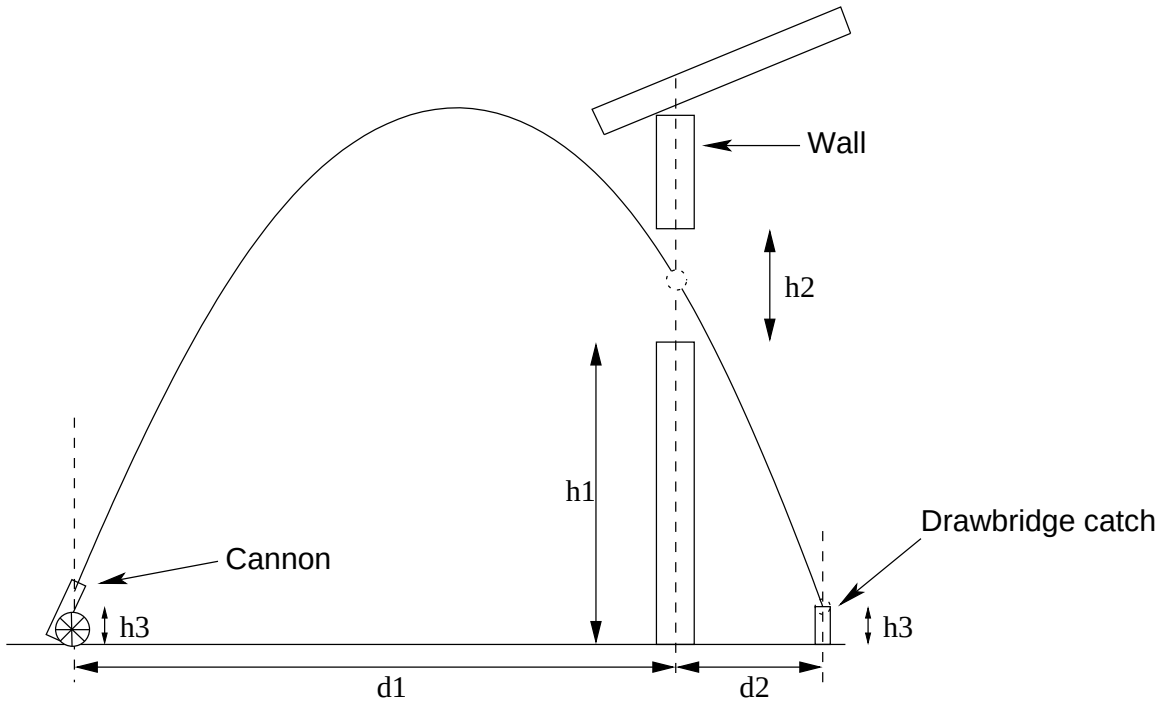
5^{th} Southern Africa Regional ACM International Collegiate Programming Contest

Sponsored by IBM

Cannon Fodder (Red Balloon)

While performing a fairly complicated experiment involving a particle accelerator, a mislabeled 4.77MHz crystal oscillator and a rubber band, you are accidentally transported back to the 18th century. At least you still have your trusty palmtop computer, or the natives would have executed you on arrival. Anyway, their problem is this: they have a fairly decent cannon, but they have virtually no idea how to aim it to hit their intended target. You 'offer' to write a simulation to help them.
Here's a diagram of the trajectory that they would like to fire their projectiles at:



Their intended target is the drawbridge catch, as indicated on the diagram. The projectile has to pass through the exact centre of the porthole, given the following parameters:

$h1$	height of the porthole sill (measured from the ground)
$h2$	height of the porthole opening
$h3$	height of the intended target
$d2$	distance from centre of wall to centre of target

You have the following information regarding the cannon:

$h3$	height of the exit point of the barrel (same as height of target)
m	mass of projectile (kg)
spd	muzzle velocity (speed) of the projectile ($m \cdot s^{-1}$)

You will have to calculate the following values:

θ	angle between cannon barrel and the ground
$d1$	the distance from the tip of the barrel to the centre of the wall

