//Union-Find

```
class unionfind:
    def __init__(self, n):
        self.parent = list(i for i in range(n))
        self.weight = [1]*n

    def find(self, n): # find the root of index n
        if self.parent[n] == n:
            return n
        self.parent[n] = self.find(self.parent[n]) # path compression
        return self.parent[n]

    def union(self, a, b):
        self.weight[self.find(a)] += self.weight[self.find(b)] # note: update weight before updating parent
        self.parent[self.find(b)] = self.find(a) # let the root of a be the root of b
```

//2D Prefix Sum

```
def prefixsum(matrix):
    row = len(matrix)
    col = len(matrix[0])
    ps = []
    for i in range(row):
        ps.append([0]*col)

    ps[0][0] = matrix[0][0]
    for i in range(1, row, 1):
        ps[i][0] = ps[i-1][0] + matrix[i][0]
    for i in range(1, col, 1):
        ps[0][i] = ps[0][i-1] + matrix[0][i]

    for i in range(1, row, 1):
        for j in range(1, col, 1):
            ps[i][j] = ps[i-1][j] + ps[i][j-1] - ps[i-1][j-1] + matrix[i][j]

    return ps

def getsum(ps, r1, c1, r2, c2):
    if r1 == 0 and c1 != 0:
        return ps[r2][c2] - ps[r2][c1-1]
    if c1 == 0 and r1 != 0:
        return ps[r2][c2] - ps[r1-1][c2]
    if r1 == 0 and c1 == 0:
        return ps[r2][c2]
    return ps[r2][c2] - ps[r2][c1-1] - ps[r1-1][c2] + ps[r1-1][c1-1]
```

//Segment Tree

```
#include<cstdio>
#include<iostream>
using namespace std;
int n; // n numbers in array
long long a[100001];
long long sum[400004];
long long ltag[400004]; // tag for lazy update
void pushup(int x)
{
	sum[x]=sum[x*2]+sum[x*2+1];
}
void build(int x,int l,int r)
{
	if(l==r)
	{
		sum[x]=a[l];
		return;
	}
	int m=(l+r)/2;
	build(x*2,l,m);
	build(x*2+1,m+1,r);
	pushup(x);
}
void pushdown(int x,int l,int r)
{
	if(ltag[x])
	{
		int m=(l+r)/2,lchild=x*2,rchild=x*2+1;
		ltag[lchild]+=ltag[x];
		sum[lchild]+=ltag[x]*(m-l+1);
		ltag[rchild]+=ltag[x];
		//sum[x]+=ltag[x]*(r-l+1); no need to update the sum here. pushup will do the job later.
		sum[rchild]+=ltag[x]*(r-m);
		ltag[x]=0;
	}
}
void update(int x,int l,int r,int L,int R,int add) // add a number "add" to each element in an interval [L,R]
// x: node in the tree, l: left bound of node x, r: right bound of node x
{
	if(l>R || r<L)return; // case 1: the current node has no intersection with [L, R]
	if(l>=L && r<=R) // case 2: the current node is totally inside of the interval [L, R]
	{
// only need to tag the current node, no need to update its children now. This is why the tag is called lazy tag.
		ltag[x]+=add;
		sum[x]+=add * (r-l+1);
		return;
	}
	// case 3: the current node partially intersects with [L, R]
	int m=(l+r)/2;
	// if there is a tag on the current node, need to push the tag to its children and clear its own tag.
	pushdown(x,l,r);
```

```
        update(x*2,l,m,L,R,add);
        update(x*2+1,m+1,r,L,R,add);
        pushup(x);
}
long long query(int x,int l,int r,int L,int R) // query the sum of the interval [L, R]
{
        if(l>R || r<L)return 0;
        if(l>=L && r<=R)return sum[x];
        int m=(l+r)/2;
        pushdown(x,l,r);
        return query(x*2,l,m,L,R)+query(x*2+1,m+1,r,L,R);
}

build(1, 1, n);                 //to build tree
update(1, 1, n, L, R, add)      //to add a number to [L, R]
query(1, 1, n, L, R)            //to query the sum of [L, R]
```

