 : modinverse(b, a % b, s1, s0 - s1 * (a /
  b)); }

ll mul(ll a, ll b, ll m) {
  return (((__int128_t)a*b)%m + m)%m;
}

struct Equation {
  ll mod, ans;
  bool valid;
  Equation(ll a, ll m) { mod = m, ans = a, valid = true; }
  Equation() { valid = false; }
  Equation(Equation a, Equation b) {
    valid = false;
    if (!a.valid || !b.valid) return;
    ll g = gcd(a.mod, b.mod);
    if ((a.ans - b.ans) % g != 0) return;
    valid = true;
    mod = a.mod * (b.mod / g);
    ll x = mul(a.mod, modinverse(a.mod, b.mod), mod);
    ans = a.ans + mul(x, (b.ans - a.ans) / g, mod);
    ans = (ans % mod + mod) % mod;
  }
};
```

DivisionTrick.h

d41d8c, 15 lines

```cpp
void floor_ranges(int n) {
    for (int l = 1, r; l <= n; l = r + 1) {
        r = n / (n / l);
        // floor(n/y) has the same value for y in [l..r]
    }
}
void ceil_ranges(int n) {
    for (int l = 1, r; l <= n; l = r + 1) {
        int x = (n + l - 1) / l;
        if (x == 1) r = n;
        else r = (n - 1) / (x - 1);
        // ceil(n/y) has the same value for y in [l..r]
    }
}
```

Phi.h

Description: *Euler's* ϕ function is defined as $\phi(n) := \#$ of positive integers $\le n$ that are coprime with n. $\phi(1) = 1$, p prime $\Rightarrow \phi(p^k) = (p-1)p^{k-1}$, m, n coprime $\Rightarrow \phi(mn) = \phi(m)\phi(n)$. If $n = p_1^{k_1} p_2^{k_2} \ldots p_r^{k_r}$ then $\phi(n) = (p_1 - 1)p_1^{k_1-1} \ldots (p_r - 1)p_r^{k_r-1}$. $\phi(n) = n \cdot \prod_{p|n}(1 - 1/p)$.
$\sum_{d|n} \phi(d) = n$, $\sum_{1 \le k \le n, \gcd(k,n)=1} k = n\phi(n)/2, n > 1$

Euler's thm: a, n coprime $\Rightarrow a^{\phi(n)} \equiv 1 \pmod n$.

Euler's thm (generalized): a, m arbitrary, $n \ge \log_2 m \Rightarrow a^n \equiv a^{\phi(m)+(n \bmod \phi(m))} \pmod m$.

d41d8c, 6 lines

```cpp
void calculatePhi() {
 for(int i=0; i<LIM; i++) phi[i] = i&1 ? i : i/2;
 for (int i = 3; i < LIM; i += 2) if(phi[i] == i)
   for (int j = i; j < LIM; j += i) phi[j] -= phi[j] / i;
}
```

Combinatorial (6)

PartitionSolver.h

d41d8c, 61 lines

```cpp
template<const int N>
struct PartitionSolver {
  vector<vector<int>> part, to, from;
  PartitionSolver() {
    vector<int> a;
    part.push_back(a);
    gen(1, N, a);
    sort(all(part));
    to.assign(sz(part), vector<int>(N + 1, -1));
    from = to;
    for (int i = 0; i < sz(part); i++) {
      int sum = 0;
      auto arr = part[i];
      for (auto x : arr) sum += x;
      to[i][0] = i;
      from[i][0] = i;
      for (int j = 1; j + sum <= N; j++) {
        arr = part[i];
        arr.push_back(j);
        sort(all(arr));
        to[i][j] = getIndex(arr);
        from[to[i][j]][j] = i;
      }
    }
  }

  int size() const { return sz(part); }
  int getIndex(const vector<int>& arr) const {
    return lower_bound(all(part), arr) - part.begin(); }
  int add(int id, int num) const { return to[id][num]; }
  int rem(int id, int num) const { return from[id][num]; }
  vector<int> getPartition(int id) const {
    return part[id]; }

  void gen(int i, int sum, vector<int>& a) {
    if (i > sum) { return; }
    a.push_back(i);
    part.push_back(a);
    gen(i, sum - i, a);
    a.pop_back();
```

```
d41       gen(i + 1, sum, a);
d41     }
d41   };

      // Number of partitions for all integers <= n
d41   vector<ll> partitionNumber(int n) {
d41     vector<ll> ans(n + 1, 0);
d41     ans[0] = 1;
d41     for (int i = 1; i <= n; i++) {
d41       for (int j = 1; j * (3 * j + 1) / 2 <= i; j++) {
d41         ll here = ans[i - j * (3 * j + 1) / 2];
d41         ans[i] = (ans[i] + (j & 1 ? here : -here));
d41       }
d41       for (int j = 1; j * (3 * j - 1) / 2 <= i; j++) {
d41         ll here = ans[i - j * (3 * j - 1) / 2];
d41         ans[i] = (ans[i] + (j & 1 ? here : -here));
d41       }
d41     }
d41     return ans;
d41   }
```

Graph (7)

7.1 Fundamentals

BellmanFord.h

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

d41d8c, 24 lines

```
d41   const ll inf = LLONG_MAX;
d41   struct Ed { int a, b, w, s() { return a < b ? a : -a; }};
d41   struct Node { ll dist = inf; int prev = -1; };

d41   void bell(vector<Node>& nodes, vector<Ed>& eds, int s) {
d41    nodes[s].dist = 0;
d41    sort(all(eds), [](Ed a, Ed b) { return a.s() < b.s(); });

d41    int lim = sz(nodes) / 2 + 2; // /3+100 with shuffled
       vertices
d41    rep(i,0,lim) for (Ed ed : eds) {
d41      Node cur = nodes[ed.a], &dest = nodes[ed.b];
d41      if (abs(cur.dist) == inf) continue;
d41      ll d = cur.dist + ed.w;
d41      if (d < dest.dist) {
d41        dest.prev = ed.a;
d41        dest.dist = (i < lim-1 ? d : -inf);
d41      }
d41    }
d41    rep(i,0,lim) for (Ed e : eds) {
d41      if (nodes[e.a].dist == -inf)
d41        nodes[e.b].dist = -inf;
d41    }
d41   }
```

FloydWarshall.h

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

d41d8c, 13 lines

```
d41   const ll inf = 1LL << 62;
d41   void floydWarshall(vector<vector<ll>>& m) {
d41    int n = sz(m);
d41    rep(i,0,n) m[i][i] = min(m[i][i], 0LL);
d41    rep(k,0,n) rep(i,0,n) rep(j,0,n)
d41      if (m[i][k] != inf && m[k][j] != inf) {
d41        auto newDist = max(m[i][k] + m[k][j], -inf);
d41        m[i][j] = min(m[i][j], newDist);
d41      }
d41    rep(k,0,n) if (m[k][k] < 0) rep(i,0,n) rep(j,0,n)
d41      if (m[i][k] != inf && m[k][j] != inf) m[i][j] = -inf;
d41   }
```

7.2 Network flow and Matching

Dinic.h

Time: - $\mathcal{O}(\min(m \cdot \text{max\_flow},\ n^2 m))$.
- For graphs with unit capacities: $\mathcal{O}\left(\min(m\sqrt{m},\ mn^{2/3})\right)$.
- If every vertex has in-degree 1 or out-degree 1: $\mathcal{O}(m\sqrt{n})$.
- With capacity scaling: $\mathcal{O}(nm\log(\text{MAXCAP}))$ with high constant factor.

d41d8c, 56 lines

```
d41   struct Dinic {
d41    const bool scaling = false;
d41    int lim;
d41    struct edge {
d41      int to, rev;
d41      ll cap, flow;
d41      bool res;
d41      edge(int to_, ll cap_, int rev_, bool res_)
d41        : to(to_), cap(cap_), rev(rev_), flow(0), res(res_){}
d41    };

d41    vector<vector<edge>> g;
d41    vector<int> lev, beg;
d41    ll F;
d41    Dinic(int n) : g(n), lev(n), beg(n), F(0) {}

d41    void add(int a, int b, ll c, ll other = 0) {
d41      g[a].emplace_back(b, c, sz(g[b]), false);
d41      g[b].emplace_back(a, other, sz(g[a])-1, true);
d41    }
d41    bool bfs(int s, int t) {
d41      fill(all(lev), -1);
d41      fill(all(beg), 0);
d41           lev[s] = 0;
d41      queue<int> q; q.push(s);
d41      while (sz(q)) {
d41        int u = q.front(); q.pop();
d41        for (auto& i : g[u]) {
d41          if (lev[i.to] != -1 or (i.flow == i.cap)) continue;
d41          if (scaling and i.cap - i.flow < lim) continue;
d41          lev[i.to] = lev[u] + 1;
d41          q.push(i.to);
d41        }
d41      }
d41      return lev[t] != -1;
d41    }
d41    ll dfs(int v, int s, ll f = INF) {
d41      if (!f or v == s) return f;
d41      for (int& i = beg[v]; i < sz(g[v]); i++) {
d41        auto& e = g[v][i];
d41        if (lev[e.to] != lev[v] + 1) continue;
d41        ll foi = dfs(e.to, s, min(f, e.cap - e.flow));
d41        if (!foi) continue;
d41        e.flow += foi, g[e.to][e.rev].flow -= foi;
d41        return foi;
d41      }
d41      return 0;
d41    }
d41    ll maxFlow(int s, int t) {
d41      for (lim = scaling ? (1<<30) : 1; lim; lim /= 2)
d41        while (bfs(s, t)) while (ll ff = dfs(s, t)) F += ff;
d41      return F;
d41    }
d41    bool inCut(int u){ return lev[u] != -1; }
d41   };
```

LowerBoundFlow.h

Description: Calculates maximum flow with lower/upper bounds on edges. Returns -1 if no feasible flow exists. `add(a, b, l, r)` adds edge $a \to b$ where flow f must satisfy $l \le f \le r$. `add(a, b, c)` adds edge $a \to b$ with capacity c (implies $0 \le f \le c$). Same complexity as Dinic.

"Dinic.h" d41d8c, 36 lines

```
d41   struct lb_max_flow : Dinic {
d41    vector<ll> d;
d41    lb_max_flow(int n) : Dinic(n + 2), d(n, 0) {}
d41    void add(int a, int b, int l, int r) {
d41      d[a] -= l;
d41      d[b] += l;
d41      Dinic::add(a, b, r - l);
d41    }
d41    void add(int a, int b, int c) {
d41      Dinic::add(a, b, c);
d41    }
d41    bool has_circulation() {
d41      int n = sz(d);
d41      ll cost = 0;
d41      rep(i, 0, n){
d41        if (d[i] > 0) {
d41          cost += d[i];
d41          Dinic::add(n, i, d[i]);
d41        } else if (d[i] < 0) {
d41          Dinic::add(i, n+1, -d[i]);
d41        }
d41      }

d41      return (Dinic::maxFlow(n, n+1) == cost);
d41    }
d41    bool has_flow(int src, int snk) {
d41      Dinic::add(snk, src, INF);
d41      return has_circulation();
d41    }
d41    ll max_flow(int src, int snk) {
d41      if (!has_flow(src, snk)) return -1;
d41      Dinic::F = 0;
d41      return Dinic::maxFlow(src, snk);
d41    }
d41   };
```

MinCost.h

Description: Min-cost max-flow. If costs can be negative, call setpi before maxflow, but note that negative cost cycles are not supported. To obtain the actual flow, look at positive values only. If graph is a DAG pi can be calculated with DP instead of Bellman ford.
Time: $\mathcal{O}(FE\log(V))$ where F is max flow. $\mathcal{O}(VE)$ for setpi.

d41d8c, 95 lines