The tricky part is that you have to take into account the atmospheric friction that the cannonball will experience. Fortunately you still remember the differential equation for linear friction, so you derive the following formula (note: **Bold Capital** letters denote vectors):

$$\mathbf{Pos}(t) = \mathbf{V}_1 \frac{m}{k} e^{k \cdot t/m} - \mathbf{G} \frac{m \cdot g \cdot t}{k} + \mathbf{P}_1$$

where (note: square brackets [ ] are used to group vector components)

$\mathbf{Pos}(t)$	=	a vector indicating the position (in meters) of object relative to its position at time 0
t	=	time in seconds, $t = 0$ when projectile leaves the muzzle
$\mathbf{V}_1$	=	$\mathbf{V}_0 + \mathbf{G}\frac{m \cdot g}{k}$
$\mathbf{V}_0$	=	[initial horizontal velocity, initial vertical velocity]
$\mathbf{G}$	=	$[0, -1]$. This is the direction of gravity
m	=	mass of object, in kg
g	=	$9.8\ m \cdot s^{-2}$
k	=	viscosity coefficient $= -1.0$
$\mathbf{P}_1$	=	$\mathbf{P}_0 - \mathbf{V}_1 \frac{m}{k}$
$\mathbf{P}_0$	=	[initial horizontal position, initial vertical position]

Note that the projectile leaves the muzzle of the cannon at *spd* meters per second; this is the linear speed, so you must calculate the correct initial velocity ($\mathbf{V}_0$ above) using the angle of the barrel. The viscosity coefficient is determined by the surface area of the projectile, as well as the density of the atmosphere. Since you don't know the geometry of the projectile, nor the exact atmospheric conditions, the specified value of $k = -1.0$ should be used.

Your task is to compute the appropriate angle θ that the cannon must be set to, as well as the corresponding distance from the wall ($d1$) that it must be placed at. The cannonball must pass through the exact centre of the porthole (at height $h1 + 0.5h2$ from the ground) and hit the top of the drawbridge catch (the point $h3$ from the ground, distance $d2$ from the middle of the wall).

Sample Input

Your input is an arbitrary number of records, each record conforming to the format:

```
h1 h2 h3 d2 m spd
```

using the definitions of these symbols as provided above. Here is a test sequence:

```
3 1 1 3 10 150
```

Sample Output

Your output must be the angle θ (in degrees) followed by the distance $d1$. The sample input sequence above will produce the output

```
22.9 884.4
```

All output values are to be rounded to one decimal place, as shown above.

5^{th} Southern Africa Regional ACM International Collegiate Programming Contest

Sponsored by IBM

Power distribution (Yellow Balloon)

It is the year 2307.2, and you are designing a power network for distributing energy using obscuroton^{tm} particles (apologies to Gene Roddenberry). The basic principle behind the obscuroton^{tm} power distribution network is the same as building an electricity grid: you want to minimise the losses throughout the network. The power network consists of a number of distribution nodes placed in hyperspace (to overcome the speed-of-light problem). Due to the influence of gravity wells (stars and such), not all connections are possible, so you have a map showing all the possible connections between the nodes. Each connection is bi-directional, with a specified resistance, which is the same for both directions. The resistance of a link is a measure of how many of the obscuroton^{tm} particles will be lost due to scattering on that link, in other words, how much energy will be lost.

Your goal is to construct the network with the lowest cumulative resistance by selecting the most appropriate connections from your map. All nodes must remain connected to all other nodes. There must be exactly one path from any given node to any other node; if there's more than one path between two nodes the resulting sub-space anomalies will tear apart the entire network, leaving you to explain the black-out to billions of irate aliens. Aliens with nasty-looking guns.

The real catch, however, is that you have a couple of Dork^{tm} distribution nodes that fell off the back of a cube, somewhere (sorry, Gene). These Dork^{tm} nodes are of an inferior design, and will double the resistance on all direct connections from *and to* it. This is because the Dork^{tm} nodes have trouble remodulating the phase of their containment fields, resulting in more scattering (it appears that all problems are either caused, or can be solved, by containment field phase modulation). If node A is connected to node B with a resistance value of 3, then the effective resistance will be $3 \times 2 \times 2 = 12$, if both A and B are Dork^{tm} nodes. It would appear that resistance is expensive, rather than futile, as is commonly believed amongst the Dork^{tm}.