//Binary Search

```
int bs1(int* a, int x, int l, int r)  // binary search, find the rightmost element that is less than or equal to x
{                                     // suppose the array is non-decreasing
        while(l < r-1)
        {
                int mid = (l+r)/2;
                if(a[mid] <= x) // if the array is non-increasing, change < to >
                {
                        l = mid;
                }
                else
                {
                        r = mid;
                }
        }
        if(a[r] <= x)                 // if the array is non-increasing, change < to >
                return r;
        return l;
}

// binary search, find the rightmost element that is less than or equal to x. if multiple x exists in the array, return the leftmost one.
int bs2(int* a, int x, int l, int r)
{                                     // suppose the array is non-decreasing
        while(l < r-1)
        {
                int mid = (l+r)/2;
                if(a[mid] < x)
                {                     // if the array is non-increasing, change < to >
                        l = mid;
                }
                else
                {
                        r = mid;
                }
        }
        if(a[r] < x)                  // if the array is non-increasing, change < to >
                return r;
        if(a[r] == x && a[l] != x)
                return r;
        return l;
}

int bs3(int* a, int x, int l, int r) // binary search, find the rightmost element that is strictly less than x
{                                     // suppose the array is non-decreasing
        while(l < r-1)
        {
                int mid = (l+r)/2;
                if(a[mid] < x)
                {
                        l = mid;
                }
```

```
                else
                {
                        r = mid;
                }
        }
        if(a[r] < x)
                return r;
        return l;
}
```

//Find the first smaller element

```
# For every element i in a list (height), find the first element that is less than element i to its left and right in O(n).
from collections import deque
heights = list(map(int, input().split()))
firstlowerleft = [-1] * len(heights)
firstlowerright = [len(heights)] * len(heights)
stack = deque()
stack.append(0)
# use a stack to keep track of an increasing sequence.
# for height[i], pop all stack element that have higher height than column i and make i the top of the stack.
# We want to compute the column that is lower than column i with the greatest index (firstlowerleft), all the
columns that are higher than column i and to the left of column i can be ignored (popped out from the stack) since
column i will be a better candidate than them.
# In worst case, each i can be pushed into stack and popped from stack once. The time complexity is O(n).
for i in range(1, len(heights), 1):
   top = stack.pop()
   while heights[top] >= heights[i]:
      if len(stack) == 0:
         top = -1
         break
      top = stack.pop()
   firstlowerleft[i] = top
   if top != -1: # if some column on the left is lower than column i, it should remain in the stack
      stack.append(top)
   stack.append(i)

stack = deque()
stack.append(len(heights) - 1)
for i in range(len(heights)-2, -1, -1):
   top = stack.pop()
   while heights[top] >= heights[i]:
      if len(stack) == 0:
         top = len(heights)
         break
      top = stack.pop()
   firstlowerright[i] = top
   if top != len(heights):
      stack.append(top)
   stack.append(i)

print(firstlowerleft)
print(firstlowerright)
```

//KMP

kmp O(n+m) string matching algorithm

basic idea: skip matching positions that are impossible to find the pattern string
the prefix of the next match must equal to the suffix of the current match.

input: text T with length n, pattern P with length m.

output: find all occurences of P in T

define next[i]: the maximum length of the common string of the prefix and suffix of P[0:i] (all range notations are inclusive)

explanation:

Suppose we have a partial match from T[x] to T[y], and the match broke at T[y+1]. Then T[x:y] = P[0:y-x], T[y+1] != P[y-x+1].

In naive method, we restart matching from T[x+1] and P[0]. However, in many cases the new matching will never succeed, only wasting time.

How can we know the next try is possible to succeed? In the last try of matching, we already gone through T[x+1] to T[y]. This information can be utilized.

The answer is, the prefix of the next match must equal to the suffix of the current match. Otherwise the match is bound to fail.

Example: T = aaaaabc P = aaaaad

In the first match, we end up like this:

```
          v
a a a a a b c
a a a a a d
          ^
```

T[0:4] = P[0:4], but the match break at T[5] and P[5].
In naive method, we move the P forward like this:

```
  v
a a a a a b c
  a a a a a d
  ^
```

and start matching from the beginning of P.
But in the last match, we already know T[0:4] = P[0:4].
By checking the "next" array, we know P[0:3] = P[1:4], therefore T[1:4] = P[0:3].
So we can directly align P[0:3] = T[1:4] and start from T[5] and P[4].

```
          v
a a a a a b c
  a a a a a d
          ^
```

By similar approach, the next match becomes

```
          v
a a a a a b c
    a a a a a d
          ^
```

Each time in the loop, either the P string move to the right at least 1 unit, or the "i pointer" (pointer of T string) add 1.

At most, i can increase n times and the P string can move to the right n-m times.

Therefore, the max number of loops = n-m.

Preprocessing needs 2m time. so total time is O(n+m).

step 1: preprocess array next

This step is like applying kmp to itself.

step 2: start pattern matching.

