// // primemath.cpp // //

// Miller-Rabin prime test and Rho-Pollard factorization.

// Last revision: Middle of 2018.

#include <algorithm>

#include <iostream>

#include <ext/numeric>

#include <cassert>

#include <list>

#define gcd \_\_gcd

**using** **namespace** std**;**

template**<**typename T**>**

struct mod\_multiplies **:** multiplies**<**T**>**

**{**

T m**;** mod\_multiplies**(**T m**)** **:** m**(**m**)** **{}**

T **operator()(**T a**,** T b**)** const

**{**

T r **=** 0**;**

**while(**a**)**

**{**

**if(**a **%** 2**)**

r **+=** b**,** r **%=** m**;**

a **/=** 2**;**

b **\*=** 2**,** b **%=** m**;**

**}**

**return** r**;**

**}**

**};**

**using** \_\_gnu\_cxx**::**power**;**

template**<**typename T**>**

T modpower**(**T a**,** T b**,** T m**)** **{** **return** power**(**a**,** b**,** mod\_multiplies**<**T**>(**m**));** **}**

// 0: 2^32 - {2, 7, 61} (3)

// 1: 3.8 \* 10^18 - {2, 3, 5, 7, 11, 13, 17, 19, 23} (9)

// 2: 2^64 - {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37} (12)

const vector**<**uint64\_t**>** MR\_PRIMES**[**3**]** **=** **{**

**{**2**,** 7**,** 61**},**

**{**2**,** 3**,** 5**,** 7**,** 11**,** 13**,** 17**,** 19**,** 23**},**

**{**2**,** 3**,** 5**,** 7**,** 11**,** 13**,** 17**,** 19**,** 23**,** 29**,** 31**,** 37**}**

**};**

bool miller\_rabin\_isprime**(**uint64\_t n**,** const vector**<**uint64\_t**>&** W**)**

**{**

**if(**n **<=** 1**)** **return** **false;** **if(**n **<=** 3**)** **return** **true;**

**if(**n **%** 2 **==** 0**)** **return** **false;**

mod\_multiplies**<**uint64\_t**>** M**(**n**);**

uint64\_t d **=** n **-** 1**;**

uint32\_t r **=** 0**;**

**while(**d **%** 2 **==** 0**)** d **/=** 2**,** r**++;**

**for(**uint64\_t a **:** W**)**

**{**

**if(**a **+** 2 **>** n**)** **continue;**

uint64\_t x **=** modpower**(**a**,** d**,** n**);**

**if(**x **==** 1 **or** x **==** n **–** 1**) continue;**

bool comp **=** **true;**

**for(**uint32\_t i **=** 0**;** i **<** r **-** 1**;** i**++)**

**{**

x **=** M**(**x**,** x**);**

**if(**x **==** 1**)**

**return** **false;**

**if(**x **==** n **-** 1**)**

**{** comp **=** **false; break;** **}**

**}**

**if(**comp**)**

**return** **false;**

**}**

**return** **true;**

**}**

bool miller\_rabin\_isprime**(**uint64\_t n**)**

**{**

**if(**n **<** **(**1llu **<<** 32**))**

**return** miller\_rabin\_isprime**(**n**,** MR\_PRIMES**[**0**]);**

**else** **if(**n **<** **((**1llu**<<**61**)+(**1llu**<<**60**)+(**1llu**<<**59**)-(**1llu**<<**58**)))**

**return** miller\_rabin\_isprime**(**n**,** MR\_PRIMES**[**1**]);**

**else**

**return** miller\_rabin\_isprime**(**n**,** MR\_PRIMES**[**2**]);**

**}**

template**<**typename T**>**

bool sqrt\_isprime**(**T n**)**

**{**

**if(**n **<=** 1**)** **return** **false;** **if(**n **<=** 3**)** **return** **true;**

**for(**T i **=** 2**;** i **\*** i **<=** n**;** i**++)**

**if(**n **%** i **==** 0**)**

**return** **false;**

**return** **true;**

**}**

uint64\_t rho\_pollard\_get\_factor**(**uint64\_t n**)**

**{**

**if(**n **%** 2 **==** 0**)**

**return** 2**;**

**if(**miller\_rabin\_isprime**(**n**))**

**return** n**;**

mod\_multiplies**<**uint64\_t**>** M**(**n**);**

**for(**uint64\_t c **=** 1**;** **true;** c**++)**

**{**

uint64\_t x **=** 2**,** y **=** 2**,** d **=** 1**;**

**while(**d **==** 1**)**

**{**

x **=** **(**M**(**x**,** x**)** **+** c**)** **%** n**;**

y **=** **(**M**(**y**,** y**)** **+** c**)** **%** n**;** y **=** **(**M**(**y**,** y**)** **+** c**)** **%** n**;**