```
d41   #include <bits/extc++.h>

d41   const ll INF = numeric_limits<ll>::max() / 4;

d41   struct MCMF {
d41     struct edge {
d41       int from, to, rev;
d41       ll cap, cost, flow;
d41     };
d41     int N;
d41     vector<vector<edge>> ed;
d41     vector<int> seen, vis;
d41     vector<ll> dist, pi;
d41     vector<edge*> par;

d41     MCMF(int N) : N(N), ed(N), seen(N), vis(N),
```

```cpp
    dist(N), pi(N), par(N) {}

  void addEdge(int from, int to, ll cap, ll cost) {
    if (from == to || cap == 0) return;
    ed[from].push_back(edge{from,to,sz(ed[to]),cap,cost,0
  });
    ed[to].push_back(edge{to,from,sz(ed[from])-1,0,-cost,0
  });
  }

  void path(int s) {
    fill(all(seen), 0);
    fill(all(dist), INF);
    dist[s] = 0;
    ll di;
    __gnu_pbds::priority_queue<pair<ll, int>> q;
    vector<decltype(q)::point_iterator> its(N);
    q.push({ 0, s });

    while (!q.empty()) {
      s = q.top().second; q.pop();
      seen[s] = 1; di = dist[s] + pi[s];
      for (edge& e : ed[s]) {
        if (!seen[e.to]) {
          ll val = di - pi[e.to] + e.cost;
          if(e.cap - e.flow > 0 && val < dist[e.to]){
            dist[e.to] = val;
            par[e.to] = &e;
            if (its[e.to] == q.end()) {
              its[e.to] = q.push({-dist[e.to], e.to});
            }
            else q.modify(its[e.to], {-dist[e.to], e.to});
          }
        }
      }
    }
    for (int i = 0; i < N; i++) {
      pi[i] = min(pi[i] + dist[i], INF);
    }
  }

  pair<ll, ll> maxflow(int s, int t) {
    setpi(s, t);
    ll totflow = 0, totcost = 0;
    while (path(s), seen[t]) {
      ll fl = INF;
      for (edge* x = par[t]; x; x = par[x->from]) {
        fl = min(fl, x->cap - x->flow);
      }
      totflow += fl;
      for (edge* x = par[t]; x; x = par[x->from]) {
        x->flow += fl;
        ed[x->to][x->rev].flow -= fl;
      }
    }
    for (int i = 0; i < N; i++) {
      for (edge& e : ed[i]) {
        totcost += e.cost * e.flow;
      }
    }
    return { totflow, totcost / 2 };
  }

  // If some costs can be negative, call this before
      maxflow:
  void setpi(int s, int t) {
    fill(all(pi), INF);
    pi[s] = 0;
    int it = N, ch = 1;
    ll v;
    while (ch-- && it--) {
      for (int i = 0; i < N; i++) {
        if (pi[i] != INF)
          for (edge& e : ed[i]) if (e.cap)
            if((v= pi[i] + e.cost)< pi[e.to])
              pi[e.to] = v, ch = 1;
      }
    }
    assert(it >= 0); // negative cost cycle
  }
};
```

PushRelabel.h

Description: Push-relabel using the highest label selection rule and the gap heuristic. Quite fast in practice. To obtain the actual flow, look at positive values only.

Time: $\mathcal{O}\left(V^2\sqrt{E}\right)$

d41d8c, 55 lines

```cpp
struct PushRelabel {
 struct Edge {
   int dest, back;
   ll f, c;
 };
 vector<vector<Edge>> g;
 vector<ll> ec;
 vector<Edge*> cur;
 vector<vector<int>> hs;
 vector<int> H;
 PushRelabel(int n) : g(n), ec(n), cur(n), hs(2*n), H(n){}

 void addEdge(int s, int t, ll cap, ll rcap=0) {
   if (s == t) return;
   g[s].push_back({t, sz(g[t]), 0, cap});
   g[t].push_back({s, sz(g[s])-1, 0, rcap});
 }

 void addFlow(Edge& e, ll f) {
   Edge &back = g[e.dest][e.back];
   if (!ec[e.dest] && f) hs[H[e.dest]].push_back(e.dest);
   e.f += f; e.c -= f; ec[e.dest] += f;
   back.f -= f; back.c += f; ec[back.dest] -= f;
 }
 ll calc(int s, int t) {
   int v = sz(g); H[s] = v; ec[t] = 1;
   vector<int> co(2*v); co[0] = v-1;
   for(int i=0; i<v; i++) cur[i] = g[i].data();
   for (Edge& e : g[s]) addFlow(e, e.c);

   for (int hi = 0;;) {
     while (hs[hi].empty()) if (!hi--) return -ec[s];
     int u = hs[hi].back(); hs[hi].pop_back();
     while (ec[u] > 0)  // discharge u
       if (cur[u] == g[u].data() + sz(g[u])) {
         H[u] = 1e9;
         for (Edge& e : g[u]){
           if (e.c && H[u] > H[e.dest]+1)
             H[u] = H[e.dest]+1, cur[u] = &e;
         }
         if (++co[H[u]], !--co[hi] && hi < v){
           for(int i=0; i<v; i++){
             if (hi < H[i] && H[i] < v)
               --co[H[i]], H[i] = v + 1;
           }
         }
         hi = H[u];
       } else if (cur[u]->c && H[u] == H[cur[u]->dest]+1){
         addFlow(*cur[u], min(ec[u], cur[u]->c));
       }else ++cur[u];
    }
  }
  bool inCut(int a) { return H[a] >= sz(g); }
};
```

Blossom.h

Description: Max matching on general Graph. $mate[i]$ = match of i

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

d41d8c, 56 lines

```cpp
vector<int> Blossom(vector<vector<int>>& g) {
    int n = sz(g), timer = -1;
    vector<int> mate(n, -1),label(n),par(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[par[mate[x]]]);
        }
    };
    auto blossom = [&](int v, int w, int a) {
        while (orig[v] != a) {
            par[v] = w; w = mate[v];
            if(label[w] == 1) label[w] = 0,q.push_back(w);
            orig[v] = orig[w] = a;
            v = par[w];
        }
    };
    auto aug = [&](int v) {
        while (v != -1) {
            int pv = par[v], nv = mate[pv];
            mate[v] = pv; mate[pv] = v; v = nv;
        }
    };
    auto bfs = [&](int root) {
        fill(all(label), -1);
        iota(all(orig), 0);
        q.clear();
        label[root] = 0; q.push_back(root);
        rep(i, 0, sz(q)) {
            int v = q[i];
            for (auto x : g[v]) {
                if (label[x] == -1) {
                    label[x] = 1; par[x] = v;
                    if (mate[x] == -1) return aug(x), 1;
                    label[mate[x]] = 0;
                    q.push_back(mate[x]);
                }
                else if(!label[x] && 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.
    rep(i, 0, n) {
        if (mate[i] == -1) bfs(i);
    }
    return mate;
}
```

HopcroftKarp.h

Description: ans is the size of the max matching.
The match of x is $l[x]$
Usage: `HopcroftKarp(|X|, |Y|, edges(x, y))`
Time: $\mathcal{O}\left(\sqrt{V}E\right)$

d41d8c, 46 lines

```cpp
struct HopcroftKarp {
    vector<int> g, l, r;
    int ans;
    HopcroftKarp(int n, int m, vector<pii> e)
        : g(sz(e)), l(n, -1), r(m, -1), ans(0) {
        shuffle(all(e), rng);
        vector<int> deg(n + 1);
        for (auto& [x, y] : e) deg[x]++;
        rep(i, 1, n+1) deg[i] += deg[i - 1];
        for (auto& [x, y] : e) g[--deg[x]] = y;

        vector<int> q(n);
        while (true) {
            vector<int> a(n, -1), p(n, -1);
            int t = 0;
            rep(i, 0, n){
                if (l[i] == -1) {
                    q[t++] = a[i] = p[i] = i;
                }
            }
            bool match = false;
            rep(i, 0, t){
                int x = q[i];
                if (~l[a[x]]) continue;
                rep(j, deg[x], deg[x+1]) {
                    int y = g[j];
                    if (r[y] == -1) {
                        while (~y) {
                            r[y] = x;
                            swap(l[x], y);
                            x = p[x];
                        }
                        match = true, ans++;
                        break;
                    }
                    if (p[r[y]] == -1) {
                        q[t++] = y = r[y];
                        p[y] = x, a[y] = a[x];
                    }
                }
            }
            if (!match) break;
        }
    }
};
```

WeightedMatching.h

Description: Given a weighted bipartite graph, matches every node on the left with a node on the right such that no nodes are in two matchings and the sum of the edge weights is minimal. Takes cost[N][M], where cost[i][j] = cost for L[i] to be matched with R[j] and returns (min cost, match), where L[i] is matched with R[match[i]]. Negate costs for max cost. Requires $N \leq M$.
Time: $\mathcal{O}\left(N^2M\right)$

d41d8c, 41 lines

```cpp
pair<ll, vector<int>> hunga(const vector<vector<ll>>& a) {
    if (a.empty()) return { 0, {} };
    int n = sz(a) + 1, m = sz(a[0]) + 1;
    vector<ll> u(n), v(m), p(m);
    vector<int> ans(n - 1);
    for (int i = 1; i < n; i++) {
        p[0] = i;
        int j0 = 0;
        vector<ll> dist(m, LLONG_MAX), pre(m, -1);
        vector<bool> done(m + 1);
        do {
            done[j0] = true;
            ll i0 = p[j0], j1 = -1, delta = LLONG_MAX;
            for (int j = 1; j < m; j++) {
                if (!done[j]) {
                    ll cur = a[i0-1][j-1] - u[i0] - v[j];
                    if (cur < dist[j])
                        dist[j] = cur, pre[j] = j0;
                    if(dist[j] < delta)
                        delta = dist[j], j1 = j;
                }
            }
            for (int j = 0; j < m; j++) {
                if (done[j])
                    u[p[j]] += delta, v[j] -= delta;
                else dist[j] -= delta;
            }
            assert(j1 != -1);
            j0 = j1;
        } while (p[j0]);
        while (j0) {
            int j1 = pre[j0];
            p[j0] = p[j1], j0 = j1;
        }
    }
    for (int j = 1; j < m; j++) {
        if (p[j]) ans[p[j] - 1] = j - 1;
    }
    return { -v[0], ans }; // min cost
}
```

GlobalMinCut.h

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

d41d8c, 22 lines

```cpp
pair<int, vi> globalMinCut(vector<vi> mat) {
 pair<int, vi> best = {INT_MAX, {}};
 int n = sz(mat);
 vector<vi> co(n);
 rep(i,0,n) co[i] = {i};
 rep(ph,1,n) {
   vi w = mat[0];
   size_t s = 0, t = 0;
   rep(it,0,n-ph) { // O(V^2) -> O(E log V) with prio.
queue
     w[t] = INT_MIN;
     s = t, t = max_element(all(w)) - w.begin();
     rep(i,0,n) w[i] += mat[t][i];
   }
   best = min(best, {w[t] - mat[t][t], co[t]});
   co[s].insert(co[s].end(), all(co[t]));
   rep(i,0,n) mat[s][i] += mat[t][i];
   rep(i,0,n) mat[i][s] = mat[s][i];
   mat[0][t] = INT_MIN;
 }
 return best;
}
```

7.3 DFS algorithms

Bridges.h

d41d8c, 24 lines

```cpp
vector<int> g[ms];
int low[ms], tin[ms], vis[ms], t;

void dfs(int u = 0, int p = -1) {
    vis[u] = true;
    low[u] = tin[u] = t++;
    for (auto v : g[u]) {
        if (v == p) continue;
        if (vis[v]) {
            low[u] = min(low[u], tin[v]);
        }
        else {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
            // if (low[v] >= tin[u] && p != -1), U is an
                articulation point
            if (low[v] > tin[u]) {
                // edge from U to V is a bridge
            }
            // children++
        }
    }
    // if(children > 1 && p == -1) root is an articulation
        point
}
```

BridgeOnline.h

Description: Maintains bridges and 2-edge-connected components (2-ECC) incrementally. `ds[0]` tracks Connected Components (CC). `ds[1]` tracks 2-ECCs. Nodes u, v are in the same 2-ECC iff `dsfind(u, 1) == dsfind(v, 1)`. `g` stores the spanning forest edges (edges that were bridges when added). An edge $(u, v) \in g$ is a current bridge iff `dsfind(u, 1) != dsfind(v, 1)`. `bridges` tracks the total count of active bridges. Use `init()` before starting.
Time: Amortized $\mathcal{O}\left(\log N\right)$

d41d8c, 75 lines