Code:

```
#include<iostream>
#include<cstring>
using namespace std;
char s1[1000001];
char s2[1000001];
int kmpnext[1000001];
// definition of kmpnext[i]: the maximum length of the common string of the prefix and suffix of s2[0:i] (inclusive)
// if index starts from 1, change the definition to the maximum length of ... s2[1:i] (inclusive)

// if index starts from 1, just offset every index by 1 and change every 0 into 1 (marked offset).
void getkmpnext(int l2)
{
// in preprocessing, i starts from 1 and j from 0, since there's no point matching the whole string with itself.
	int i=1, j=0; // offset
	kmpnext[0] = 0;
	while(i<l2)
	{
		if(s2[i] == s2[j])
		{
			kmpnext[i] = j+1;
			i++;
			j++;
		}
		else if(j>0) // offset
		{
			j = kmpnext[j-1];
		}
		else //j = 0 and s2[i]!=s2[j]
		{
			kmpnext[i]=0; // offset
			i++;
		}
	}
}
```

```
void kmp(int l1, int l2)
{
	// in real matching, i and j both start from 0.
	int i=0,j=0; // offset
	while(i<l1)
	{
		if(s1[i] == s2[j]) // if current character matches, continue matching
		{
			i++;
			j++;
			if(j==l2) // finished matching
			{
				cout<<i-l2<<endl;
				j = kmpnext[j-1];
			}
		}
		else if(j>0) // offset, if match is broken, move j back to the next possible position
		{
			j = kmpnext[j-1];
		}
		else //j = 0 and s1[i]!=s2[j]
		{
			i++;
		}
	}
}

int main()
{
	cin>>s1;
	cin>>s2;
	int l1 = strlen(s1);
	int l2 = strlen(s2);
	getkmpnext(l2);
	kmp(l1, l2);
}
```

//01 Knapsack Problem

```
def knapsack(maxweight:int, weight: list, value: list):
	# define dp[i][j]: the max value can be obtained when trying to grab the first i items within total weight j
	# Formula: dp[i][j] = max(dp[i-1][j], dp[i-1][j-weight[i]]+value[]i), initial condition: dp[0][j] = 0 for any j
	dp = [0]*(maxweight+1)
  for i in weight:
    for j in range(maxweight, 0, -1): # since we use rolling array, need to loop in reverse order since later dp element depends on earlier dp element
      if j >= i:
        dp[j] = max(dp[j], dp[j-i]+value[i])
      #else:
      #   dp[j] = dp[j]
  return dp[maxweight]
```

// Longest Increasing/Non-Decreasing Sequence (LIS)

Input: a list l.

O(n^2) solution:

Define dp[i]: the longest increasing sequence that ends at i.

Formula:

dp[i] = max(dp[x from 0 to i-1] if l[i] > l[x]) + 1

Initial condition:

dp[i] = 1 for any i.

Answer:

max(dp)

O(nlogn) solution:

Define dp[i]: the least ending number of all the increasing subsequence with length i

Formula:

for j from 0 to n-1:
if l[j] is larger than every number in dp (the last number in dp), then append l[j] to the end of dp.
otherwise dp[i+1] = l[j], where dp[i] is the rightmost number that is less than l[j].

Explanation:
Suppose we already have dp at j-1th loop. At this point we have already computed len(dp) possible increasing sequence. For l[j], we can add it to any possible increasing sequence whose last number is less than l[j].

Example: we have dp = [1, 2, 4, 5] and l[j] = 3.
Can put 3 after 1 to form an increasing sequence of length 2. However, the least ending number of length 2 sequence is already 2. 3 > 2 so we don't update it.

Can put 3 after 2 to form an increasing sequence of length 3. The known least ending number of length 3 sequence is 4. so update 4 to 3.

For items in dp larger than 3, 3 cannot be added after them to form an increasing sequence.

Initial condition:

dp[0] = -Infinity

Answer:

dp[-1] (the last number in dp)

Since dp is non-decreasing ordered, can use binary search to find i in log(n) time.

Code:

```
int dp[100001];
int h[100001];
const int inf = 2147483647;

int bs(int x, int l, int r) // binary search, find the rightmost element that is strictly less than x
{                           // suppose the array is non-decreasing
        while(l < r-1)
        {
                int mid = (l+r)/2;
                if(dp[mid] < x)
                {
                        l = mid;
                }
                else
                {
                        r = mid;
                }
        }
        if(dp[r] < x)
                return r;
        return l;
}

int LIS() // longest (strictly) increasing sequence
{
        dp[0] = -inf;
        int dplen = 0;
        for(int i=1;i<=n;i++)
        {
                if(h[i] > dp[dplen])
                        dp[++dplen] = h[i];
                else
                {
                        int x = bs(h[i], 0, dplen);
                        // dp[x] is the rightmost element that is strictly less than x, so h[i] can be put after dp[x]
                        dp[x+1] = h[i];
                }
        }
        return dplen;
}
```

Longest Common Sequence (LCS)

Input: two strings a and b.