d **=** gcd**(**x**>**y **?** x**-**y **:** y**-**x**,** n**);**

**}**

**if(**d **!=** n**)**

**return** d**;**

**}**

**return** 1**;**

**}**

list**<**uint64\_t**>** rho\_pollard\_factorize**(**uint64\_t n**)**

**{**

**if(**n **==** 1**)**

**return** **{};**

uint64\_t d **=** rho\_pollard\_get\_factor**(**n**);**

list**<**uint64\_t**>** result**;**

**if(**miller\_rabin\_isprime**(**d**))**

result**.**merge**({**d**});**

**else**

result**.**merge**(**rho\_pollard\_factorize**(**d**));**

**if(**miller\_rabin\_isprime**(**n**/**d**))**

result**.**merge**({**n**/**d**});**

**else**

result**.**merge**(**rho\_pollard\_factorize**(**n**/**d**));**

**return** result**;**

**}**

uint32\_t pi\_prime\_count**(**uint64\_t n**)**

**{**

uint32\_t r **=** 0**;**

**for(**uint64\_t i **=** 2**;** i **<=** n**;** i**++)**

r **+=** miller\_rabin\_isprime**(**i**);**

**return** r**;**

**}**

#undef gcd

// // fenwick.cpp // //

// Fenwick tree. Calculates prefix sums and allows for point changes.

// Compact, 0-based implementation.

// lower\_bound based on https://codeforces.com/blog/entry/61364

// Last revision: October 2018

#pragma once

#include <cstddef>

**using** std**::**size\_t**;**

template**<**typename T**>**

constexpr inline T lsb**(**T x**)** **{** **return** x **&** **(-**x**);** **}**

template**<**typename T**>**

struct fenwick\_tree

**{**

size\_t n**;**

T**\*** F**;**

fenwick\_tree**(**size\_t \_n**)** **{** n **=** \_n**+**1**;** F **=** **new** T**[**n**+**1**];** fill**(**F**,** F **+** **(**n**+**1**),** 0**);** **}**

**~**fenwick\_tree**()** **{** **delete[]** F**;** **}**

T get\_prefix**(**size\_t p**)** const // Sum in [0, p)

**{** T r **=** 0**;** p**++;** **while(**p**)** r **+=** F**[**p**],** p **-=** lsb**(**p**);** **return** r**;** **}**

void delta**(**size\_t p**,** T v**)**

**{** p **+=** 2**;** **while(**p **<=** n**)** F**[**p**]** **+=** v**,** p **+=** lsb**(**p**);** **}**

T get**(**size\_t a**,** size\_t b**)** const // Get on interval [a, b]

**{** **return** get\_prefix**(**b**+**1**)** **-** get\_prefix**(**a**);** **}**

T get**(**size\_t p**)** const // Get on point [p]

**{** **return** get**(**p**,** p**);** **}**

void set**(**size\_t p**,** T v**)**

**{** **return** delta**(**p**,** v **-** get**(**p**));** **}**

size\_t lower\_bound**(**T v**)**

**{**

T s **=** 0**;** size\_t p **=** 0**;**

**for(**size\_t i **=** **(**32 **-** \_\_builtin\_clz**(**n**));** i **-->** 0**;** **)** // \log2(n)/+1

**{**

**if(**p **+** **(**1u **<<** i**)** **<** n **and** s **+** F**[**p **+** **(**1u **<<** i**)]** **<** v**)**

s **+=** F**[**p **+** **(**1u **<<** i**)],** p **+=** 1u **<<** i**;**

**}**

**return** p**;**

**}**

size\_t find\_by\_order**(**size\_t i**)** **{** **return** lower\_bound**(**i**+**1**)** **-** 1**;** **}**

**};**

// // manacher.cpp // //

// Manacher's palindrome-finding algorithm.

// Returns a vector of both odd and even palindromes

// (in a spliced manner: odd/even/odd/even...)