```cpp
int bridges;
int ds[2][ms], sz[2][ms];
int h[ms], pai[ms], old[ms];
vector<int> g[ms];

void init() {
    bridges = 0;
    rep(i, 0, ms) {
        g[i].clear(), h[i] = 0;
        ds[0][i] = ds[1][i] = i;
        sz[0][i] = sz[1][i] = 1;
    }
}

int dsfind(int j, int i) {
    if(j == ds[i][j]) return ds[i][j];
    return ds[i][j] = dsfind(ds[i][j], i);
}

void dfs(int u, int p, int l) {
    h[u] = l;
    pai[u] = p;
    old[u] = dsfind(u, 1);
    for (int v : g[u]) {
        if (v == p) continue;
        dfs(v, u, l + 1);
    }
}

void updateNodes(int u, int p) {
    if (old[u] == old[p]) {
        ds[1][u] = ds[1][p];
    }
    else ds[1][u] = u;
    for (int v : g[u]) {
        if (v == p) continue;
        updateNodes(v, u);
    }
}
```

```cpp
void mergeTrees(int a, int b) {
    bridges++;
    int iniA = a, iniB = b;
    a = dsfind(a, 0), b = dsfind(b, 0);
    if (sz[0][a] < sz[0][b]) swap(a, b), swap(iniA, iniB);
    dfs(iniB, iniA, h[iniA] + 1);
    old[iniA] = -1;
    updateNodes(iniB, iniA);
    ds[0][b] = a;
    sz[0][a] += sz[0][b];
}

void removeBridges(int a, int b) {
    a = dsfind(a, 1), b = dsfind(b, 1);
    while (a != b) {
        bridges--;
        if (h[a] < h[b]) swap(a, b);
        // ponte entre (a, pai[a]) deixou de existir
        ds[1][a] = dsfind(pai[a], 1);
        a = ds[1][a];
    }
}

void addEdge(int a, int b) {
    if (dsfind(a, 0) == dsfind(b, 0)) {
        removeBridges(a, b);
    }
    else {
        // nova ponte entre (a, b)
        g[a].push_back(b);
        g[b].push_back(a);
        mergeTrees(a, b);
    }
}
```

BlockCutTree.h

Description: Constructs the Block-Cut Tree, which is a bipartite graph with blocks (maximal 2-vertex-connected components) on one side and articulation points on the other. Works for disconnected graphs. Tree size is $\leq 2N$. Be careful with self loops and multi edges. `art[i]`: number of new components created by removing i (AP if ≥ 1). `blocks[i]`, `edgblocks[i]`: vertices/edges of block i. `tree[i]`: the tree node index corresponding to block i. `pos[i]`: the tree node index corresponding to vertex i.
Time: $\mathcal{O}(N+M)$

d41d8c, 66 lines

```cpp
struct block_cut_tree {
 vector<vector<int>> g, blocks, tree;
 vector<vector<pair<int, int>>> edgblocks;
 stack<int> s;
 stack<pair<int, int>> s2;
 vector<int> id, art, pos;

 block_cut_tree(vector<vector<int>> g_) : g(g_) {
   int n = sz(g);
   id.resize(n, -1), art.resize(n), pos.resize(n);
   build();
 }

 int dfs(int i, int& t, int p = -1) {
   int lo = id[i] = t++;
   s.push(i);

   if (p != -1) s2.emplace(i, p);
   for (int j : g[i])
          if (j != p and id[j] != -1) s2.emplace(i, j);

   for (int j : g[i]) if (j != p) {
     if (id[j] == -1) {
       int val = dfs(j, t, i);
       lo = min(lo, val);

       if (val >= id[i]) {
         art[i]++;
         blocks.emplace_back(1, i);
         while (blocks.back().back() != j)
           blocks.back().push_back(s.top()), s.pop();

         edgblocks.emplace_back(1, s2.top()), s2.pop();
         while (edgblocks.back().back() != pii(j, i))
           edgblocks.back().push_back(s2.top()), s2.pop();
       }
     }
     else lo = min(lo, id[j]);
   }
   if (p == -1) {
          if(art[i]) art[i]--;
          else{
              blocks.emplace_back(1, i);
              edgblocks.emplace_back();
          }
      }
   return lo;
 }

 void build() {
   int t = 0;
    rep(i, 0, sz(g)) if(id[i] == -1) dfs(i, t, -1);
   tree.resize(sz(blocks));
   rep(i, 0, sz(g)) if (art[i])
     pos[i] = sz(tree), tree.emplace_back();

   rep(i, 0, sz(blocks)) for (int j : blocks[i]) {
     if (!art[j]) pos[j] = i;
     else {
              tree[i].push_back(pos[j]);
              tree[pos[j]].push_back(i);
          }
   }
 }
};
```

DominatorTree.h

Description: Builds the Dominator Tree of a directed graph rooted at `root`. Node u dominates v if every path from `root` to v passes through u. The immediate dominator of v is the unique dominator closest to v (excluding v). Returns a vector `par` where `par[u]` is the parent of u in the tree. Roots and unreachable nodes satisfy `par[u] = u`.
Time: $\mathcal{O}(M \log N)$

d41d8c, 55 lines

```cpp
struct dominator_tree {
 int n, t;
 vector<vector<int>> g, rg, bucket;
 vector<int> arr, par, rev, sdom, dom, ds, lbl;

 dominator_tree(int n) : n(n),t(0),g(n),rg(n),bucket(n),
   arr(n,-1),par(n),rev(n),sdom(n),dom(n),ds(n),lbl(n) {}

 void add_edge(int u, int v) { g[u].push_back(v); }

 void dfs(int u) {
   arr[u] = t;
   rev[t] = u;
   lbl[t] = sdom[t] = ds[t] = t;
    t++;
   for (int w : g[u]) {
     if (arr[w] == -1) {
       dfs(w);
       par[arr[w]] = arr[u];
     }
     rg[arr[w]].push_back(arr[u]);
   }
 }
 int find(int u, int x=0) {
   if (u == ds[u]) return x ? -1 : u;
   int v = find(ds[u], x+1);
   if (v < 0) return u;
   if(sdom[lbl[ds[u]]] < sdom[lbl[u]]) lbl[u]= lbl[ds[u]];
   ds[u] = v;
   return x ? v : lbl[u];
 }

 vector<int> run(int root) {
   dfs(root);
   iota(all(dom), 0);
   for (int i=t-1; i>=0; i--) {
     for(int w : rg[i]) sdom[i] = min(sdom[i],sdom[find(w)
]);
     if (i) bucket[sdom[i]].push_back(i);
     for (int w : bucket[i]) {
       int v = find(w);
       if (sdom[v] == sdom[w]) dom[w] = sdom[w];
       else dom[w] = v;
     }
     if (i > 1) ds[i] = par[i];
   }
   rep(i, 1, t) {
     if (dom[i] != sdom[i]) dom[i] = dom[dom[i]];
   }
   vector<int> par(n);
   iota(all(par), 0);
   rep(i, 0, t) par[rev[i]] = rev[dom[i]];
   return par;
 }
};
```

EulerPath.h

Description: Receives as input graph(node, edge index), number of edges and source. Returns list of node, index of edge he came from, if path/circuit does not exists returns empty list.

d41d8c, 27 lines

```cpp
vector<pii> eulerPath(const vector<vector<pii>>& g, int
nedges, int src) {
    int n = sz(g);
    vector<int> deg(n, 0), its(n, 0), used(nedges + 1, 0);
    vector<pii> s = { {src, -1} };
    //deg[src]++; //to allow paths, not only circuits
    vector<pii> ret;
    while (!s.empty()) {
        int u = s.back().first, &it = its[u];
        if (it == sz(g[u])) {
            ret.push_back(s.back());
            s.pop_back();
            continue;
        }
        auto& [nxt, id] = g[u][it++];
        if (!used[id]) {
            deg[u]--, deg[nxt]++;
            used[id] = 1;
            s.push_back({ nxt, id });
        }
    }
    for (int x : deg) {
        if (x < 0 || sz(ret) != (nedges + 1)) return {};
    }
    reverse(ret.begin(), ret.end());
    return ret;
}
```

SCC.h

Description: Kosaraju algorithm for calculating strongly connected components. Components are ordered in topological order.

d41d8c, 36 lines

```
d41 struct SCC {
d41     int n, ncomp;
d41     vector<vector<int>> g, inv;
d41     vector<int> comp, vis, stk;
d41     SCC(){}
d41     SCC(int n)
d41         : n(n),ncomp(0),g(n),inv(n),comp(n, -1),vis(n){}

d41     void dfs(int u) {
d41         vis[u] = 1;
d41         for (int v : g[u]) if (!vis[v]) dfs(v);
d41         stk.push_back(u);
d41     }
d41     void dfs_inv(int u) {
d41         comp[u] = ncomp;
d41         for (int v : inv[u]) {
d41             if (comp[v] == -1) dfs_inv(v);
d41         }
d41     }
d41     void solve() {
d41         for (int i = 0; i < n; i++) {
d41             if (!vis[i]) dfs(i);
d41         }
d41         reverse(all(stk));
d41         for (int u : stk) {
d41             if (comp[u] != -1) continue;
d41             dfs_inv(u);
d41             ncomp++;
d41         }
d41     }
d41     void add_edge(int a, int b) {
d41         g[a].push_back(b);
d41         inv[b].push_back(a);
d41     }
d41 };
```

TwoSat.h

Usage: not A = ~A

"SCC.h" d41d8c, 37 lines

```
d41 struct TwoSat{
d41  int n;
d41  SCC scc;
d41  vector<int> value;
d41  vector<pii> e;
d41  TwoSat(int n) : n(n){}
d41  bool solve(){
d41    value.resize(n);
d41    scc = SCC(2*n);
d41    for(auto &x : e) scc.add_edge(x.first, x.second);
d41    scc.solve();
d41    for(int i=0; i<2*n; i++)
d41      if(scc.comp[i] == scc.comp[i^1]) return false;
d41    for(int i=0; i<n; i++)
d41      value[i] = scc.comp[id(i)] > scc.comp[id(~i)];
d41    return true;
d41  }
d41  void atMostOne(vector<int> &li){
d41    if(sz(li) <= 1) return;
d41    int cur = ~li[0];
d41    for(int i = 2; i < sz(li); i++) {
d41      int next = n++;
d41      addOr(cur, ~li[i]);
d41      addOr(cur, next);
d41      addOr(~li[i], next);
d41      cur = ~next;
d41    }
d41    addOr(cur, ~li[1]);
d41  }
d41  int id(int v) { return v < 0 ? (~v) * 2 ^ 1 : v * 2; }
d41  void add(int a, int b) { e.push_back({id(a), id(b)}); }
d41  void addOr(int a, int b) { add(~a, b); add(~b, a); }
d41  void addImp(int a, int b) { addOr(~a, b); }
d41  void addEqual(int a, int b){ addOr(a, ~b); addOr(~a, b);
     }
d41  void isFalse(int a) { addImp(a, ~a); }
d41 };
```

7.4 Coloring

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: $\mathcal{O}(NM)$

d41d8c, 32 lines

```
d41 vi edgeColoring(int N, vector<pii> eds) {
d41  vi cc(N + 1), ret(sz(eds)), fan(N), free(N), loc;
d41  for (pii e : eds) ++cc[e.first], ++cc[e.second];
d41  int u, v, ncols = *max_element(all(cc)) + 1;
d41  vector<vi> adj(N, vi(ncols, -1));
d41  for (pii e : eds) {
d41    tie(u, v) = e;
d41    fan[0] = v;
d41    loc.assign(ncols, 0);
d41    int at = u, end = u, d, c = free[u], ind = 0, i = 0;
d41    while (d = free[v], !loc[d] && (v = adj[u][d]) != -1)
d41      loc[d] = ++ind, cc[ind] = d, fan[ind] = v;
d41    cc[loc[d]] = c;
d41    for (int cd = d; at != -1; cd ^= c ^ d, at = adj[at][cd
     ])
d41      swap(adj[at][cd], adj[end = at][cd ^ c ^ d]);
d41    while (adj[fan[i]][d] != -1) {
d41      int left = fan[i], right = fan[++i], e = cc[i];
d41      adj[u][e] = left;
d41      adj[left][e] = u;
d41      adj[right][e] = -1;
d41      free[right] = e;
d41    }
d41    adj[u][d] = fan[i];
d41    adj[fan[i]][d] = u;
d41    for (int y : {fan[0], u, end})
d41      for (int& z = free[y] = 0; adj[y][z] != -1; z++);
d41  }
d41  rep(i,0,sz(eds))
d41    for (tie(u, v) = eds[i]; adj[u][ret[i]] != v;) ++ret[i
     ];
d41  return ret;
d41 }
```

7.5 Heuristics

MaxClique.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.

d41d8c, 53 lines

