Contents

1 Animal Zones

Gloria the Hippo and Melman the Giraffe are shocked to find they have been asked to design a new zoo, where each animal species will live in its own rectangular *animal zone*. They decide to have different companies design each animal zone. All companies submit coordinates for their animal zone, and construction begins when zones do not overlap. Gloria and Melman hope this takes a very long time to work out.

You have been hired to write a program that analyzes zone boundaries and reports to Gloria and Melman whether any animal zones overlap. Two zones are considered to overlap if one contains part of the other, if they share a portion of a common side, or if they share a common corner.

Each company will submit the (x,y) coordinates of the lower left and top right corners of their animal zone, where (0,0) indicates the bottom left corner of the zoo. You may assume that all zones fit within a 20 by 20 area (coordinates range between 0..19).

Input/Output Format:

The input will consist of a line containing the number of animal zones, followed by one line per zone in the format: x1 y1 x2 y2 where (x1,y1) is the lower left-hand corner of the rectangle and (x2,y2) is the upper right-hand corner of the rectangle. The output will be "Overlap" if any zones overlap, otherwise "No Overlap".

Example:

Sample Input 1:

```
3
1 1 2 2
2 5 4 7
4 2 7 3
```

Sample Output 1:

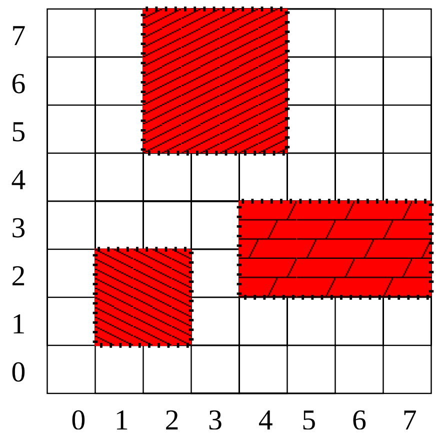
```
No Overlap
```

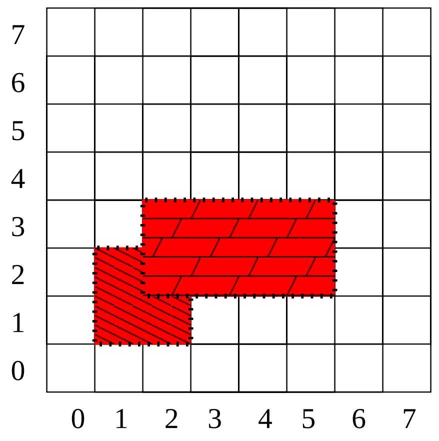
Sample Input 2:

```
2
1 1 2 2
2 2 5 3
```

Sample Output 2:

```
Overlap
```





2 Decoding Roman Numerals

King Julien of the Lemurs tells Gloria the Hippo and Melman the Zebra about a community of Geckos. These Geckos do not use the same numbering system that the Lemurs use. When Gloria meets the Geckos, their leader Geckus Leaderus explains that they use Roman Numerals. Help Gloria convert the Roman Numerals into decimal (base ten) numerals.

Recall that Roman Numerals work as follows: Each numeral-character (I,V,X,L,C) has a single value: I=1, V=5, X=10, L=50, C=100. The value of a Roman Numeral is the sum of its numeral-characters *unless* a smaller numeral-character is placed in front of a larger numeral-character, in which case the smaller numeral-character is subtracted from the total. For instance, VI=5+1, while IV=5-1. You may assume that at most one subtraction is required per number.

Input/Output Format:

Inputs are in the form of text strings for Roman numerals, separated by spaces on a single line. Output must a single line for each numeral in the format ""X translated to Y", where X is the string for the Roman numeral and Y is its decimal value.

Example:

Sample Input 1:

```
I   V   X
```

Sample Output 1:

```
I translated to 1
V translated to 5
X translated to 10
```

Sample Input 2:

```
VI   IV   CLII
```

Sample Output 2:

```
VI translated to 6
IV translated to 4
CLII translated to 152
```

3 Cyclic Redundancy Codes

Alex the Lion has been sending messages from Madagascar to his friends back at the zoo by tapping into a satellite network. Unfortunately solar flares have been disrupting his messages by changing some data during transmissions. Trying to discover why his messages are being garbled, Alex finds out that data that traverses a network consists of ones and zeroes grouped into bytes that represent characters, pictures, music samples, etc. When these ones and zeroes are encoded on wires or into radio waves, sometimes they don't come back out the same way they went in.

Fortunately, a solution exists for detecting these errors. To check if the same numbers came out as went in, a few more numbers (error-detecting codes) can be added to each message.[1] Alex decides to use Cyclic Redundancy Code (CRC), an error-detecting code based on polynomial division. To build a message with a CRC, you'll have to *divide* the bits in the message by a *polynomial* to calculate the remainder: the remainder is appended to the message. Checking a CRC requires dividing the message with the remainder. If the message wasn't corrupted, the new remainder is zero!