// Last revision: Middle of 2018

#pragma once

#include <cstdint>

#include <string>

#include <vector>

**using** std**::**vector**;** **using** std**::**string**;**

**using** std**::**uint32\_t**;** **using** std**::**iterator\_traits**;**

template**<**typename Iterator**,** class T **=** typename iterator\_traits**<**Iterator**>::**value\_type **>**

vector**<**uint32\_t**>** manacher**(**Iterator first**,** Iterator last**,** T leaf **=** '$'**)**

**{**

vector**<**T**>** A**(**distance**(**first**,** last**)\***2 **+** 1**);**

**for(**auto it **=** A**.**begin**();** first **!=** last**;** first**++)**

**{**

**\***it**++** **=** leaf**;**

**\***it**++** **=** **\***first**;**

**}**

A**.**back**()** **=** '$'**;**

vector**<**uint32\_t**>** P**(**A**.**size**(),** 0**);**

uint32\_t right **=** 0**,** center **=** 0**;**

**for(**uint32\_t i **=** 1**;** i **<** A**.**size**();** i**++)**

**{**

uint32\_t mirror **=** 2**\***center **-** i**;**

**if(**i **+** P**[**mirror**]** **<=** min**(**right**,** A**.**size**()** **-** 1**))**

P**[**i**]** **=** P**[**mirror**];**

**while(**i **>=** P**[**i**]+**1 **and** i**+**P**[**i**]+**1 **<** A**.**size**()**

**and** A**[**i**-**P**[**i**]-**1**]** **==** A**[**i**+**P**[**i**]+**1**])**

P**[**i**]++;**

**if(**i **+** P**[**i**]** **>** right**)**

center **=** i**,** right **=** center **+** P**[**i**];**

**}**

**for(**uint32\_t i **=** 0**;** i **<** A**.**size**();** i**++)**

P**[**i**]** **=** P**[**i**]/**2 **+** i**%**2**;**

**return** P**;**

**}**

// // kmp.txt // //

/\*

// Preprocessing (Prefixosuffixes)

input:

character S[K] // pattern

integer T[K]; T[0] = 0

for i in 1..K-1:

j = T[i - 1]

while j > 0 and S[i] != S[j]:

j = T[j - 1]

if S[i] == S[j]:

j += 1

T[i] = j

// Search

input:

character M[N] // text

j = 0 // current character in pattern

for i in 0..N-1: // current character in text

while j > 0 and M[i] != S[j]:

j = T[j - 1]

if M[i] == S[j]:

j += 1

if j == K: // match

yield i + 2 - K // begin index

j = T[j - 1] // next matching with overlap

// without overlap: j = 0

\*/

// // rolling\_hash.cpp // //

// Rolling hashes.

// Last revision: Middle of 2018

#pragma once

#include <cstddef>

#include <cstdint>

**using** std**::**size\_t**;**

**using** std**::**uint64\_t**;**

// Mods: 1e9+7, 1e9+11, 1e9+21, 1e9+33, 2^31 - 1

template**<**typename T**,** T MOD**,** T BASE**,** size\_t N**>**

struct rolling\_hash

**{**

T H**[**N**];**

T base\_pow**[**N**];**

size\_t n**;**

template**<**typename Iterator**>**

rolling\_hash**(**Iterator begin**,** Iterator end**,** T fix **=** 0**)**

**{**

n **=** distance**(**begin**,** end**);**

Iterator it **=** begin**;**

**for(**size\_t i **=** 0**;** i **<** n**;** i**++)**

**{**

**if(**i **>** 0**)**

H**[**i**]** **=** H**[**i**-**1**]** **\*** BASE**,** H**[**i**]** **%=** MOD**;**

H**[**i**]** **+=** T**(\***it**)** **+** fix**;**

H**[**i**]** **%=** MOD**;**

**if(**i **>** 0**)**

base\_pow**[**i**]** **=** **(**base\_pow**[**i**-**1**]** **\*** BASE**)** **%** MOD**;**

**else**

base\_pow**[**i**]** **=** 1**;**

it**++;**

**}**

**}**

const T**&** **operator[]** **(**size\_t i**)** const **{** **return** H**[**i**];** **}**

T **operator()** **(**size\_t i**,** size\_t j**)** const

**{**

**if(**i **==** 0**)**

**return** H**[**j**];**

**else**

**return** **(**2**\***MOD **+** H**[**j**]** **-** **((**H**[**i**-**1**]\***base\_pow**[**j **-** i **+** 1**])** **%** MOD**))** **%** MOD**;**