In any case, you have N nodes in total, with D of them being Dork^{tm} nodes (you got them for free, so you have to use all of them). You have to place these Dork^{tm} nodes where they will do the least damage. Your output will be the total effective resistance of the most efficient network possible given the input.

Sample input

Your input consists of an arbitrary number of records, where each record represents a network node map. Each network node map has the following structure:

```
N D
nodeA1 nodeB1 resistance
nodeA2 nodeB2 resistance, etc.
```

The first number specifies the size of the network (number of nodes); the second specifies the number of Dork^{tm} nodes you have ($D \leq N$). Each following line specifies the two nodes that are connected, and the resistance associated with that link, thus forming triples of the form (nodeA, nodeB, resistance), without the commas. Here's an example network map:

```
3 1
```

```
1 2 4
1 3 5
2 3 6
-1
```

It has 3 nodes, one of which is a Dork$^{\text{tm}}$ node. On the second line we see that the node 1 is connected to node 2 with a resistance of 4. Line three tells us that node 1 is connected to node 3 with a resistance of 5, and so on. Nodes are numbered 1 through N. A -1 signifies the end of the current record; another record may follow.

Sample output

Your output must be the sum of all the effective resistance values, with the Dork$^{\text{tm}}$ nodes placed where they have the least impact on the cumulative network resistance. Note that the network configuration for which you report this value must satisfy the condition requiring that all nodes must be connected to all other nodes along exactly one path, so all redundant connections must be removed before computing the total resistance. For the example above, note that the cumulative resistance can be minimised by making node 2 the Dork$^{\text{tm}}$ node. This results in the output

```
13
```

since $5 + 2 \times 4 = 13$, which is less than $6 + 2 \times 4 + 2 \times 5$ and $4 + 2 \times 5 + 2 \times 6$.

5^{th} Southern Africa Regional ACM International Collegiate Programming Contest

Sponsored by IBM

Fibonacci codes (Blue Balloon)

Everyone knows that rabbits multiply like, well, rabbits! It has been suggested that Fibonacci numbers predict the number of rabbits you would have after a number of generations. For those of non-*leporine* origin, here's a refresher on Fibonacci numbers:

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-2} + F_{n-1}, \text{ for } n \geq 2 \end{aligned}$$

thus the first 8 number of the sequence are

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21$$

The Fibonacci sequence has another useful property, other than the rabbit thing, that is. The sequence forms a (non-superadditive) base for a number system. This implies that any positive integer can be expressed as a combination of terms from the sequence, *e.g.*

$$33 = 1 + 3 + 8 + 21$$

More concisely, any positive integer n can be expressed as

$$n = B_1 \cdot F_1 + B_2 \cdot F_2 + B_3 \cdot F_3 + B_4 \cdot F_4 + \ldots + B_k \cdot F_k$$

where F_k represents the Fibonacci number k, and B_j represents a binary digit (0 or 1). Clearly $B_j = 0$ for all j such that $F_j > n$, so we only have to keep the digits B_1 through B_k where $F_{k-1} < n \leq F_k$. This means the number 33 can be expressed as follows:

$$33 = 1 \cdot 1 + 0 \cdot 2 + 1 \cdot 3 + 0 \cdot 5 + 1 \cdot 8 + 0 \cdot 13 + 1 \cdot 21$$

or, more compactly

$$\text{Fib}(33) = 1010101$$

Some more examples:

$$\begin{aligned} \text{Fib}(18) &= 000101 \\ \text{Fib}(6) &= 101 \end{aligned}$$

This mapping is not unique, since the base is not superadditive. Every binary string maps to some integer, and every integer has at least one representation is the Fibonacci base. In summary, instead of using the usual powers of two as base (1,2,4,8,16,32, etc.), we just use the Fibonacci sequence.