```
d41 using vb = vector<bitset<200>>;
d41 struct Maxclique {
d41  double limit=0.025, pk=0;
d41  struct Vertex { int i, d=0; };
d41  using vv = vector<Vertex>;
d41  vb e;
d41  vv V;
d41  vector<vector<int>> C;
d41  vector<int> qmax, q, S, old;
d41  void init(vv& r) {
d41    for (auto& v : r) v.d = 0;
d41    for (auto& v : r) for (auto j : r) v.d += e[v.i][j.i];
d41    sort(all(r), [](auto a, auto b) { return a.d > b.d; });
d41    int mxD = r[0].d;
d41    for(int i=0; i<sz(r); i++) r[i].d = min(i, mxD) + 1;
d41  }
d41  void expand(vv& R, int lev = 1) {
d41    S[lev] += S[lev - 1] - old[lev];
d41    old[lev] = S[lev - 1];
d41    while (sz(R)) {
d41      if (sz(q) + R.back().d <= sz(qmax)) return;
d41      q.push_back(R.back().i);
d41      vv T;
d41      for(auto v : R)
d41        if (e[R.back().i][v.i]) T.push_back({v.i});
d41      if (sz(T)) {
d41        if (S[lev]++ / ++pk < limit) init(T);
d41        int j = 0, mxk = 1, mnk = max(sz(qmax)-sz(q)+1, 1);
d41        C[1].clear(), C[2].clear();
d41        for (auto v : T) {
d41          int k = 1;
d41          auto f = [&](int i) { return e[v.i][i]; };
d41          while (any_of(all(C[k]), f)) k++;
d41          if (k > mxk) mxk = k, C[mxk + 1].clear();
d41          if (k < mnk) T[j++].i = v.i;
d41          C[k].push_back(v.i);
d41        }
d41        if (j > 0) T[j - 1].d = 0;
d41        for(int k=mnk; k<mxk + 1; k++){
d41                    for (int i : C[k])
d41              T[j].i = i, T[j++].d = k;
d41                }
d41        expand(T, lev + 1);
d41      } else if (sz(q) > sz(qmax)) qmax = q;
d41      q.pop_back(), R.pop_back();
d41    }
d41  }
d41  vector<int> maxClique(){ init(V),expand(V); return qmax;}
d41  Maxclique(vb conn) : e(conn),C(sz(e)+1),S(sz(C)),old(S){
d41    for(int i=0; i<sz(e); i++) V.push_back({i});
d41  }
d41 };
```

MaximalCliques.h

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

Time: $\mathcal{O}\left(3^{n/3}\right)$, much faster for sparse graphs

d41d8c, 13 lines

```
d41 typedef bitset<128> B;
d41 template<class F>
d41 void cliques(vector<B>& eds, F f, B P = ~B(), B X={}, B R
     ={}) {
d41  if (!P.any()) { if (!X.any()) f(R); return; }
d41  auto q = (P | X)._Find_first();
d41  auto cands = P & ~eds[q];
d41  rep(i,0,sz(eds)) if (cands[i]) {
d41    R[i] = 1;
d41    cliques(eds, f, P & eds[i], X & eds[i], R);
d41    R[i] = P[i] = 0; X[i] = 1;
d41  }
d41 }
```

7.6 Trees

Centroid.h

Description: Call decomp(0) to solve, marked array should be initially set to zero.
Time: $\mathcal{O}(N \log N)$

d41d8c, 27 lines

```
d41 int tam[ms], marked[ms];

d41 int calc_tam(int u, int p) {
d41   tam[u] = 1;
d41   for (int v : g[u]) {
d41     if (v != p && !marked[v]) tam[u] += calc_tam(v, u);
d41   }
d41   return tam[u];
d41 }

d41 int get_centroid(int u, int p, int tot) {
d41   for (int v : g[u]) {
d41     if (v != p && !marked[v] && (tam[v] > (tot / 2)))
d41       return get_centroid(v, u, tot);
d41   }
d41   return u;
d41 }
    // Cent is a child of P in the centroid tree
d41 void decomp(int u, int p = -1) {
d41   calc_tam(u, -1);
d41   int cent = get_centroid(u, -1, tam[u]);
d41   marked[cent] = 1;
d41   for (int v : g[cent]) {
d41     if (!marked[v]) decomp(v, cent);
d41   }
d41 }
```

HLD.h

Description: If values are stored on edges, set EDGE = true and store each edge's value at the endpoint farther from the root (the deeper node).
rp[i] is the representative (head) of the heavy path containing node i: it is the node in that chain that is closest to the root.

d41d8c, 51 lines

```
d41 template<bool EDGE> struct HLD {
d41     int n, t;
d41     vector<vector<int>> g;
d41     vector<int> pai, rp, tam, pos, val, arr;
d41     Seg seg;
d41     HLD(int n, vector<vector<int>>& g, vector<int>& val)
d41         : n(n), t(0), g(g), pai(n), rp(n), tam(n, 1),
d41           pos(n), val(val), arr(n) {
d41         calc_tam(0, -1);
d41         dfs(0, -1);
d41         seg.build(arr);
d41     }

d41     int calc_tam(int u, int p) {
d41         pai[u] = p;
d41         for (int& v : g[u]) {
d41             if (v == p) continue;
d41             tam[u] += calc_tam(v, u);
d41             if(tam[v] > tam[g[u][0]] || g[u][0] == p)
d41                 swap(g[u][0], v);
d41         }
d41         return tam[u];
d41     }

d41     void dfs(int u, int p) {
d41         pos[u] = t++;
d41         arr[pos[u]] = val[u];
d41         for (int v : g[u]) {
d41             if (v == p) continue;
d41             rp[v] = (v == g[u][0] ? rp[u] : v);
d41             dfs(v, u);
d41         }
d41     }

d41     int query(int a, int b) { // query on the path from a
    to b
d41         int ans = 0; // neutral value
d41         while (rp[a] != rp[b]) {
d41             if (pos[a] < pos[b]) swap(a, b);
d41             ans = max(ans, seg.query(pos[rp[a]], pos[a]));
d41             a = pai[rp[a]];
d41         }
d41         if(pos[a] > pos[b]) swap(a, b);
d41         ans = max(ans, seg.query(pos[a] + EDGE, pos[b]));
d41         return ans;
d41     }

d41     void update(int a, int x) {
d41         seg.update(pos[a], x);
d41     }
d41 };
```

LCA.h

Description: LCA algorithm using binary lifting, $is\_ancestor(a, b)$ returns true if a is an ancestral of b and false otherwise.
Time: $\mathcal{O}(N \log N)$

d41d8c, 26 lines

```
d41 int tin[MAXN], tout[MAXN], timer=0;
d41 int up[MAXN][BITS];
d41 void dfs(int u, int p){
d41     tin[u] = timer++, up[u][0] = p;
d41     for (int i=1; i<BITS; i++) {
d41         up[u][i] = up[up[u][i-1]][i-1];
d41     }
d41     for (int v: g[u]) if (v != p) dfs(v, u);
d41     tout[u] = timer;
d41 }

d41 bool is_ancestor(int u, int v){
d41     return (tin[u] <= tin[v] && tout[u] >= tout[v]);
d41 }

d41 int lca(int u, int v){
d41     if (is_ancestor(u, v)) return u;
d41     if (is_ancestor(v, u)) return v;
d41     for (int i=BITS-1; i>=0; i--) {
d41         if (up[u][i] && !is_ancestor(up[u][i], v)) {
d41             u = up[u][i];
d41         }
d41     }
d41     return up[u][0];
d41 }
```

VirtualTree.h

Description: Given a rooted tree and a subset S of nodes, compute the minimal subtree that contains all the nodes by adding all (at most $|S|-1$) pairwise LCA's and compressing edges. virt[u] is the adjacency list of the virtual tree: it stores pairs (v, dist), where v is a neighbor of u in the virtual tree and dist is the distance between u and v in the original tree.
Time: $\mathcal{O}(|S| \log |S|)$

"LCA.h" d41d8c, 24 lines

```
d41 vector<pair<int, int>> virt[ms];

d41 void build_virt(vector<int>& v) {
d41   auto cmp = [&](int i, int j){ return tin[i] < tin[j]; };
d41   sort(all(v), cmp);
d41   for (int i = 0, n = sz(v);i + 1 < n; i++)
d41     v.push_back(lca(v[i], v[i + 1]));
d41   sort(all(v), cmp);
d41   v.erase(unique(all(v)), v.end());
d41   stack<int> st;
d41   for (auto u : v) {
d41     if (st.empty()) {
d41       st.push(u);
d41     }
d41     else {
d41       while(sz(st) && !is_ancestor(st.top(), u)) st.pop();
d41       int p = st.top();
d41       virt[p].emplace_back(u, abs(lvl[u] - lvl[p]));
d41       virt[u].emplace_back(p, abs(lvl[u] - lvl[p]));
d41       st.push(u);
d41     }
d41   }
d41 }
```

DirectedMST.h

Description: Finds a minimum spanning tree/arborescence of a directed graph, given a root node. If no MST exists, returns -1.
Time: $\mathcal{O}(E \log V)$

"../data-structures/UnionFindRollback.h" d41d8c, 61 lines

```
d41 struct Edge { int a, b; ll w; };
d41 struct Node {
d41  Edge key;
d41  Node *l, *r;
d41  ll delta;
d41  void prop() {
d41    key.w += delta;
d41    if (l) l->delta += delta;
d41    if (r) r->delta += delta;
d41    delta = 0;
d41  }
d41  Edge top() { prop(); return key; }
d41 };
d41 Node *merge(Node *a, Node *b) {
d41  if (!a || !b) return a ?: b;
d41  a->prop(), b->prop();
d41  if (a->key.w > b->key.w) swap(a, b);
d41  swap(a->l, (a->r = merge(b, a->r)));
d41  return a;
d41 }
d41 void pop(Node*& a) { a->prop(); a = merge(a->l, a->r); }

d41 pair<ll, vi> dmst(int n, int r, vector<Edge>& g) {
d41  RollbackUF uf(n);
d41  vector<Node*> heap(n);
d41  for (Edge e : g) heap[e.b] = merge(heap[e.b], new Node{e
    });
d41  ll res = 0;
d41  vi seen(n, -1), path(n), par(n);
d41  seen[r] = r;
d41  vector<Edge> Q(n), in(n, {-1,-1}), comp;
d41  deque<tuple<int, int, vector<Edge>>> cycs;
d41  rep(s,0,n) {
d41    int u = s, qi = 0, w;
d41    while (seen[u] < 0) {
d41      if (!heap[u]) return {-1,{}};
d41      Edge e = heap[u]->top();
d41      heap[u]->delta -= e.w, pop(heap[u]);
d41      Q[qi] = e, path[qi++] = u, seen[u] = s;
d41      res += e.w, u = uf.find(e.a);
d41      if (seen[u] == s) {
d41        Node* cyc = 0;
d41        int end = qi, time = uf.time();
d41        do cyc = merge(cyc, heap[w = path[--qi]]);
d41        while (uf.join(u, w));
d41        u = uf.find(u), heap[u] = cyc, seen[u] = -1;
d41        cycs.push_front({u, time, {&Q[qi], &Q[end]}});
d41      }
```