**}**

**};**

// Mod 2^31 - 1 (should work for any P = 2^K - 1, P is prime)

// the conditional can be deleted for additional speed

template**<**uint64\_t K **=** 31**>**

uint64\_t mersenne\_mod**(**uint64\_t x**)**

**{**

const uint64\_t P **=** **(**1 **<<** K**)** **-** 1**;**

x **=** **(**x **>>** K**)** **+** **(**x **&** P**);**

x **=** **(**x **>>** K**)** **+** **(**x **&** P**);**

**return** x **==** P **?** 0 **:** x**;**

**}**

// // extended\_euclidean.cpp // //

// Extended Euclidean algorithm

// Finds solutions to ax + by = p\*gcd(a, b)

// Bezout's identity:

// x' = x + k (b / gcd(a, b))

// y' = y - k (a / gcd(a, b))

// ax' + by' = p\*gcd(a, b)

// Last revision: 2017/2018

#pragma once

#include <algorithm>

#define gcd \_\_gcd

template**<**typename T**>**

struct gcdext\_result **{** T d**,** x**,** y**;** **};**

template**<**typename T**>**

gcdext\_result**<**T**>** gcdext**(**T a**,** T b**)**

**{**

**if(**a **==** 0**)**

**return** **{**b**,** 0**,** 1**};**

auto next **=** gcdext**(**b **%** a**,** a**);**

**return** **{**next**.**d**,** next**.**y **-** **(**b**/**a**)\***next**.**x**,** next**.**x**};**

**}**

#undef gcd

// // geometry.cpp // //

#include <bits/stdc++.h>

**using** **namespace** std**;**

template**<**typename T**>**

**using** point\_t **=** complex**<**T**>;**

#define X real()

#define Y imag()

// Dot product

template**<**typename T**>**

T cross**(**const point\_t**<**T**>&** a**,** const point\_t**<**T**>&** b**)**

**{**

**return** a**.**X **\*** b**.**Y **-** a**.**Y **\*** b**.**X**;**

**}**

int64\_t sign**(**int64\_t x**)** **{** **return** **(**x **>=** 0**)** **-** **(**x **<=** 0**);** **}**

// orientation of triplet [triangle] (a, b, c)

// = 0 if colinear

// > 0 clockwise

// < 0 counterclockwise

template**<**typename T**>**

T ori**(**const point\_t**<**T**>&** a**,** const point\_t**<**T**>&** b**,** const point\_t**<**T**>&** c**)**

**{**

**return** sign**((**c**.**Y**-**a**.**Y**)** **\*** **(**b**.**X**-**a**.**X**)** **-** **(**b**.**Y**-**a**.**Y**)** **\*** **(**c**.**X**-**a**.**X**));**

**}**

// on segment

// point p is on segment |ab|

template**<**typename T**>**

bool ons**(**const point\_t**<**T**>&** a**,** const point\_t**<**T**>&** b**,** const point\_t**<**T**>&** p**)**

**{**

**return** **(**b**.**X **<=** max**(**a**.**X**,** p**.**X**)** **and** b**.**X **>=** min**(**a**.**X**,** p**.**X**)** **and**

b**.**Y **<=** max**(**a**.**Y**,** p**.**Y**)** **and** b**.**Y **>=** min**(**a**.**Y**,** p**.**Y**));**

**}**

template**<**typename T**>**

bool intersects**(**const point\_t**<**T**>&** a**,** const point\_t**<**T**>&** b**,**

const point\_t**<**T**>&** c**,** const point\_t**<**T**>&** d**)**

**{**

int64\_t o1 **=** ori**(**a**,** b**,** c**),** o2 **=** ori**(**a**,** b**,** d**),**

o3 **=** ori**(**c**,** d**,** a**),** o4 **=** ori**(**c**,** d**,** b**);**

**if(**o1 **!=** o2 **and** o3 **!=** o4**)**

**return** **true;**

**else** **if((**o1 **==** 0 **and** ons**(**a**,** c**,** b**))** **or**

**(**o2 **==** 0 **and** ons**(**a**,** d**,** b**))** **or**

**(**o3 **==** 0 **and** ons**(**c**,** a**,** d**))** **or**

**(**o4 **==** 0 **and** ons**(**c**,** b**,** d**)))**

**return** **true;**

**else**

**return** **false;**

**}**

// Double, oriented area

// (+ccw, -cw)