Define dp[i][j]: the length of the longest common sequence of a[0, i] and b[0, j], inclusive.

Formula:

dp[i][j] = dp[i-1][j-1] + 1 if a[i] == b[j]

dp[i][j] = max(dp[i-1][j],dp[i][j-1]) otherwise

dp is an non-decreasing sequence for i and j. i.e. dp[i+1][j] >= dp[i][j], dp[i][j+1] >= dp[i][j]
so only need to compare dp[i-1][j] and dp[i][j-1], which are the two possible largest known element in dp.

Initial condition:

dp[0][0] = 1 if a[0] == b[0]
dp[i][0] = 1 if any a[x from 0 to i] == b[0]
dp[0][j] = 1 if any a[y from 0 to j] == a[0]
we can just start i, j from 1 and set the first row and column of dp to 0.
in this case there is no need to manually initialize dp.

Answer:
dp[n][n]

Code:

```
int dp[1001][1001];
char s1[1001];
char s2[1001];

int LCS(int n, char* s1, char* s2)
{
	int i,j,ans=0;
	memset(dp,0,sizeof(dp));
	for(i=1;i<=n;i++)
	{
		for(j=1;j<=n;j++)
		{
			if(s1[i]==s2[j])
			{
				dp[i][j] = dp[i-1][j-1]+1;
			//dp[i-1][j-1]+1 is large or equal to dp[i-1]][j] and dp[i][j-1] so no need to compare them
			// because dp[i-1][j] is at most dp[i-1][j-1]+1 and the same for dp[i][j-1]
			}
			else
			{
				dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
			}
		}
```

```
	}
	/*for(i=0;i<l1;i++)
		for(j=0;j<l2;j++)
			if(dp[i][j]>ans)
				ans=dp[i][j]; again, dp is increasing, so dp[n][n] is guaranteed to be the largest element*/
	return dp[n][n];
}

// memory optimization: rolling array
// idea: notice dp[i][j] only depends on dp[i-1][j], dp[i][j-1] and dp[i-1][j-1], which are the last "row" and "column" in the dp table
// The last "column" is computed in the loop of j. We only need to keep the last row.
int dp[2][1001];

void swap(int& a, int& b)
{
	int tmp = a;
	a = b;
	b = tmp;
}

int LCS(int n, char* s1, char* s2)
{

	int i,j,ans=0;
	memset(dp,0,sizeof(dp));
	int lastrow = 0;
	int thisrow = 1;
	for(i=1;i<=n;i++)
	{
		for(j=1;j<=n;j++)
		{
			if(s1[i]==s2[j])
			{
				dp[thisrow][j] = dp[lastrow][j-1]+1;
			}
			else
			{
				dp[thisrow][j]=max(dp[lastrow][j],dp[thisrow][j-1]);
			}
		}
		swap(lastrow, thisrow);
	}

	return dp[lastrow][n];
}
```

//Some Notes on Geometry

check whether a point p is in a polygon:

loop through the vertices of the polygon a1 ... an, in clockwise order.
if all angle (a1, p, a+1) are counterclockwise, the point is in the polygon.
To check whether the angles is counterclockwise, use the same cross product technique used in convex hull.
That is, (a1 p) x (p a2) > 0, the angle is counterclockwise.

Area or triangle: 0.5 * a * b sin(theta), where theta is the angle between side a and b
Area of regular n-gon: 0.5 * n * r * r * sin(2 * PI / n);

sum of angle of n-gon: (n-2) * PI

Hint: rotational symmetry and polar coordinates can be useful sometimes.

Converting Cartesian to polar
The value of tan-1(y/x) may need to be adjusted:
Quadrant I: Use the calculator value
Quadrant II: Add 180°
Quadrant III: Add 180°
Quadrant IV: Add 360°

Code:

```
from math import pi
from math import sqrt
from math import atan
def cartesian2polar(x, y):
    r = sqrt(x**2+y**2)
    if x != 0: # avoid division of zero
        alpha = atan(y/x)
        if x<0: # atan only returns angle from -90 to 90. need to manually adjust.
            alpha += pi
        elif x>0 and y<0:
            alpha += 2*pi
    else:
        if y>=0:
            alpha = pi/2
        elif y<0:
            alpha = 3*pi/2

    return r, alpha
```