```
d41     }
d41     rep(i,0,qi) in[uf.find(Q[i].b)] = Q[i];
d41   }

d41   for (auto& [u,t,comp] : cycs) { // restore sol (optional)
d41     uf.rollback(t);
d41     Edge inEdge = in[u];
d41     for (auto& e : comp) in[uf.find(e.b)] = e;
d41     in[uf.find(inEdge.b)] = inEdge;
d41   }
d41   rep(i,0,n) par[i] = in[i].a;
d41   return {res, par};
d41 }
```

Geometry (8)

8.1 Geometric primitives

Point.h

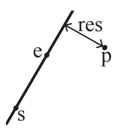
Description: Class to handle points in the plane. T can be e.g. double or long long. (Avoid int.)

d41d8c, 29 lines

```
d41 template <class T> int sgn(T x) { return (x > 0) - (x < 0); }
d41 template<class T>
d41 struct Point {
d41  typedef Point P;
d41  T x, y;
d41  explicit Point(T x=0, T y=0) : x(x), y(y) {}
d41  bool operator<(P p) const { return tie(x,y) < tie(p.x,p.y); }
d41  bool operator==(P p) const { return tie(x,y)==tie(p.x,p.y); }
d41  P operator+(P p) const { return P(x+p.x, y+p.y); }
d41  P operator-(P p) const { return P(x-p.x, y-p.y); }
d41  P operator*(T d) const { return P(x*d, y*d); }
d41  P operator/(T d) const { return P(x/d, y/d); }
d41  T dot(P p) const { return x*p.x + y*p.y; }
d41  T cross(P p) const { return x*p.y - y*p.x; }
d41  T cross(P a, P b) const { return (a-*this).cross(b-*this); }
d41  T dist2() const { return x*x + y*y; }
d41  double dist() const { return sqrt((double)dist2()); }
     // angle to x-axis in interval [-pi, pi]
d41  double angle() const { return atan2(y, x); }
d41  P unit() const { return *this/dist(); } // makes dist()=1
d41  P perp() const { return P(-y, x); } // rotates +90 degrees
d41  P normal() const { return perp().unit(); }
     // returns point rotated 'a' radians ccw around the origin
d41  P rotate(double a) const {
d41    return P(x*cos(a)-y*sin(a),x*sin(a)+y*cos(a)); }
d41  friend ostream& operator<<(ostream& os, P p) {
d41    return os << "(" << p.x << "," << p.y << ")"; }
d41 };
```

lineDistance.h

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

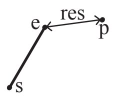


"Point.h" d41d8c, 5 lines

```
d41 template<class P>
d41 double lineDist(const P& a, const P& b, const P& p) {
d41  return (double)(b-a).cross(p-a)/(b-a).dist();
d41 }
```

SegmentDistance.h

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



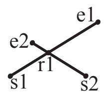
Usage: `Point<double> a, b(2,2), p(1,1);`
`bool onSegment = segDist(a,b,p) < 1e-10;`

"Point.h" d41d8c, 7 lines

```
d41 typedef Point<double> P;
d41 double segDist(P& s, P& e, P& p) {
d41  if (s==e) return (p-s).dist();
d41  auto d = (e-s).dist2(), t = min(d,max(.0,(p-s).dot(e-s)));
d41  return ((p-s)*d-(e-s)*t).dist()/d;
d41 }
```

SegmentIntersection.h

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



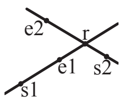
Usage: `vector<P> inter = segInter(s1,e1,s2,e2);`
`if (sz(inter)==1)`
`cout << "segments intersect at " << inter[0] << endl;`

"Point.h", "OnSegment.h" d41d8c, 14 lines

```
d41 template<class P> vector<P> segInter(P a, P b, P c, P d) {
d41  auto oa = c.cross(d, a), ob = c.cross(d, b),
d41       oc = a.cross(b, c), od = a.cross(b, d);
     // Checks if intersection is single non-endpoint point.
d41  if (sgn(oa) * sgn(ob) < 0 && sgn(oc) * sgn(od) < 0)
d41    return {(a * ob - b * oa) / (ob - oa)};
d41  set<P> s;
d41  if (onSegment(c, d, a)) s.insert(a);
d41  if (onSegment(c, d, b)) s.insert(b);
d41  if (onSegment(a, b, c)) s.insert(c);
d41  if (onSegment(a, b, d)) s.insert(d);
d41  return {all(s)};
d41 }
```

lineIntersection.h

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



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

"Point.h" d41d8c, 9 lines

```
d41 template<class P>
d41 pair<int, P> lineInter(P s1, P e1, P s2, P e2) {
d41  auto d = (e1 - s1).cross(e2 - s2);
d41  if (d == 0) // if parallel
d41    return {-(s1.cross(e1, s2) == 0), P(0, 0)};
d41  auto p = s2.cross(e1, e2), q = s2.cross(e2, s1);
d41  return {1, (s1 * p + e1 * q) / d};
d41 }
```

sideOf.h

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

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

"Point.h" d41d8c, 10 lines

```
d41 template<class P>
d41 int sideOf(P s, P e, P p) { return sgn(s.cross(e, p)); }

d41 template<class P>
d41 int sideOf(const P& s, const P& e, const P& p, double eps) {
d41  auto a = (e-s).cross(p-s);
d41  double l = (e-s).dist()*eps;
d41  return (a > l) - (a < -l);
d41 }
```

OnSegment.h

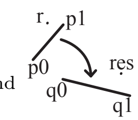
Description: Returns true iff p lies on the line segment from s to e. Use `(segDist(s,e,p)<=epsilon)` instead when using Point<double>.

"Point.h" d41d8c, 4 lines

```
d41 template<class P> bool onSegment(P s, P e, P p) {
d41  return p.cross(s, e) == 0 && (s - p).dot(e - p) <= 0;
d41 }
```

linearTransformation.h

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



"Point.h" d41d8c, 7 lines

```
d41 typedef Point<double> P;
d41 P linearTransformation(const P& p0, const P& p1,
d41     const P& q0, const P& q1, const P& r) {
d41  P dp = p1-p0, dq = q1-q0, num(dp.cross(dq), dp.dot(dq));
d41  return q0 + P((r-p0).cross(num), (r-p0).dot(num))/dp.dist2();
d41 }
```

LineProjectionReflection.h

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

"Point.h" d41d8c, 6 lines

```
d41 template<class P>
d41 P lineProj(P a, P b, P p, bool refl=false) {
d41  P v = b - a;
d41  return p - v.perp()*(1+refl)*v.cross(p-a)/v.dist2();
d41 }
```

Angle.h

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

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

d41d8c, 36 lines

```
d41 struct Angle {
```

```cpp
  int x, y;
  int t;
  Angle(int x, int y, int t=0) : x(x), y(y), t(t) {}
  Angle operator-(Angle b) const { return {x-b.x, y-b.y, t}; }
  int half() const {
    assert(x || y);
    return y < 0 || (y == 0 && x < 0);
  }
  Angle t90() const { return {-y, x, t + (half() && x >= 0)}; }
  Angle t180() const { return {-x, -y, t + half()}; }
  Angle t360() const { return {x, y, t + 1}; }
};
bool operator<(Angle a, Angle b) {
  // add a.dist2() and b.dist2() to also compare distances
  return make_tuple(a.t, a.half(), a.y * (ll)b.x) <
         make_tuple(b.t, b.half(), a.x * (ll)b.y);
}

// Given two points, this calculates the smallest angle between
// them, i.e., the angle that covers the defined line segment.
pair<Angle, Angle> segmentAngles(Angle a, Angle b) {
  if (b < a) swap(a, b);
  return (b < a.t180() ?
          make_pair(a, b) : make_pair(b, a.t360()));
}
Angle operator+(Angle a, Angle b) { // point a + vector b
  Angle r(a.x + b.x, a.y + b.y, a.t);
  if (a.t180() < r) r.t--;
  return r.t180() < a ? r.t360() : r;
}
Angle angleDiff(Angle a, Angle b) { // angle b - angle a
  int tu = b.t - a.t; a.t = b.t;
  return {a.x*b.x + a.y*b.y, a.x*b.y - a.y*b.x, tu - (b < a)};
}
```

HalfPlane.h

Description: Computes the intersection of a set of half-planes. Half-planes are sorted by angle and processed with a deque, removing redundant or conflicting constraints. Parallel half-planes are handled explicitly. Returns the convex polygon of the intersection, or empty if infeasible.
Time: $\mathcal{O}(n \log n)$

"Point.h" d41d8c, 72 lines

```cpp
using ld = long double;
using P = Point<ld>;

struct Hp { // Half plane struct
  // 'p' is a passing point of the line and 'pq' is the direction vector of the line.
  P p, pq;
  ld angle;

  Hp() {}
  Hp(const P& a, const P& b) : p(a), pq(b - a) {
    angle = atan2l(pq.y, pq.x);
  }
  bool out(const P& r) { return pq.cross(r - p) < -eps; }
  bool operator < (const Hp& e) const {
    return angle < e.angle;
  }
  friend P inter(const Hp& s, const Hp& t) {
    ld alpha = (t.p - s.p).cross(t.pq) / s.pq.cross(t.pq);
    return s.p + (s.pq * alpha);
  }
};

vector<P> hp_intersect(vector<Hp>& H) {
  P box[4] = { P(inf, inf), P(-inf, inf),
               P(-inf, -inf), P(inf, -inf) };

  for(int i = 0; i<4; i++) {
    Hp aux(box[i], box[(i+1) % 4]);
    H.push_back(aux);
  }
  sort(all(H));
  deque<Hp> dq;
  int len = 0;
  for(int i = 0; i < sz(H); i++) {
    while(len>1 && H[i].out(inter(dq[len-1], dq[len-2]))){
      dq.pop_back();
      --len;
    }
    while (len > 1 && H[i].out(inter(dq[0], dq[1]))){
      dq.pop_front();
      --len;
    }
    if(len && fabsl(H[i].pq.cross(dq[len-1].pq)) < eps){
      if (H[i].pq.dot(dq[len-1].pq) < 0.0)
        return vector<P>();
      if (H[i].out(dq[len-1].p)) {
        dq.pop_back();
        --len;
      }
      else continue;
    }
    dq.push_back(H[i]);
    ++len;
  }

  while(len> 2 && dq[0].out(inter(dq[len-1], dq[len-2]))){
    dq.pop_back();
    --len;
  }
  while (len > 2 && dq[len-1].out(inter(dq[0], dq[1]))){
    dq.pop_front();
    --len;
  }
  if (len < 3) return vector<P>();
  vector<P> ret(len);
  for(int i = 0; i+1 < len; i++) {
    ret[i] = inter(dq[i], dq[i+1]);
  }
  ret.back() = inter(dq[len-1], dq[0]);
  return ret;
}
```

8.2 Circles

CircleIntersection.h

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

"Point.h" d41d8c, 12 lines

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

CircleTangents.h

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

"Point.h" d41d8c, 14 lines

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

CircleLine.h

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

"Point.h" d41d8c, 10 lines

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

CirclePolygonIntersection.h

Description: Returns the area of the intersection of a circle with a ccw polygon.
Time: $\mathcal{O}(n)$

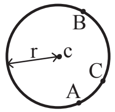
"../../content/geometry/Point.h" d41d8c, 20 lines

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

```
d41 }
```

circumcircle.h

Description:



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

"Point.h" d41d8c, 10 lines

```
d41 typedef Point<double> P;
d41 double ccRadius(const P& A, const P& B, const P& C) {
d41   return (B-A).dist()*(C-B).dist()*(A-C).dist()/
d41       abs((B-A).cross(C-A))/2;
d41 }
d41 P ccCenter(const P& A, const P& B, const P& C) {
d41   P b = C-A, c = B-A;
d41   return A + (b*c.dist2()-c*b.dist2()).perp()/b.cross(c)/2;
d41 }
```

MinimumEnclosingCircle.h

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

"circumcircle.h" d41d8c, 18 lines