// use abs(polygon.area)/2 for actual value

template**<**typename T**>**

T area**(**const vector**<**point\_t**<**T**>>&** polygon**)**

**{**

T result **=** 0**;**

size\_t prev\_i **=** polygon**.**size**()** **-** 1**,** next\_i **=** 1**;**

**for(**size\_t i **=** 0**;** i **<** polygon**.**size**();** i**++)**

**{**

result **+=** polygon**[**i**].**X **\*** **(**polygon**[**prev\_i**].**Y **-** polygon**[**next\_i**].**Y**);**

prev\_i **=** i**;**

next\_i **=** i**+**1**;**

**if(**next\_i **==** polygon**.**size**())** next\_i **=** 0**;**

**}**

**return** result**;**

**}**

// Convex hull

// Andrew's monotone chain algorithm

template**<**typename T**,** bool INCLUDE\_COLINEAR **=** **true>**

vector**<**point\_t**<**T**>>** convex\_hull**(**vector**<**point\_t**<**T**>>&** points**)**

**{**

sort**(**

points**.**begin**(),** points**.**end**(),**

**[](**const point\_t**<**T**>&** lhs**,** const point\_t**<**T**>&** rhs**)** **{**

**return** lhs**.**X **<** rhs**.**X **or** **(**lhs**.**X **==** rhs**.**X **and** lhs**.**Y **<** rhs**.**Y**);**

**});**

vector**<**point\_t**<**T**>>** hull**;** hull**.**reserve**(**points**.**size**());**

**for(**uint32\_t phase **=** 0**;** phase **<** 2**;** phase**++)**

**{**

size\_t start **=** hull**.**size**();**

**for(**const point\_t**<**T**>&** point **:** points**)**

**{**

**while(**

hull**.**size**()** **>=** start **+** 2 **and**

**(**INCLUDE\_COLINEAR

**?** **(**ori**(**point**,** hull**[**hull**.**size**()-**1**],** hull**[**hull**.**size**()-**2**])** **<=** 0**)**

**:** **(**ori**(**point**,** hull**[**hull**.**size**()-**1**],** hull**[**hull**.**size**()-**2**])** **<** 0**))**

**)** hull**.**pop\_back**();**

hull**.**push\_back**(**point**);**

**}**

hull**.**pop\_back**();**

reverse**(**points**.**begin**(),** points**.**end**());**

**}**

**if(**hull**.**size**()** **==** 2 **and** hull**[**0**]** **==** hull**[**1**])** hull**.**pop\_back**();**

**return** hull**;**

**}**

// // randutil.cpp // //

// Randomization utility

// random\_device{}() - randomizes seed. this is constant on some compilers

// for some reason. You can use any valid seed.

// uniform\_int\_distribution<T>{a, b} - prepares a caller for the generator.

// Works on any range contained in numeric\_limits<T>::min(), ~::max()

// Use gen.seed() before usage, with some random integer

// (time(0) should be enough)

#pragma once

#include <random>

std**::**mt19937 gen**{**std**::**random\_device**{}()};**

template**<**typename T**>**

T randint**(**T a**,** T b**)**

**{** **return** std**::**uniform\_int\_distribution**<**T**>{**a**,** b**}(**gen**);** **}**

// // chronohelper.cpp // //

#pragma once

#include <chrono>

// https://stackoverflow.com/questions/21856025/getting-an-accurate-execution-time-in-c-micro-seconds

// Get time stamp in milliseconds.

uint64\_t millis**()**

**{**

uint64\_t ms **=** std**::**chrono**::**duration\_cast**<**std**::**chrono**::**milliseconds**>(**std**::**chrono**::**high\_resolution\_clock**::**

now**().**time\_since\_epoch**()).**count**();**

**return** ms**;**

**}**

// Get time stamp in microseconds.

uint64\_t micros**()**

**{**

uint64\_t us **=** std**::**chrono**::**duration\_cast**<**std**::**chrono**::**microseconds**>(**std**::**chrono**::**high\_resolution\_clock**::**

now**().**time\_since\_epoch**()).**count**();**

**return** us**;**

**}**

// Get time stamp in nanoseconds.

uint64\_t nanos**()**

**{**

uint64\_t ns **=** std**::**chrono**::**duration\_cast**<**std**::**chrono**::**nanoseconds**>(**std**::**chrono**::**high\_resolution\_clock**::**

now**().**time\_since\_epoch**()).**count**();**

**return** ns**;**

**}**