The most fascinating thing about the binary Fibonacci representation is that there exists for every positive integer a binary sequence that never contains two consecutive '1' digits. These can be found by first decomposing the number using the largest suitable number from the Fibonacci sequence. For example, to compute Fib(n), first find k such that $F_k \leq n < F_{k+1}$. Now we see that

$$\text{Fib}(n) = \text{Fib}(n - F_k) \mid 1$$

where | denotes concatenation. For example, note that Fib(3) = 001, and not 11. Anyway, note that the last digit is always '1', because we don't have to keep trailing zeros. This presents an interesting opportunity for constructing a variable-length code for use in a compression algorithm.

To see why, consider that you want to use the binary representation using the powers of two as a base for storing variable-length codes:

$$\begin{aligned} \text{Base}_2(33) &= 100001 \\ \text{Base}_2(18) &= 01001 \\ \text{Base}_2(6) &= 011 \end{aligned}$$

Note that the digits are written in the reverse order of what you may be used to. The reason for this will become clear shortly. If we simply concatenate the digits, we find that

$$\text{Base}_2(33)\text{Base}_2(6)\text{Base}_2(18) = 10000101101001$$

It is impossible to uniquely decode this string, since we don't know where the one number ends and the next one begins.

Looking at the the Fibonacci-base representation, we see that by appending a '1' to every number's binary representation gives us a decodable sequence:

$$\begin{aligned} \text{Fib}_1(33) &= 10101011 \\ \text{Fib}_1(18) &= 0001011 \\ \text{Fib}_1(6) &= 1011 \end{aligned}$$

so that

$$\text{Fib}_1(33)\text{Fib}_1(18)\text{Fib}_1(6) = 1010101100010111011$$

Now you can easily tell where one number ends and the next one begins: every time you find two consecutive 1's, you know that you have reached the end of the current number (keep in mind that this 'extra' 1 digit does not form part of the value of the number, and is discarded after the end of the number has been determined). This trick won't work with the base-2 representation; try it if you like.

This scheme provides us with a way to use fewer bits for representing small numbers. Many compression algorithms use such variable-length codes to encode numbers whenever a smaller number is expected to occur more frequently than a large one, for example the run lengths in a run-length encoder.

Sample Input

Your program will read in a contiguous stream of ASCII '0' and '1' characters (i.e. ASCII characters number 48 and 49). This stream will be a list of concatenated binary Fibonacci representations, complete with the extra '1' appended to each number (as shown above). You must decode each number, and print out the corresponding decimal value. Here is some sample input:

```
0101100000011001110101111
```

Note that this input stream uses one byte to store each bit, so you won't see any real compression. The largest number you will be expected to decode will be smaller than 2 to the power 32.

Sample Output

For the sample input string listed above, the output will be:

```
7 21 3 12 1
```

5^{th} Southern Africa Regional ACM International Collegiate Programming Contest

Sponsored by IBM

FSK Demodulation (Pink Balloon)

While working for a somewhat secret government branch involved in the SETI program, you are given a stream of numbers to decode. The colleague that handed you the data mentioned that they suspect the stream is a Frequency Shift Keyed (FSK) modulated signal. Your task is to write a demodulator, so that you can decode the message, and see what the ETs had to say.

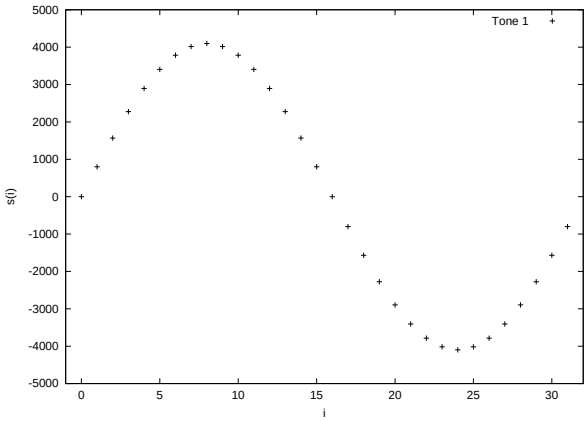
FSK modulation is the simplest of all modulation techniques. The modulation consists of a number of tones (frequencies) that are used to represent the data that you wish to transmit. For example, if you have four different tones, you can carry two bits of information per tone. Each of the different tones is simply a sine wave of a specific frequency, so that the modulated signal is the concatenation of a number of short snippets of sine waves. The tones are always phase continuous so that there is no discontinuity from one tone to the next. Since you have been given a stream of numbers, we will be working exclusively in the digital domain.