//Convex Hull

```
n = int(input())

def vec(a:'point',b:'point'):
   return (b[0]-a[0], b[1]-a[1])

def ccw(a:'vector', b:'vector'): # check counterclockwise using a crossproduct b
   return a[0]*b[1] - a[1]*b[0]

def buildhull(points, factor): #factor is used to find upper hull and lower hull
   hull = []
   for p in points:
      while len(hull) >= 2:
         a = vec(hull[-2], hull[-1])
         b = vec(hull[-1], p)

         if ccw(a,b) * factor > 0:
            break
         else:
            hull.pop()

      while len(hull) > 0 and p == hull[-1]:
         hull.pop()

      hull.append(p)
   return hull

while n:
   points = []
   for i in range(n):
      x, y = input().split()
      points.append((int(x),int(y)))

   points.sort()

   lower_hull = buildhull(points, 1)
   upper_hull = buildhull(points, -1)

   print(len(lower_hull) + max(0, len(upper_hull)-2))

   for p in lower_hull:
      print(p[0],p[1])
   for i in range(len(upper_hull)-2, 0, -1):
      print(upper_hull[i][0], upper_hull[i][1])


   n = int(input())
```

//Extended Euclidean Algorithm (EXGCD)

Goal: Find x, y that satisfies a*x + b*y = gcd(a,b)
i.e. express gcd(a,b) as a linear combination of a and b

idea: in each step of the Euclidean algorithm, a % b is computed.
a % b is already expressed as a linear combination of a and b.
a = b*(a//b) + a % b
a % b = a - b*(a//b)
Now a % b is expressed as a linear combination of a and b: 1 * a - (a//b) * b.
The two coefficients are 1 and -a//b, which are shown in l2 in the prep function below.

Step 1: Preperation
Let the list of numbers generated in the gcd process be l1.
Let the coefficients of linear combination be l2.
That is, l1[i] = l2[i][0] * l1[i-2] + l2[i][1] * l1[i-1] (Eq.1),
Where l2[i][1] = -l1[i-2] // l1[i-1]
Notice the first coefficient is always 1, we can just let l2 be and 1D array, only storing the second coefficient.

Step 2: Recursion Relation
Now our goal is to express l1[-1] as a linear combination of l1[0] and l1[1].
Suppose we have a magical function exgcd(i) that returns the linear combination of l1[i] in l1[0] and l1[1], the coefficients form a tuple of 2 elements.
Expressing l1[i-1] and l1[i-2] in exgcd:
l1[i-1] = exgcd(i-1)[0] * l[0] + exgcd(i-1)[1] * l[1]
l1[i-2] = exgcd(i-2)[0] * l[0] + exgcd(i-2)[1] * l[1]

Plug these values into the definition of l2 (Eq.1):
l1[i] = l2[i][0] * l1[i-2] + l2[i][1] * l1[i-1]
= l2[i][0] * (exgcd(i-2)[0] * l1[0] + exgcd(i-2)[1] * l1[1]) + l2[i][1] * (exgcd(i-1)[0] * l1[0] + exgcd(i-1)[1] * l1[1])
= (l2[i][0] * exgcd(i-2)[0] + l2[i][1] * exgcd(i-1)[0]) * l1[0] + (l2[i][0] * exgcd(i-2)[1] + l2[i][1] * exgcd(i-1)[1]) * l1[1] (Just pulling out l1[0] and l1[1])
The coefficient of l1[0] is (l2[i][0] * exgcd(i-2)[0] + l2[i][1] * exgcd(i-1)[0]) (Call it c1)
The coefficient of l1[1] is (l2[i][0] * exgcd(i-2)[1] + l2[i][1] * exgcd(i-1)[1]) (Call it c2)

Now we have a recursion relation: exgcd(i) = (c1, c2). Up to now we can actually implement the function recursively.
Notice the base case: l1[0] and l1[1] are justs linear combination of themselves.