```
d41 pair<P, double> mec(vector<P> ps) {
d41   shuffle(all(ps), mt19937(time(0)));
d41   P o = ps[0];
d41   double r = 0, EPS = 1 + 1e-8;
d41   rep(i,0,sz(ps)) if ((o - ps[i]).dist() > r * EPS) {
d41     o = ps[i], r = 0;
d41     rep(j,0,i) if ((o - ps[j]).dist() > r * EPS) {
d41       o = (ps[i] + ps[j]) / 2;
d41       r = (o - ps[i]).dist();
d41       rep(k,0,j) if ((o - ps[k]).dist() > r * EPS) {
d41         o = ccCenter(ps[i], ps[j], ps[k]);
d41         r = (o - ps[i]).dist();
d41       }
d41     }
d41   }
d41   return {o, r};
d41 }
```

8.3 Polygons

InsidePolygon.h

Description: Returns true if p lies within the polygon. If strict is true, it returns false for points on the boundary. The algorithm uses products in intermediate steps so watch out for overflow.
Usage: vector<P> v = {P{4,4}, P{1,2}, P{2,1}};
bool in = inPolygon(v, P{3, 3}, false);
Time: $\mathcal{O}(n)$

"Point.h", "OnSegment.h", "SegmentDistance.h" d41d8c, 12 lines

```
d41 template<class P>
d41 bool inPolygon(vector<P> &p, P a, bool strict = true) {
d41   int cnt = 0, n = sz(p);
d41   rep(i,0,n) {
d41     P q = p[(i + 1) % n];
d41     if (onSegment(p[i], q, a)) return !strict;
        //or: if (segDist(p[i], q, a) <= eps) return !strict;
d41     cnt ^= ((a.y<p[i].y) - (a.y<q.y)) * a.cross(p[i], q) > 0;
d41   }
d41   return cnt;
d41 }
```

PolygonArea.h

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

"Point.h" d41d8c, 7 lines

```
d41 template<class T>
d41 T polygonArea2(vector<Point<T>>& v) {
d41   T a = v.back().cross(v[0]);
d41   rep(i,0,sz(v)-1) a += v[i].cross(v[i+1]);
d41   return a;
d41 }
```

PolygonCenter.h

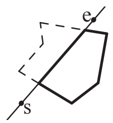
Description: Returns the center of mass for a polygon.
Time: $\mathcal{O}(n)$

"Point.h" d41d8c, 10 lines

```
d41 typedef Point<double> P;
d41 P polygonCenter(const vector<P>& v) {
d41   P res(0, 0); double A = 0;
d41   for (int i = 0, j = sz(v) - 1; i < sz(v); j = i++) {
d41     res = res + (v[i] + v[j]) * v[j].cross(v[i]);
d41     A += v[j].cross(v[i]);
d41   }
d41   return res / A / 3;
d41 }
```

PolygonCut.h

Description:



Returns a vector with the vertices of a polygon with everything to the left of the line going from s to e cut away.
Usage: vector<P> p = ...;
p = polygonCut(p, P(0,0), P(1,0));

"Point.h" d41d8c, 14 lines

```
d41 typedef Point<double> P;
d41 vector<P> polygonCut(const vector<P>& poly, P s, P e) {
d41   vector<P> res;
d41   rep(i,0,sz(poly)) {
d41     P cur = poly[i], prev = i ? poly[i-1] : poly.back();
d41     auto a = s.cross(e, cur), b = s.cross(e, prev);
d41     if ((a < 0) != (b < 0))
d41       res.push_back(cur + (prev - cur) * (a / (a - b)));
d41     if (a < 0)
d41       res.push_back(cur);
d41   }
d41   return res;
d41 }
```

PolygonUnion.h

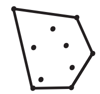
Description: Calculates the area of the union of n polygons (not necessarily convex). The points within each polygon must be given in CCW order. (Epsilon checks may optionally be added to sideOf/sgn, but shouldn't be needed.)
Time: $\mathcal{O}(N^2)$, where N is the total number of points

"Point.h", "sideOf.h" d41d8c, 34 lines

```
d41 typedef Point<double> P;
d41 double rat(P a, P b) { return sgn(b.x) ? a.x/b.x : a.y/b.y; }
d41 double polyUnion(vector<vector<P>>& poly) {
d41   double ret = 0;
d41   rep(i,0,sz(poly)) rep(v,0,sz(poly[i])) {
d41     P A = poly[i][v], B = poly[i][(v + 1) % sz(poly[i])];
d41     vector<pair<double, int>> segs = {{0, 0}, {1, 0}};
d41     rep(j,0,sz(poly)) if (i != j) {
d41       rep(u,0,sz(poly[j])) {
d41         P C = poly[j][u], D = poly[j][(u + 1) % sz(poly[j])];
d41         int sc = sideOf(A, B, C), sd = sideOf(A, B, D);
d41         if (sc != sd) {
d41           double sa = C.cross(D, A), sb = C.cross(D, B);
d41           if (min(sc, sd) < 0)
d41             segs.emplace_back(sa / (sa - sb), sgn(sc - sd));
d41         } else if (!sc && !sd && j<i && sgn((B-A).dot(D-C))>0){
d41           segs.emplace_back(rat(C - A, B - A), 1);
d41           segs.emplace_back(rat(D - A, B - A), -1);
d41         }
d41       }
d41     }
d41     sort(all(segs));
d41     for (auto& s : segs) s.first = min(max(s.first, 0.0), 1.0);
d41     double sum = 0;
d41     int cnt = segs[0].second;
d41     rep(j,1,sz(segs)) {
d41       if (!cnt) sum += segs[j].first - segs[j - 1].first;
d41       cnt += segs[j].second;
d41     }
d41     ret += A.cross(B) * sum;
d41   }
d41   return ret / 2;
d41 }
```

ConvexHull.h

Description:



Returns a vector of the points of the convex hull in counter-clockwise order. Points on the edge of the hull between two other points are not considered part of the hull. If you want to keep the collinear points in the convex hull, change the comparison to $h[t-2].cross(h[t-1], p) < 0$ and the size of the vector h to $2 * sz(pts) + 1$.
Time: $\mathcal{O}(n \log n)$

"Point.h" d41d8c, 14 lines

```
d41 typedef Point<ll> P;
d41 vector<P> convexHull(vector<P> pts) {
d41   if (sz(pts) <= 1) return pts;
d41   sort(all(pts));
d41   vector<P> h(sz(pts)+1);
d41   int s = 0, t = 0;
d41   for (int it = 2; it--; s = --t, reverse(all(pts)))
d41     for (P p : pts) {
d41       while (t >= s + 2 && h[t-2].cross(h[t-1], p) <= 0) t--;
d41       h[t++] = p;
d41     }
d41   return {h.begin(), h.begin() + t - (t == 2 && h[0] == h[1])};
d41 }
```

HullDiameter.h

Description: Returns the two points with max distance on a convex hull (ccw, no duplicate/collinear points).
Time: $\mathcal{O}(n)$

"Point.h" d41d8c, 13 lines

```
d41 typedef Point<ll> P;
d41 array<P, 2> hullDiameter(vector<P> S) {
d41   int n = sz(S), j = n < 2 ? 0 : 1;
d41   pair<ll, array<P, 2>> res({0, {S[0], S[0]}});
d41   rep(i,0,j)
d41     for (;; j = (j + 1) % n) {
d41       res = max(res, {(S[i] - S[j]).dist2(), {S[i], S[j]}});
d41       if ((S[(j + 1) % n] - S[j]).cross(S[i + 1] - S[i]) >= 0)
d41         break;
d41     }
d41   return res.second;
```

```
d41 }
```

PointInsideHull.h

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

"Point.h", "sideOf.h", "OnSegment.h" d41d8c, 15 lines

```
d41 typedef Point<ll> P;

d41 bool inHull(const vector<P>& l, P p, bool strict = true) {
d41   int a = 1, b = sz(l) - 1, r = !strict;
d41   if (sz(l) < 3) return r && onSegment(l[0], l.back(), p);
d41   if (sideOf(l[0], l[a], l[b]) > 0) swap(a, b);
d41   if (sideOf(l[0], l[a], p) >= r || sideOf(l[0], l[b], p)<= -r)
d41     return false;
d41   while (abs(a - b) > 1) {
d41     int c = (a + b) / 2;
d41     (sideOf(l[0], l[c], p) > 0 ? b : a) = c;
d41   }
d41   return sgn(l[a].cross(l[b], p)) < r;
d41 }
```

LineHullIntersection.h

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

"Point.h" d41d8c, 40 lines

```
d41 #define cmp(i,j) sgn(dir.perp().cross(poly[(i)%n]-poly[(j)%n]))
d41 #define extr(i) cmp(i + 1, i) >= 0 && cmp(i, i - 1 + n) < 0
d41 template <class P> int extrVertex(vector<P>& poly, P dir) {
d41   int n = sz(poly), lo = 0, hi = n;
d41   if (extr(0)) return 0;
d41   while (lo + 1 < hi) {
d41     int m = (lo + hi) / 2;
d41     if (extr(m)) return m;
d41     int ls = cmp(lo + 1, lo), ms = cmp(m + 1, m);
d41     (ls < ms || (ls == ms && ls == cmp(lo, m)) ? hi : lo) = m;
d41   }
d41   return lo;
d41 }

d41 #define cmpL(i) sgn(a.cross(poly[i], b))
d41 template <class P>
d41 array<int, 2> lineHull(P a, P b, vector<P>& poly) {
d41   int endA = extrVertex(poly, (a - b).perp());
d41   int endB = extrVertex(poly, (b - a).perp());
d41   if (cmpL(endA) < 0 || cmpL(endB) > 0)
d41     return {-1, -1};
d41   array<int, 2> res;
d41   rep(i,0,2) {
d41     int lo = endB, hi = endA, n = sz(poly);
d41     while ((lo + 1) % n != hi) {
d41       int m = ((lo + hi + (lo < hi ? 0 : n)) / 2) % n;
d41       (cmpL(m) == cmpL(endB) ? lo : hi) = m;
d41     }
d41     res[i] = (lo + !cmpL(hi)) % n;
d41     swap(endA, endB);
d41   }
d41   if (res[0] == res[1]) return {res[0], -1};
d41   if (!cmpL(res[0]) && !cmpL(res[1]))
d41     switch ((res[0] - res[1] + sz(poly) + 1) % sz(poly)) {
d41       case 0: return {res[0], res[0]};
d41       case 2: return {res[1], res[1]};
d41     }
d41   return res;
d41 }
```

Minkowski.h

Description: Computes the Minkowski sum of two convex polygons. Polygons must be convex and given in CCW order. Returns the vertices of the Minkowski sum polygon in CCW order.
Time: $\mathcal{O}(n+m)$

"Point.h" d41d8c, 24 lines

```
d41 using P = Point<ll>;

d41 vector<P> minkowski(vector<P> p, vector<P> q) {
d41   auto fix = [](vector<P>& A) {
d41     int pos = 0;
d41     for (int i = 1; i < sz(A); i++) {
d41       if(A[i].y < A[pos].y || (A[i].y == A[pos].y && A[i].x < A[pos].x))
d41         pos = i;
d41     }
d41     rotate(A.begin(), A.begin() + pos, A.end());
d41     A.push_back(A[0]), A.push_back(A[1]);
d41   };
d41   fix(p), fix(q);
d41   vector<P> result;
d41   int i = 0, j = 0;
d41   while (i < sz(p) - 2 || j < sz(q) - 2) {
d41     result.push_back(p[i] + q[j]);
d41     auto cross = (p[i + 1] - p[i]).cross(q[j + 1] - q[j]);
d41     if (cross >= 0 && i < sz(p) - 2) i++;
d41     if (cross <= 0 && j < sz(q) - 2) j++;
d41   }
d41   return result;
d41 }
```

Extreme.h

Description: Finds an extreme vertex of a convex polygon according to a unimodal comparator. The comparator defines a total order along the polygon (given in CCW order).
Time: $\mathcal{O}(\log n)$

"Point.h" d41d8c, 26 lines