The version of FSK that we will be using is somewhat simplified, chosen specifically to ensure that the modulated signal can easily be kept phase continuous. The first parameter that we have to specify is the number of samples used to represent each tone, N_s. For simplicity, this value will be fixed at $N_s = 32$. This means that our base tone will complete one sinusoidal cycle in N_s samples. The equation for computing the samples of a tone, $s_0 s_1 \ldots s_{N_s-1}$, is given as

$$s_i = A \sin\left(\frac{2\pi f i}{N_s}\right) \tag{1}$$

where A represents the amplitude of the signal, and f represents the frequency of the tone. To further simplify things, you may assume that $A = 4096$. Note that it becomes more difficult to decode the signal in the presence of noise for larger values of f, so we will restrict ourselves to the set $f \in \{1, 2, 3, 4\}$, corresponding to tones 1 through 4. Here's what tone 1 looks like:

```
    0   799  1567  2275  2896
 3405  3784  4017  4096  4017
 3784  3405  2896  2275  1567
  799     0  -799 -1567 -2275
-2896 -3405 -3784 -4017 -4096
-4017 -3784 -3405 -2896 -2275
-1567  -799
```



The list of numbers above is the actual sample values s_i for tone 1, listed left-to-right, top-to-bottom. Note that the last sample is not zero. This is because equation (1) completes a cycle in $N_s + 1$ samples, rather than only N_s samples. This definition of equation (1) allows us to simply concatenate two tones

while still maintaining phase continuity. The alternative would have required us to integrate the phase over the tone transition, which we'd just as soon avoid.

Having defined the tones, it is relatively straightforward to detect the presence of a tone in an arbitrary sequence of numbers. The normalised correlation coefficient is defined as

$$\text{corr}(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\sqrt{\|\mathbf{a}\|\|\mathbf{b}\|}} \tag{2}$$

where $\mathbf{a}$ and $\mathbf{b}$ are vectors. The samples of tone 1, s_i, where $0 \leq i < N_s$, could be used as the vector $\mathbf{a}$, while the vector $\mathbf{b}$ can be formed by taking N_s consecutive values from the sequence that we are trying to decode. Note that $0 \leq \text{corr}(\mathbf{a}, \mathbf{b}) \leq 1$, with a value of 1 indicating that $\mathbf{a}$ and $\mathbf{b}$ are identical.

Let d_i denote the stream of input N samples that you wish to decode, where $0 \leq i < N$. Let $\mathbf{b}_i$ denote the subset of d_i, where

$$\mathbf{b}_i \equiv (d_i, d_{i+1}, \ldots, d_{i+N_s-1})$$

Further, let $\mathbf{t}_f$ denote the sequence

$$\mathbf{t}_f \equiv \left(A \sin\left(\frac{2\pi f \times (0)}{N_s}\right), A \sin\left(\frac{2\pi f \times (1)}{N_s}\right), \ldots, A \sin\left(\frac{2\pi f \times (N_s - 1)}{N_s}\right)\right)$$

so that $\mathbf{t}_f$ represents the samples of tone f. Note that the sequence d_i may start with an arbitrary number of "non-coding" samples, i.e. samples that are not part of any tone. We have to detect the index where the first tone starts. Assuming that the input d_i represents a noise-free FSK modulated signal, the first tone starts at d_k where k is the smallest integer such that $\text{corr}(\mathbf{b}_k, \mathbf{t}_f) = 1.0$ (for some $f \in \{1, 2, 3, 4\}$). Thus k represents the index of the start of the first detected tone. From there on, we expect the next tone to be in the vicinity of $k + N_s$, and the next at $k + 2N_s$, and so on.