Code:

```
def prep(a,b):
    l1 = [a,b]
    l2 = [None,None]
    while l1[-2] % l1[-1] != 0:
        # l1: record every number generated in the gcd progress. l1[-1] is gcd(a,b)
        l1.append(l1[-2] % l1[-1])
        # l2: record how the current number can be represented by a linear combination of the last two numbers
        l2.append((1,-(l1[-3]//l1[-2])))
    return l1,l2

def exgcd(n, l1, l2): #brute recursive algorithm
    if n == 0:
```

```
        return (1,0)
    if n == 1:
        return (0,1)

    return (exgcd(n-2, l1, l2)[0] * l2[n][0] + exgcd(n-1, l1, l2)[0] * l2[n][1] , exgcd(n-2, l1, l2)[1] * l2[n][0] + exgcd(n-1, l1, l2)[1] * l2[n][1])

# The performance of recursive implementaion sucks.
# There should be a smarter linear algorithm.
# Do it in an iterative way (just like memoizing the result of each exgcd call):
# Using the same relationship, just replacing exgcd(i-1), exgcd(i-2) to dp[i-1], dp[i-2]

def exgcd_dp(l2): #dp time complexity optimization O(2^n) -> O(n)
#dp[i]: records how the ith number can be represented by a linear combination of a and b(the numbers inputed)
    dp = [(1,0), (0,1)]
    for i in range(2, len(l2)):
        dp.append((dp[i-2][0] * l2[i][0] + dp[i-1][0] * l2[i][1], dp[i-2][1] * l2[i][0] + dp[i-1][1] * l2[i][1]))
    return dp[-1]

# Notice, each iterative step only depends on the last 2 elements of the dp array.
# Only need to keep track of 2 elements.
def exgcd_dp1(l2): #space complexity optimization O(n) -> O(1)
    last2, last1 = (1,0), (0,1)
    current = last1 # for the case b evenly divides a
    for i in range(2, len(l2)):
        current = (last2[0] * l2[i][0] + last1[0] * l2[i][1], last2[1] * l2[i][0] + last1[1] * l2[i][1])
        last2, last1 = last1, current
    return current

# it works for a>b or a<b, a|b or b|a.
if __name__ == '__main__':
    a,b = input().split()
    a = eval(a)
    b = eval(b)
    l1, l2 = prep(a,b)
    #print(exgcd(len(l1) - 1, l1, l2))
    #print(exgcd_dp(l2))
    print(exgcd_dp1(l2))

# Usage for finding Modular Multiplicative Inverse:
# Definition of Modular Multiplicative Inverse:
# x is the MMI of a (mod b) if ax % b == 1
# In other words, xa + yb  = 1

# We know gcd(a,b) = xa + yb
# When gcd(a,b) = 1, exgcd gives us exactly x and y.
# x is the answer.
# x can be negative sometimes, mod b to make it positive.
```

//Chinese Remainder Theorem (CRT)

```
# Goal: Given a set of Linear Congruence Equations (all n's are relatively prime):
# x % n1 = a1 % n1
# x % n2 = a2 % n2
# ...
# x % nk = ak % nk
# find x

def prep(a,b):
    l1 = [a,b]
    l2 = [None,None]
    while l1[-2] % l1[-1] != 0:
        # l1: record every number generated in the gcd progress. l1[-1] is gcd(a,b)
        l1.append(l1[-2] % l1[-1])
        # l2: record how the current number can be represented by a linear combination of the last two numbers
        l2.append((1,-(l1[-3]//l1[-2])))
    return l1,l2

def exgcd_dp1(l2): #space complexity optimization O(n) -> O(1)
    last2, last1 = (1,0), (0,1)
    current = last1 # for the case b evenly divides a
    for i in range(2, len(l2)):
        current = (last2[0] * l2[i][0] + last1[0] * l2[i][1], last2[1] * l2[i][0] + last1[1] * l2[i][1])
        last2, last1 = last1, current
    return current

def mi(a, b):
    '''multiplicative inverse of a (mod b), not to be confused with m_i below'''
    l1, l2 = prep(a,b)
    mul_inv = exgcd_dp1(l2)[0] # the first element is the answer
    return mul_inv % b # make it positive

def crt(a: list, n: list):
    # Step 1: calculate the product of all modular numbers in list n
    nprod = 1
    for i in n:
        nprod *= i

    # Step 2: for the ith equation,
    ans = 0
    for i in range(len(n)):
        # Step 2.1 calculate m_i = nprod//n_i
        m_i = nprod//n[i]

        # Step 2.2 calculate the Multiplicative Inverse of mi (mod ni), m_i_inv
        m_i_inv = mi(m_i, n[i])

        # Step 2.3 calculate c_i = m_i * m_i_inv
        c_i = m_i * m_i_inv
        ans += a[i] * c_i;
```

```
    # Step 3: the answer is sum(a_i*c_i) % nprod.
    return ans % nprod

if __name__ == '__main__':
    k = int(input())
    a = [0] * k # a is the remainder
    n = [0] * k # n is the modular number
    for i in range(k):
        x, y = input().split()
        a[i] = int(x)
        n[i] = int(y)

    print(crt(a, n))

# Extended CRT
# Addresses the case that the modular numbers are not relatively prime.
# First consider the case of two equations:
# x % n1 = a1 % n1, x % n2 = a2 % n2.
# Convert them to undetermined equation:
# x = n1*p+a1 = n2*q+a2, where p and q are integers
# n1*p - n2*q = a2 - a1
# Using Bezout's identity, when a2-a1 cannot be evenly divided by gcd(n1, n2), there is no solultion.
# Otherwise can express (a2 - a1) as a linear combination of gcd(n1, n2) using exgcd,
# then express (a2 - a1) as a linear combination of n1 and n2, producing one solution for (p, q), and we can get x.

# For multiple (more than 2) equations,
# Write the solution above in the form of linear congruent equation:
# x % M = b % M, where b = n1*p + a1 (n2*q + a2 is also fine), M = lcm(m1, m2).
# In this case we eliminated 2 equations, and added 1 new equation to the equation set.
# Do this elimination until there's only 1 equation.
```