```
d41 using P = Point<ll>;
d41 int extreme(vector<P> &pol, const function<bool(P, P)>& cmp) {
d41   int n = pol.size();
d41   auto extr = [&](int i, bool& cur_dir) {
d41     cur_dir = cmp(pol[(i+1)%n], pol[i]);
d41     return !cur_dir and !cmp(pol[(i+n-1)%n], pol[i]);
d41   };
d41   bool last_dir, cur_dir;
d41   if (extr(0, last_dir)) return 0;
d41   int l = 0, r = n;
d41   while (l+1 < r) {
d41     int m = (l+r)/2;
d41     if (extr(m, cur_dir)) return m;
d41     bool rel_dir = cmp(pol[m], pol[l]);
d41     if ((!last_dir and cur_dir) or
d41         (last_dir == cur_dir and rel_dir == cur_dir)) {
d41       l = m;
d41       last_dir = cur_dir;
d41     } else r = m;
d41   }
d41   return l;
d41 }
d41 int max_dot(vector<P> &pol, P v) {
d41   return extreme([&](P p, P q) { return p.dot(v) > q.dot(v); });
d41 }
```

Tangents.h

Description: Finds the left and right tangent points from an external point p to a convex polygon given in CCW order. A tangent point is a vertex where the segment p->v touches the polygon without intersecting its interior, defining the limits of visibility from p. Returns the indices of the left and right tangent vertices.
Time: $\mathcal{O}(\log n)$

"Point.h", "Extreme.h" d41d8c, 11 lines

```
d41 using P = Point<ll>;

d41 bool ccw(P p, P q, P r) {
d41   return (q-p).cross(r-q) > 0;
d41 }
d41 pair<int, int> tangents(vector<P> &pol, P p) {
d41     auto L = [&](P q, P r) { return ccw(p, r, q); };
d41     auto R = [&](P q, P r) { return ccw(p, q, r); };
d41     return {extreme(pol, L), extreme(pol, R)};
d41 }
```

8.4 Misc. Point Set Problems

ClosestPair.h

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

"Point.h" d41d8c, 18 lines

```
d41 typedef Point<ll> P;
d41 pair<P, P> closest(vector<P> v) {
d41   assert(sz(v) > 1);
d41   set<P> S;
d41   sort(all(v), [](P a, P b) { return a.y < b.y; });
d41   pair<ll, pair<P, P>> ret{LLONG_MAX, {P(), P()}};
d41   int j = 0;
d41   for (P p : v) {
d41     P d{1 + (ll)sqrt(ret.first), 0};
d41     while (v[j].y <= p.y - d.x) S.erase(v[j++]);
d41     auto lo = S.lower_bound(p - d), hi = S.upper_bound(p + d);
d41     for (; lo != hi; ++lo)
d41       ret = min(ret, {(*lo - p).dist2(), {*lo, p}});
d41     S.insert(p);
d41   }
d41   return ret.second;
d41 }
```

ManhattanMST.h

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

"Point.h" d41d8c, 24 lines

```
d41 typedef Point<int> P;
d41 vector<array<int, 3>> manhattanMST(vector<P> ps) {
d41   vi id(sz(ps));
d41   iota(all(id), 0);
d41   vector<array<int, 3>> edges;
d41   rep(k,0,4) {
d41     sort(all(id), [&](int i, int j) {
d41           return (ps[i]-ps[j]).x < (ps[j]-ps[i]).y;});
d41     map<int, int> sweep;
```

```cpp
    for (int i : id) {
      for (auto it = sweep.lower_bound(-ps[i].y);
                it != sweep.end(); sweep.erase(it++)) {
        int j = it->second;
        P d = ps[i] - ps[j];
        if (d.y > d.x) break;
        edges.push_back({d.y + d.x, i, j});
      }
      sweep[-ps[i].y] = i;
    }
    for (P& p : ps) if (k & 1) p.x = -p.x; else swap(p.x, p.y);
  }
  return edges;
}
```

kdTree.h

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

"Point.h" d41d8c, 64 lines

```cpp
typedef long long T;
typedef Point<T> P;
const T INF = numeric_limits<T>::max();

bool on_x(const P& a, const P& b) { return a.x < b.x; }
bool on_y(const P& a, const P& b) { return a.y < b.y; }

struct Node {
  P pt; // if this is a leaf, the single point in it
  T x0 = INF, x1 = -INF, y0 = INF, y1 = -INF; // bounds
  Node *first = 0, *second = 0;

  T distance(const P& p) { // min squared distance to a point
    T x = (p.x < x0 ? x0 : p.x > x1 ? x1 : p.x);
    T y = (p.y < y0 ? y0 : p.y > y1 ? y1 : p.y);
    return (P(x,y) - p).dist2();
  }

  Node(vector<P>&& vp) : pt(vp[0]) {
    for (P p : vp) {
      x0 = min(x0, p.x); x1 = max(x1, p.x);
      y0 = min(y0, p.y); y1 = max(y1, p.y);
    }
    if (vp.size() > 1) {
      // split on x if width >= height (not ideal...)
      sort(all(vp), x1 - x0 >= y1 - y0 ? on_x : on_y);
      // divide by taking half the array for each child (not
      // best performance with many duplicates in the middle)
      int half = sz(vp)/2;
      first = new Node({vp.begin(), vp.begin() + half});
      second = new Node({vp.begin() + half, vp.end()});
    }
  }
};

struct KDTree {
  Node* root;
  KDTree(const vector<P>& vp) : root(new Node({all(vp)})) {}

  pair<T, P> search(Node *node, const P& p) {
    if (!node->first) {
      // uncomment if we should not find the point itself:
      // if (p == node->pt) return {INF, P()};
      return make_pair((p - node->pt).dist2(), node->pt);
    }

    Node *f = node->first, *s = node->second;
    T bfirst = f->distance(p), bsec = s->distance(p);
    if (bfirst > bsec) swap(bsec, bfirst), swap(f, s);

    // search closest side first, other side if needed
    auto best = search(f, p);
    if (bsec < best.first)
      best = min(best, search(s, p));
    return best;
  }

  // find nearest point to a point, and its squared distance
  // (requires an arbitrary operator< for Point)
  pair<T, P> nearest(const P& p) {
    return search(root, p);
  }
};
```

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 'circ'. Returns triangles in order {t[0][0], t[0][1], t[0][2], t[1][0], . . . }, all counter-clockwise.

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

"Point.h" d41d8c, 89 lines

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

struct Quad {
  Q rot, o; P p = arb; bool mark;
  P& F() { return r()->p; }
  Q& r() { return rot->rot; }
  Q prev() { return rot->o->rot; }
  Q next() { return r()->prev(); }
} *H;

bool circ(P p, P a, P b, P c) { // is p in the circumcircle?
  lll p2 = p.dist2(), A = a.dist2()-p2,
      B = b.dist2()-p2, C = c.dist2()-p2;
  return p.cross(a,b)*C + p.cross(b,c)*A + p.cross(c,a)*B > 0;
}
Q makeEdge(P orig, P dest) {
  Q r = H ? H : new Quad{new Quad{new Quad{new Quad{0}}}};
  H = r->o; r->r()->r() = r;
  rep(i,0,4) r = r->rot, r->p = arb, 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 = makeEdge(a->F(), b->p);
  splice(q, a->next());
  splice(q->r(), b);
  return q;
}

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

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

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

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

8.5 3D

PolyhedronVolume.h

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

d41d8c, 7 lines

```cpp
template<class V, class L>
double signedPolyVolume(const V& p, const L& trilist) {
  double v = 0;
  for (auto i : trilist) v += p[i.a].cross(p[i.b]).dot(p[i.c]);
  return v / 6;
}
```

Point3D.h

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

d41d8c, 33 lines

```
d41 template<class T> struct Point3D {
d41  typedef Point3D P;
d41  typedef const P& R;
d41  T x, y, z;
d41  explicit Point3D(T x=0, T y=0, T z=0) : x(x), y(y), z(z) {}
d41  bool operator<(R p) const {
d41    return tie(x, y, z) < tie(p.x, p.y, p.z); }
d41  bool operator==(R p) const {
d41    return tie(x, y, z) == tie(p.x, p.y, p.z); }
d41  P operator+(R p) const { return P(x+p.x, y+p.y, z+p.z); }
d41  P operator-(R p) const { return P(x-p.x, y-p.y, z-p.z); }
d41  P operator*(T d) const { return P(x*d, y*d, z*d); }
d41  P operator/(T d) const { return P(x/d, y/d, z/d); }
d41  T dot(R p) const { return x*p.x + y*p.y + z*p.z; }
d41  P cross(R p) const {
d41    return P(y*p.z - z*p.y, z*p.x - x*p.z, x*p.y - y*p.x);
d41  }
d41  T dist2() const { return x*x + y*y + z*z; }
d41  double dist() const { return sqrt((double)dist2()); }
     //Azimuthal angle (longitude) to x-axis in interval [-pi, pi]
d41  double phi() const { return atan2(y, x); }
     //Zenith angle (latitude) to the z-axis in interval [0, pi]
d41  double theta() const { return atan2(sqrt(x*x+y*y),z); }
d41  P unit() const { return *this/(T)dist(); } //makes dist()=1
     //returns unit vector normal to *this and p
d41  P normal(P p) const { return cross(p).unit(); }
     //returns point rotated 'angle' radians ccw around axis
d41  P rotate(double angle, P axis) const {
d41    double s = sin(angle), c = cos(angle); P u = axis.unit();
d41    return u*dot(u)*(1-c) + (*this)*c - cross(u)*s;
d41  }
d41 };
```

3dHull.h

Description: Computes all faces of the 3-dimension hull of a point set. *No four points must be coplanar*, or else random results will be returned. All faces will point outwards.

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

"Point3D.h" d41d8c, 50 lines

```
d41 typedef Point3D<double> P3;

d41 struct PR {
d41  void ins(int x) { (a == -1 ? a : b) = x; }
d41  void rem(int x) { (a == x ? a : b) = -1; }
d41  int cnt() { return (a != -1) + (b != -1); }
d41  int a, b;
d41 };

d41 struct F { P3 q; int a, b, c; };

d41 vector<F> hull3d(const vector<P3>& A) {
d41  assert(sz(A) >= 4);
d41  vector<vector<PR>> E(sz(A), vector<PR>(sz(A), {-1, -1}));
d41 #define E(x,y) E[f.x][f.y]
d41  vector<F> FS;
d41  auto mf = [&](int i, int j, int k, int l) {
d41    P3 q = (A[j] - A[i]).cross((A[k] - A[i]));
d41    if (q.dot(A[l]) > q.dot(A[i]))
d41      q = q * -1;
d41    F f{q, i, j, k};
d41    E(a,b).ins(k); E(a,c).ins(j); E(b,c).ins(i);
d41    FS.push_back(f);
d41  };
d41  rep(i,0,4) rep(j,i+1,4) rep(k,j+1,4)
d41    mf(i, j, k, 6 - i - j - k);

d41  rep(i,4,sz(A)) {
d41    rep(j,0,sz(FS)) {
d41      F f = FS[j];
d41      if(f.q.dot(A[i]) > f.q.dot(A[f.a])) {
d41        E(a,b).rem(f.c);
d41        E(a,c).rem(f.b);
d41        E(b,c).rem(f.a);
d41        swap(FS[j--], FS.back());
d41        FS.pop_back();
d41      }
d41    }
d41    int nw = sz(FS);
d41    rep(j,0,nw) {
d41      F f = FS[j];
d41 #define C(a, b, c) if (E(a,b).cnt() != 2) mf(f.a, f.b, i, f.c);
d41      C(a, b, c); C(a, c, b); C(b, c, a);
d41    }
d41  }
d41  for (F& it : FS) if ((A[it.b] - A[it.a]).cross(
d41    A[it.c] - A[it.a]).dot(it.q) <= 0) swap(it.c, it.b);
d41  return FS;
d41 };
```

sphericalDistance.h

Description: Returns the shortest distance on the sphere with radius radius between the points with azimuthal angles (longitude) f1 (ϕ_1) and f2 (ϕ_2) from x axis and zenith angles (latitude) t1 (θ_1) and t2 (θ_2) from z axis (0 = north pole). All angles measured in radians. The algorithm starts by converting the spherical coordinates to cartesian coordinates so if that is what you have you can use only the two last rows. dx*radius is then the difference between the two points in the x direction and d*radius is the total distance between the points.

d41d8c, 9 lines