In the real world, however, the signal will not be noise-free. This means that the correlation coefficient will not likely be equal to 1.0, even though we are at the correct offset k. The solution is to look for the maximum correlation value, rather than looking for a perfect correlation value of 1.0. To keep things simple, the first tone of any sequence of tones will be tone 1. This tone should be detected, but not reported, *i.e.* the first tone is used as a synchronisation pattern and does not form part of the data payload of the signal.

You may assume that the signal consists of contiguous tones after the first tone, *i.e.* there are no gaps in the signal. Finally, the signal may end with an arbitrary number of "non-coding" samples.

Your input will consist of a sequence of integers, approximately in the range -4096 to 4096, representing the samples d_i of a noisy FSK encoded signal. Due to the noise you may have some values larger than 4096, but the normalisation part of the correlation function will compensate for this. You must detect the position of the synchronisation tone (a tone 1), and decode all subsequent tones. Each tone will give you two bits of information. Thus, tone 1 encodes the binary value '00', tone 2 is '01', tone 3 is '10' and tone 4 is '11'. These bits must be grouped into bytes of 8 bits each (the most significant bit is transmitted first), which must then be displayed as an ASCII string. For this problem, the signal will always be strong compared to the noise component, so that the signal to noise ratio will be a factor of 8 or better.

Sample Input

Here's a sample input sequence, complete with some non-coding samples at the front and back. The sequence starts with a tone 1, like all signals will. This is followed by tones 2 1 3 2, thus encoding the letter 'I'.

```
   16   511  -458  -510   214       309   438   -46   393   634
  805   -67   363   -21   455      2084  2747  2342  2986  3596
 4118  4183  3993  3328  3600      2033  2346  1113   359  -346
  122 -1369 -2176 -3079 -3892     -4138 -2965 -3715 -4854 -3797
-4148 -2975 -2469 -2051  -776      -284  1605  3798  4178  3496
```

```
 3545  2743  2169    35 -2518     -3921 -4354 -3858 -3664 -1764
-2063  -335  1064  2881  3839      4313  4395  3415  1588  -452
-1928 -3392 -3883 -5045 -3315     -3823 -1648   281  1372  1703
 2261  3736  3485  4095  4126      4203  3887  3149  3301  3652
 2648  2098   242   397  -660     -1948 -1339 -2904 -3184 -3992


-3409 -3710 -3704 -4155 -3840     -2536 -2162 -1566  -586  -400
 2190  3290  4231  2740  1231     -1896 -2897 -4226 -3233  -679
  430  2201  3790  2973  2424      1041 -2005 -3737 -3105 -2457
 -227   834  2763  3490  3375      1749 -1591 -3703 -4251 -3652
-2841  -520  1838  2493  2964      5059  3739  2899  1980  -305


-2224 -2853 -3990 -3921 -2710     -3232 -2372   358  1467  2031
 3339  3968  3948  2354  1352       270 -2675 -2007 -4449 -4182
-4781 -3523 -1299 -1095   618       841   129   266
```

Sample Output

The output for the sample input given above is the string consisting of the letter 'I'. Longer input samples will yield more meaningful messages, assuming that the ETs have used the ASCII alphabet (well, they did in Independence Day!).

5^{th} Southern Africa Regional ACM International Collegiate Programming Contest

Sponsored by IBM

No-key Encryption (Green Balloon)

It is widely known that one-time pad encryption techniques are the only ones that are truly safe. At least, they are as safe as their key distribution mechanisms permit. The one time pad works by sharing a key between the sender and the receiver that is at least as long as the message; this means that if there is no repeating pattern in the key, then all statistical attacks will fail. It is called a one-time pad because any particular key is only ever used once (*i.e.* on one message), and then discarded.