//Topological Sort

```
def toposortdfs(al): #al: adjacent list, al[i] stores all vertices that i can reach directly
    vis = [False] * n
    ts = []

    for i in range(len(al)):
        if vis[i] == False:
            toposortdfs_r(i, al, vis, ts)

    ts.reverse()
    return ts

def toposortdfs_r(v, al, vis, ts):
    vis[v] = True
    for i in al[v]:
        if vis[i] == False:
            toposortdfs_r(i)
    ts.append(v) # the only thing added from a standard dfs
    # v is a prerequisite of all i's.
    # appending v to ts after all i's are searched guarantees v appear later than all i's.
    # therefore ts reversed is a topological sort.

# Kahn's algorithm
from collections import deque
def toposortbfs(al):
    ts = []
    # calculate incoming degree
    ind = [0] * len(al)
    for i in al:
        for j in i:
            ind[j] += 1

    # put all vertices with 0 incoming degree to a queue
    # (maybe a priority queue, to satisfy other requirements of the sort)
    q = deque()
    for i in range(len(ind)):
        if ind[i] == 0:
            q.append(i)

    while len(q) != 0:
        v = q.popleft()
        ts.append(v)
        # removal of vertex v can cause deduction of incoming degree of other vertices
        for i in al[v]:
            ind[i] -= 1
            if ind[i] == 0:
                q.append(i)

    return ts
    # this time ts does not need to be reversed
```

//Find Loop in a Graph

```
def findloop_dfs(al, v, vis, instack, stack): # if vertex v can reach some vertex that is on a loop, the loop can be found. In other words, if some vertex was reached from v without finding a loop, it is not on a loop.
    vis[v] = 1
    instack[v] = 1
    stack.append(v)
    for i in al[v]:
        if vis[i] == 0:
            res = findloop_dfs(al, i, vis, instack, stack)
            if res != None:
                return res
        elif instack[i] == 1: # found a loop, if the graph is undirectional, use elif instack[i] == 1 and i != stack[-2]
            for j in range(len(stack)): # find which vertices are on the loop
                if stack[j] == i:
                    res = stack[j:]
                    stack.pop(-1)
                    instack[v] = 0
                    return res
    stack.pop(-1)
    instack[v] = 0

# suppose each loop is separate. In other words, one vertex can only belong to one loop.
def findloop(al: 'adjacent list'):
    n = len(al)
    vis = [0] * n
    for i in range(n):
        if vis[i] == 0:
            res = findloop_dfs(al, i, vis, [0]*n, [])
            if res != None:
                return res
```

//Dijkstra with Heap Optimization

```
#include<iostream>
#include<queue>
#include<vector>
using namespace std;

int n,m,s; // n vertices, m edges, s is the starting point
const int inf = 2000000000;

vector<pair<int, int> > al[100001];
int d[100001];
bool vis[100001];

struct node
{
	int no;
	int dist;
}tmp;

struct cmp
{
	bool operator()(node a, node b)
	{
		return a.dist > b.dist;
	}
}

priority_queue<node, vector<node>, cmp> q;

void dijk(int o)
{
	vis[o] = 1;
	for(int i=1;i<=n;i++)
	{
		d[i] = inf;
	}
	d[o] = 0;
	int k;

	for(int i=0;i<al[o].size();i++)
	{
		int t = al[o][i].first;
		int w = al[o][i].second;
		d[t] = w;
		tmp.no = t;
		tmp.dist = w;
		q.push(tmp);
	}

	for(int i=1;i<=n-1;i++)
	{
```

```
                if(q.empty()) break;
                while(!q.empty())
                {
                        tmp = q.top();
                        if(vis[tmp.no] == true)
                        {
                                q.pop();
                                continue;
                        }
                        k = tmp.no;
                        break;
                }

                vis[k] = 1;
                // update d
                for(int j=0;j<al[k].size();j++)
                {
                        int t = al[k][j].first;
                        int w = al[k][j].second;
                        if(d[k] + w < d[t] && vis[t] == 0)
                        {
                                d[t] = d[k] + w;
                                tmp.no = t;
                                tmp.dist = d[t];
                                q.push(tmp);
                        }
                }
        }
}

int main()
{
        cin>>n>>m>>s;
        int a, b, w;
        for(int i=0;i<m;i++)
        {
                cin>>a>>b>>w;
                al[a].push_back(make_pair(b, w));
                al[b].push_back(make_pair(a, w));
        }
}
```