```
d41 double sphericalDistance(double f1, double t1,
d41    double f2, double t2, double radius) {
d41  double dx = sin(t2)*cos(f2) - sin(t1)*cos(f1);
d41  double dy = sin(t2)*sin(f2) - sin(t1)*sin(f1);
d41  double dz = cos(t2) - cos(t1);
d41  double d = sqrt(dx*dx + dy*dy + dz*dz);
d41  return radius*2*asin(d/2);
d41 }
```

Strings (9)

AhoCorasick.h

d41d8c, 46 lines

```
d41 int trie[ms][sigma], fail[ms], terminal[ms], superfail[ms];
d41 bool present[ms];
d41 int z = 1;

d41 int val(char c) { return c - 'a'; }

d41 void add(string& p) {
d41     int cur = 0;
d41     for (int i = 0; i < (int)p.size(); i++) {
d41         int& nxt = trie[cur][val(p[i])];
d41         if (nxt == 0) nxt = z++;
d41         cur = nxt;
d41     }
d41     present[cur] = true;
d41     terminal[cur]++;
d41 }

d41 void build() {
d41     queue<int> q;
d41     for (q.push(0); !q.empty(); q.pop()) {
d41         int on = q.front();
d41         for (int i = 0; i < sigma; i++) {
d41             int& to = trie[on][i];
d41             int f = (on == 0 ? 0 : trie[fail[on]][i]);
d41             int sf = (present[f] ? f : superfail[f]);
d41             if (!to) {
d41                 to = f;
d41             }
d41             else {
d41                 fail[to] = f;
d41                 superfail[to] = sf;
                    // merge infos (ex: terminal[to] += terminal[f])
d41                 q.push(to);
d41             }
d41         }
d41     }
d41 }

d41 void search(string& s) {
d41     int cur = 0;
d41     for (char c : s) {
d41         cur = trie[cur][val(c)];
            // process infos on current node (ex: ocurrences += terminal[cur])
d41     }
d41 }
```

Hash.h

Description: C can also be random, operator is $[l, r]$

d41d8c, 28 lines

```
d41 using ull = uint64_t;
d41 struct H {
d41     ull x; H(ull x = 0) : x(x) {}
d41     H operator+(H o) { return x + o.x + (x + o.x < x); }
d41     H operator-(H o) { return *this + ~o.x; }
d41     H operator*(H o) {
d41         auto m = (__uint128_t)x * o.x;
d41         return H((ull)m) + (ull)(m >> 64);
d41     }
d41     ull get() const { return x + !~x; }
d41     bool operator==(H o) const{ return get() == o.get();}
d41     bool operator<(H o) const{ return get() < o.get();}
d41 };
d41 static const H C = (ll)1e11 + 3;
d41 struct Hash {
d41     vector<H> h, pw;
d41     Hash(string& str) : h(str.size()), pw(str.size()) {
d41         pw[0] = 1, h[0] = str[0];
d41         for (int i = 1; i < str.size(); i++) {
d41             h[i] = h[i - 1] * C + str[i];
d41             pw[i] = pw[i - 1] * C;
d41         }
d41     }
d41     H operator()(int l, int r) {
d41         return h[r] - (l ? h[l - 1] * pw[r - l + 1] : 0);
d41     }
d41 };
```

KMP.h

Description: pi[x] computes the length of the longest prefix of s that ends at x, other than s[0...x] itself (abacaba -> 0010123).

d41d8c, 10 lines

```cpp
vector<int> pi(const string& s) {
    vector<int> p(sz(s));
    for(int i = 1; i < sz(s); i++) {
        int g = p[i-1];
        while (g && s[i] != s[g]) g = p[g-1];
        p[i] = g + (s[i] == s[g]);
    }
    return p;
}
```

KmpAutomaton.h

Description: $go[i][j]$ = length of the longest prefix of s that is a suffix of $s[0..i]$ followed by the letter j (i.e., the next matched prefix length if, at state i, we read letter j).

d41d8c, 17 lines

```cpp
int go[ms][sigma];
int val(char c) { return c - 'a'; }
void automaton(string& s) {
    for (int i = 0; i < sigma; i++)
        go[0][i] = (i == val(s[0]));

    for (int i = 1, bdr = 0; i <= (int)s.size(); i++) {
        for (int j = 0; j < sigma; j++) {
            go[i][j] = go[bdr][j];
        }
        if (i < (int)s.size()) {
            go[i][val(s[i])] = i + 1;
            bdr = go[bdr][val(s[i])];
        }
    }
}
```

Manacher.h

Description: $p[0][i+1]$ is the length of matches of even length palindrome, starting from $[i, i+1]$.
$p[1][i]$ is the length of matches of odd length palindrome, starting from $[i, i]$.
(abaxx -> p[0] = 00001)
(abaxx -> p[1] = 01000)

d41d8c, 17 lines

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

MinRotation.h

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

d41d8c, 14 lines

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

SuffixArray.h

Description: $lcp[i]$ is the length of the longest common prefix between the suffixes $s[sa[i]..n-1]$ and $s[sa[i-1]..n-1]$.
If we concatenate multiple strings using separator characters, the separator that appears furthest to the right must be the smallest character in the alphabet.

d41d8c, 31 lines

```cpp
struct SuffixArray {
 vector<int> sa, lcp;
 SuffixArray(string s, int lim=256) {
   s.push_back('$');
   int n = sz(s), k = 0, a, b;
   vector<int> x(all(s)), y(n), ws(max(n, lim));
   sa = lcp = y, iota(all(sa), 0);
   for(int j = 0, p = 0; p < n; j= max(1, j*2), lim = p) {
     p = j, iota(all(y), n - j);
     for(int i=0; i<n; i++){
       if (sa[i] >= j) y[p++] = sa[i] - j;
     }
     fill(all(ws), 0);
     for(int i=0; i<n; i++) ws[x[i]]++;
     for(int i=1; i<lim; i++) ws[i] += ws[i - 1];
     for (int i = n; i--;) sa[--ws[x[y[i]]]] = y[i];
     swap(x, y), p = 1, x[sa[0]] = 0;
     for(int i=1; i<n; i++){
       a = sa[i - 1], b = sa[i];
       x[b] = p-1;
       if(y[a] != y[b] || y[a+j] != y[b+j]) x[b] = p++;
     }
   }
   for (int i = 0, j; i < n - 1; lcp[x[i++]] = k)
     for (k && k--, j = sa[x[i] - 1];
         s[i + k] == s[j + k]; k++);
   sa = vector<int>(sa.begin() + 1, sa.end());
   lcp = vector<int>(lcp.begin() + 1, lcp.end());
 }
};
```

Zfunc.h

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

d41d8c, 13 lines

```cpp
vector<int> ZFunc(const string& s) {
    int n = sz(s), a = 0, b = 0;
    vector<int> z(n, 0);
    if (!z.empty()) z[0] = 0;
    for (int i = 1; i < n; i++) {
        int end = i;
        if (i < b) end = min(i + z[i - a], b);
        while (end < n && s[end] == s[end - i]) ++end;
        z[i] = end - i; if (end > b) a = i, b = end;
    }
    return z;
}
```

Various (10)

10.1 Misc. algorithms

Dates.h

Description: dateToInt converts Gregorian date to integer (Julian day number). intToDate converts integer (Julian day number) to Gregorian date: month/day/year. intToDay converts Julian day number to day of the week

d41d8c, 23 lines

```cpp
string day[] = { "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun" };
int dateToInt(int m, int d, int y) {
    return
        1461 * (y + 4800 + (m - 14) / 12) / 4 +
        367 * (m - 2 - (m - 14) / 12 * 12) / 12 -
        3 * ((y + 4900 + (m - 14) / 12) / 100) / 4 +
        d - 32075;
}
void intToDate(int jd, int& m, int& d, int& y) {
    int x, n, i, j;
    x = jd + 68569;
    n = 4 * x / 146097;
    x -= (146097 * n + 3) / 4;
    i = (4000 * (x + 1)) / 1461001;
    x -= 1461 * i / 4 - 31;
    j = 80 * x / 2447;
    d = x - 2447 * j / 80;
    x = j / 11;
    m = j + 2 - 12 * x;
    y = 100 * (n - 49) + i + x;
}
string intToDay(int jd) { return day[jd % 7]; }
```

MultisetHash.h

d41d8c, 8 lines

```cpp
ull hashify(ull sum) {
    sum += FIXED_RANDOM;
    sum += 0x9e3779b97f4a7c15;
    sum = (sum ^ (sum >> 30)) * 0xbf58476d1ce4e5b9;
    sum = (sum ^ (sum >> 27)) * 0x94d049bb133111eb;
    return sum ^ (sum >> 31);
}
```

Rand.h

d41d8c, 8 lines

```cpp
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
// _64

int uniform(int l, int r) { // [l, r]
    uniform_int_distribution<int> uid(l, r);
    return uid(rng);
}
```

10.2 Dynamic programming

KnuthDP.h

Description: When doing DP on intervals: $dp[i][j] = \min_{i<k<j}(dp[i][k] + dp[k][j]) + f(i,j)$, where the (minimal) optimal k increases with both i and j. This is known as Knuth DP. Sufficient criteria for this are if $f(b,c) \le f(a,d)$ and $f(a,c) + f(b,d) \le f(a,d) + f(b,c)$ for all $a \le b \le c \le d$. Another sufficient criteria is: $opt[i][j-1] \le opt[i][j] \le opt[i+1][j]$
Time: $\mathcal{O}(N^2)$

d41d8c, 22 lines

```cpp
ll knuth(){
    memset(opt, -1, sizeof opt);
    for(int i=n-1; i>=0; i--){
        dp[i][i] = 0; // base case
        opt[i][i] = i;
        for(int j=i+1; j<n; j++){
```

```
d41             int optL = (!j ? 0 : opt[i][j-1]);
d41             int optR = (~opt[i+1][j] ? opt[i+1][j] : n-1);
d41             ll cst = cost(i, j);
d41             dp[i][j] = INF;
d41             optL = max(i, optL), optR = min(j-1, optR);
d41             for(int k=optL; k<=optR; k++){
d41                 ll now = dp[i][k] + dp[k+1][j] + cst;
d41                 if(now <= dp[i][j]){
d41                     dp[i][j] = now;
d41                     opt[i][j] = k;
d41                 }
d41             }
d41         }
d41     }
d41 }
```

DivideAndConquerDP.h

Description: Divide and Conquer DP maintaining cost, can be used when $opt[i][j] \leq opt[i][j+1]$. In this code everything is 1-based. Memory can be optmized by keeping only the last row

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

d41d8c, 42 lines

```
d41 void add(int idx) {}
d41 void rem(int idx) {}

d41 void deC(int i, int l, int r, int optL, int optR) {
d41     if (l > r) return;
d41     int j = (l + r) / 2;
d41     for (int k = r; k > j; k--) rem(k);
d41     int opt = optL;
d41     for (int k = optL; k <= min(optR, j); k++) {
            // cost = cost[k, j]
d41         int val = dp[i - 1][k - 1] + cost;
d41         if (val < dp[i][j]) {
d41             dp[i][j] = val;
d41             opt = k;
d41         }
d41         rem(k);
d41     }
d41     for (int k = min(optR, j); k >= optL; k--) add(k);
d41     rem(j);
d41     deC(i, l, j - 1, optL, opt);

d41     for (int k = j; k <= r; k++) add(k);
d41     for (int k = optL; k < opt; k++) rem(k);
d41     deC(i, j + 1, r, opt, optR);

d41     for (int k = optL; k < opt; k++) add(k);
d41 }

d41 int solve(int N, int M) { // 1-based
d41     for (int i = 0; i <= M; i++) {
d41         for (int j = 0; j <= N; j++){
d41             dp[i][j] = inf; // base case
d41         }
d41     }
d41     cost = 0; // neutral value
d41     for (int i = 1; i <= N; i++) add(i);
d41     for (int i = 1; i <= M; i++) {
d41         deC(i, 1, N, 1, N);
d41     }
d41     return dp[M][N];
d41 }
```