To illustrate, consider the following simple encryption scheme: Given a message $m[0] \ldots m[n]$ and a key $k[0] \ldots k[n]$, the cyphertext is constructed by adding $k[i]$ to $m[i]$, or more accurately, adding $k[i]$ to the ASCII code of $m[i]$, and converting the result back to an ASCII character. Here's an example (Note that each symbol of the key is simply a digit between 0 and 9. They represent integers, not ASCII characters):

plaintext	=	`hello world`
key	=	`10362811345`
cyphertext	=	`ieorq(xpupi`

As long as the key satisfies certain statistical properties, this method is unbreakable, unless the key leaks out. Which, of course, leaves us with the main problem, namely key distribution. What if you could devise a scheme that relied on one-time pad encryption, but that did not require the receiver to have the matching decryption key?

Well, someone once thought that they had solved the problem in the following way: Bob wants to send Alice a message, so he encrypts the message using his one-time pad. This is written as $E_B(m)$, where $E_B(\cdot)$ means Encrypt using Bob's key, and it was applied to the message m. This means that Alice receives $E_B(m)$, but has no way to decode it. So Alice just encrypts the message using her own key by applying $E_A(\cdot)$, and sends it back to Bob, that is, Bob receives $E_A(E_B(m))$, or $(E_A \circ E_B)(m)$. Now Bob removes his key by decrypting this message with his own decryption key, so that the result is $(D_B \circ E_A \circ E_B)(m)$, where $D_B(\cdot)$ means Decrypt using Bob's key. If a suitable encryption algorithm is used, these processes commute, thus

$$(D_B \circ E_A \circ E_B)(m) = (D_B \circ E_B \circ E_A)(m) = E_A(m)$$

because $D_B(\cdot)$ cancels the effect of $E_B(\cdot)$. Now Bob has his own message back, but it is encrypted using Alice's key. This means it is safe for Bob to send it back to Alice, since only Alice can decrypt it. Thus, Alice receives $E_A(m)$, and applies her decryption key, so that she ends up with $(D_A \circ E_A)(m) = m$, the original message that Bob sent.

There is a subtle flaw in this technique. Your task will be to spot the flaw, and exploit it to decrypt the message without explicitly being given the key. But first, here's two encryption algorithms with the desired commutative property:

XOR encryption: Each cyphertext character $c[i]$ is obtained by applying

$$c[i] = m[i] \text{ XOR } k[i].$$

Note that this can be done in C or Java using `c[i] = m[i] ^ k[i]`. The XOR operation commutes with itself, so to decrypt the message we see that

$$c[i] \text{ XOR } k[i] \quad = \quad (m[i] \text{ XOR } k[i]) \text{ XOR } k[i]$$

$$\begin{aligned} &= m[i] \text{ XOR } (k[i] \text{ XOR } k[i]) \\ &= m[i] \end{aligned}$$

since a XOR $a = 0$, and a XOR $0 = a$. Thus we can decrypt a message by encrypting it a second time using the exact same key.

Addition in $\mathcal{Z}_{256}$: Each cyphertext character $c[i]$ is obtained by applying

$$c[i] = (m[i] + k[i]) \text{ modulo } 256$$

Note that this can be done in C or Java using `c[i] = (m[i] + k[i]) % 256`. The decryption process works by subtracting the key, or, in $\mathcal{Z}_{256}$, adding its additive inverse, thus

$$\begin{aligned} (c[i] + (256 - k[i])) \text{ modulo } 256 &= (m[i] + k[i] + (256 - k[i])) \text{ modulo } 256 \\ &= (m[i] + 256) \text{ modulo } 256 \\ &= m[i] \end{aligned}$$

In other words, the decryption key d is computed as $d[i] = (256 - k[i])$ modulo 256. If this decryption key is added modulo 256 to encrypted message, the result is the original message.

You know that Bob and Alice always use one of these two algorithms. They will either both use XOR, or both use Addition in $\mathcal{Z}_{256}$. They have some external method for deciding which one they will use, and you simply don't have access to that information. All you can do is eavesdrop on all the messages that they exchange.