//Tarjan

```
/*Tarjan: compute strongly connected components (SCC)
maintain variable dfu and low for each vertex.
dfn[u]: The order node n being visited in the dfs process
low[u]: The lowest node dfn reachable from u. All vertices in a SCC have the same low value.

maintain a stack

Suppose we are currently at u. Do dfs for all v's such that there's and edge u->v. When backtrack (after dfs(v)),
update low[u] to min(low[u], low[v]) if v is on stack.

When find an SCC (dfn[u] == low[u]) after all dfs from u finishes, remove all elements from the stack. */

#include<iostream>
#include<vector>
using namespace std;

int n, m;
vector<int> al[100];
bool vis[100];

int dfn[100];
int maxdfn;
int low[100];

bool onstack[100];
int stack[100];
int stacklen;
int scccnt;

void dfs(int v)
{
	// record the order v being searched
	dfn[v] = maxdfn;
	// set low to itself
	low[v] = dfn[v];
	maxdfn++;

	// push v into the stack
	stack[stacklen] = v;
	stacklen++;
	onstack[v] = true;
	vis[v] = true;

	// do dfs
	for(int i=0;i<al[v].size();i++)
	{
		int next = al[v][i];
		if(!vis[next])
		{
			dfs(next);
```

```
			}
			if(onstack[next])
			{
				low[v] = min(low[v], low[next]);
			}
		}

		// check whether a SCC was found
		if(low[v] == dfn[v]) // SCC found
		{
			// pop all the nodes in the current SCC from the stack
			while(stacklen > 0)
			{
				// pop 1 node from the stack until the start of the SCC (inclusive)
				stacklen--;
				onstack[stack[stacklen]] = false;
				if(stack[stacklen] == v) //notice here is v, not dfn[v] since the stack stores vertex number
					break;
			}

			scccnt++;
		}

}

int main()
{
		cin>>n>>m;
		for(int i=0;i<m;i++)
		{
			int f, t;
			cin>>f>>t;
			al[f].push_back(t);
		}
		for(int i=0;i<n;i++)
		{
			if(!vis[i])
				dfs(i);
		}
		cout<<scccnt<<endl;
		//vertices of the same SCC should have the same low value.
		for(int i=0;i<n;i++)
		{
			cout<<low[i]<<" ";
		}
		cout<<endl;
}
```

//Print _int128

```
void printint128(__int128 x)
{
	if(x<0)
	{
		putchar('-');
		x=-x;
	}
	if(x<10)
	{
		putchar(x+48);
		return;
	}
	print(x/10);
	putchar(x%10+48);
}
```

//Python Priority Queue

```
import heapq

class minq:
   def __init__(self):
      self.h = []

   def push(self, x):
      heapq.heappush(self.h, x)

   def pop(self):
      return heapq.heappop(self.h)

   def top(self):
      return self.h[0]

   def __len__(self):
      return len(self.h)

class maxq:
   def __init__(self):
      self.h = []

   def push(self, x):
      heapq.heappush(self.h, -x)

   def pop(self):
      return -heapq.heappop(self.h)

   def top(self):
      return -self.h[0]

   def __len__(self):
      return len(self.h)
```

//Some Random Notes

1. bfs cannot keep track all possible paths.

2. recursion and stack limit

```
import resource, sys
resource.setrlimit(resource.RLIMIT_STACK, (2**29,-1)) # may not work
sys.setrecursionlimit(10**6) # Will mostly cause stack overflow or TLE if need this many recursions
```

3. Python binary and int

Convert binary string bs to int in Python: int(bs, 2)
Convert int to binary string: bin(x)

4. Large Numbers

when processing very large numbers (e.g. 10^(10^5)), store lower digits in lower index for convenience.

5. Greedy

Sometimes can convert things to intervals, sort the intervals by their upper bound and use greedy.
Sometimes greedy is not just coming up with a single equation. Can do some brute-force if time complexity allows.

6. Sorted Structure

Don't try to modify things in a priority queue/heap. Just insert new entries, they will "mask off" the old ones, and remember to ignore the old ones.

7. High-level View

Sometimes do a high-level counting may works... instead of diving too deep into the structural details.

8. C++ Long Long

Use LONG LONG! Don't bother deciding which to use int and which to use long long when the numbers are large. Just use everything as Long Long.

9. READ PROBLEMS CAREFULLY!

Remember the weird problems: INPUT ON MULTIPLE LINES / you DO NOT need to minimize k