Your program will receive three messages as input: $E_B(m)$, $(E_A \circ E_B)(m)$, and $(D_B \circ E_A \circ E_B)(m)$. You have to discover a way of decoding the original message m. Since you don't know in advance which algorithm they used, you will have to determine that on the fly. The only thing that you do know is that the message m consists only of letters of the alphabet, plus the punctuation symbols `<space>`, '.' and ','.

Sample Input

Ignoring white spaces, your input will consist of an arbitrary number of records, each record conforming to the following format:

$$\begin{array}{llll} < n > & & & \\ E_B(m)[0] & E_B(m)[1] & \ldots & E_B(m)[n-1] \\ (E_A \circ E_B)(m)[0] & (E_A \circ E_B)(m)[1] & \ldots & (E_A \circ E_B)(m)[n-1] \\ (D_B \circ E_A \circ E_B)(m)[0] & (D_B \circ E_A \circ E_B)(m)[1] & \ldots & (D_B \circ E_A \circ E_B)(m)[n-1] \end{array}$$

where $< n >$ denotes the length of the original message. This is followed by the three encrypted versions of the message, where each character is encoded as a decimal integer in the range 0–255. Here's a concrete example:

```
12
  28  13   5  31  79  73   4  79  16   3   6  93
 110 108 107 123  32  36  36  59 117 123 114 125
  26   4   2   8   0  77  87  27  23  20  16  14
```

The original messages have a maximum length of 4096 characters. The input may contain any number of records in this format.

Sample Output

Your output will be the decoded message, followed by a newline. You must convert the numeric values of the decoded message back into ASCII characters, so that the result is a printable string. The example above would yield the following output:

```
hello world.
```

5^{th} Southern Africa Regional ACM International Collegiate Programming Contest

Sponsored by IBM

The Answer (White Balloon)

All educated people know that the answer to 'Life, the universe, and Everything', is, of course, 42. The problem, according to the late Douglas Adams, is to understand the question. You will now attempt to come up with a suitable question to which the answer is 42. (Or the white mice will take your brain home in a jar.)

You will construct your questions by forming expressions using the first N integers in sequence, such that your expression will be of the form

$$1 \quad \square \quad 2 \quad \square \quad 3 \quad \square \quad 4 \quad \square \quad \ldots \quad \square \quad N$$

Each of the boxes represent an operator, which can be any one of the following:

Symbol	Meaning
`|`	concatenate the digits, *i.e.* `2 | 3 = 23`
`#`	delete number immediately to the left, *i.e.* `1 + 2 # 3 = 1 + 3`
`+`	add left and right numbers, *i.e.* `1 + 2 = 3`
`-`	subtract right from left number, *i.e.* `1 - 2 = -1`
`*`	multiply left and right numbers, *i.e.* `3 * 4 = 12`

Operator precedence is as follows:

Priority	Symbol(s)	Associativity
first	`#`	unary left
second	`|, *`	binary, left-to-right
third	`+, -`	binary, left-to-right

Here's an example sequence, illustrating the operator precedence. Parentheses are used to illustrate which subexpression will be evaluated during this step:

```
  1 * 2 * 3 * 4 * 5 - (6 # 7) | 8
```

First, the delete operators are applied, resulting in

```
  1 * 2 * 3 * 4 * 5 - 7 | 8
```

Next, the multiplies and concatenation operators are evaluated, in left to right order, yielding

```
  (1 * 2 * 3 * 4 * 5) - (7 | 8)
= 120 - 78
```

Lastly, the addition and subtraction operators are evaluated, in left to right order, resulting in the desired answer, 42.

Here's another one, this time it doesn't result in an answer of 42.

```
    1 - 2 # 3 * 4 + 5 | 6 - 7 # 8
=   1 - 3 * 4 + 5 | 6 - 8
=   1 - 12 + 56 - 8
=   37
```

Sample input

Your input will consist of several integers N such that $2 \leq N \leq 12$. You must construct and evaluate all the possible expressions (as defined above) using the first N (inclusive) integers. Example input:

```
4 5
```

Sample output

Your output will be the number of unique questions resulting in the answer 42. For the sample input above, your output will be:

```
0
1
```