Your task is to build a CRC generator and checker for Alex to check pictures of Madagascar being sent back to his friends stuck in the zoo. The first line of input will be the coefficients of a binary polynomial in ones and zeroes. This will be the divisor in the division.

```
1 0 0 1 1
```

The bit string 10011 means that the polynomial is $x^5 + x + 1$. The second bit string will be the message. You will need to construct the dividend by left-shifting the bit string by $n - 1$ positions, where n is the number of digits in the divisor. You can do this by adding $n - 1$ zeros to the end of the message. This will allow you to calculate a $n - 1$ digit remainder.

```
1 1 0 1 0 1 1 0 1 1    ->    1 1 0 1 0 1 1 0 1 1 0 0 0 0
                                                 ^ ^ ^ ^
```

Your goal is to compute and print the *remainder*. It is the remainder that has fixed length; here, the remainder will be only four bits long. The *quotient* does not matter at all: it is as long as the original message! Your output line will be the correct remainder.

```
1110
```

How did this become the remainder? Note that this isn't plain-old integer division with really big integers: the value above in decimal is $512 + 256 + 64 + 16 + 8 + 2 + 1 = 859$, and $859 \bmod 19 = 4$. The remainder is 14 because the CRC runs over a binary field with *no carry*:

$$\begin{aligned} 1 - 1 &= 0 \\ 0 - 1 &= 1 \\ 1 - 0 &= 1 \\ 0 - 0 &= 0 \end{aligned}$$

[1]The simplest possible error-detecting code is to send all your data twice. Obviously, this would be inefficient, and not all that effective, because the same bit might get lost twice! Two main classes of error-detecting code are used in the Internet: the checksum, which uses addition, and the Cyclic Redundancy Code (CRC), which uses division.

The checksum is pretty easy…too easy for a programming contest. To build a message that has a checksum, you would add up all the bytes in a message, make the sum negative, and put the value into the message. Then, to check the message, you would add up all the bytes including the checksum to see if zero is the result!

Unfortunately, the checksum is vulnerable to simple changes: $5 + 3 = 4 + 4 = 2 + 6$, so something more complicated and less predictable is often used.

This means that computing the division is even easier! Check out the following example. The remainder will be one bit shorter than the divisor. The dividend is extended by as many zeroes as will be in the remainder.

$$
\begin{array}{ccccc|cccccccccccccc}
 & & & & & & & & & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ \cline{6-19}
1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
 & & & & & -1 & 0 & 0 & 1 & 1 & \downarrow & & & & & & & & \\ \cline{6-10}
 & & & & & & 1 & 0 & 0 & 1 & 1 & & & & & & & & \\
 & & & & & & -1 & 0 & 0 & 1 & 1 & \downarrow & & & & & & & \\ \cline{7-11}
 & & & & & & & 0 & 0 & 0 & 0 & 1 & & & & & & & \\
 & & & & & & & -0 & 0 & 0 & 0 & 0 & \downarrow & & & & & & \\ \cline{8-12}
 & & & & & & & & 0 & 0 & 0 & 1 & 0 & & & & & & \\
 & & & & & & & & -0 & 0 & 0 & 0 & 0 & \downarrow & & & & & \\ \cline{9-13}
 & & & & & & & & & 0 & 0 & 1 & 0 & 1 & & & & & \\
 & & & & & & & & & -0 & 0 & 0 & 0 & 0 & \downarrow & & & & \\ \cline{10-14}
 & & & & & & & & & & 0 & 1 & 0 & 1 & 1 & & & & \\
 & & & & & & & & & & -0 & 0 & 0 & 0 & 0 & \downarrow & & & \\ \cline{11-15}
 & & & & & & & & & & & 1 & 0 & 1 & 1 & 0 & & & \\
 & & & & & & & & & & & -1 & 0 & 0 & 1 & 1 & \downarrow & & \\ \cline{12-16}
 & & & & & & & & & & & & 0 & 1 & 0 & 1 & 0 & & \\
 & & & & & & & & & & & & -0 & 0 & 0 & 0 & 0 & \downarrow & \\ \cline{13-17}
 & & & & & & & & & & & & & 1 & 0 & 1 & 0 & 0 & \\
 & & & & & & & & & & & & & -1 & 0 & 0 & 1 & 1 & \downarrow \\ \cline{14-18}
 & & & & & & & & & & & & & & 0 & 1 & 1 & 1 & 0 \\
 & & & & & & & & & & & & & & -0 & 0 & 0 & 0 & 0 \\ \cline{15-19}
 & & & & & & & & & & & & & & & \mathit{1} & \mathit{1} & \mathit{1} & \mathit{0}
\end{array}
$$

Input/Output Format:

The input will consist of two lines. The first line of input will be the 0-1 coefficients of a binary polynomial, separated by spaces. The second line will be the message in ones and zeros, separated by spaces. The output will be a single line containing the correct remainder in ones and zeros, separated by spaces.

Example:

Sample Input 1:
1 0 1 0 0 0
Sample Output 1:
0

Sample Input 2:
1 0 0 1 1 1 1 0 1 0 1 1 0 1 1
Sample Output 2:
1 1 1 0

4 T9-style Text Messaging

Melman the Giraffe misses his friends back at the zoo. The penguins inform him that they have a cell phone they use to contact friends in New York. Melman is excited, but he finds that it's impossible for him to hold the phone to his ear because his neck is so long. So the penguins suggest that he try text messaging. Melman can do this, but he finds that writing text messages with his hooves still takes a long time.

Melman hears of a system called T9 that would make it easier to write text messages. Melman's phone keypad looks like this:

1	ABC 2	DEF 3
GHI 4	JKL 5	MNO 6
PQRS 7	TUV 8	WXYZ 9
∗	0	♯

Without T9, Melman uses the following system to enter characters:

action (without T9)	letter displayed
press 2 once	A
press 2 twice	B
press 2 three times	C
press 3 once	D
press 3 twice	E
press 3 three times	F
press 4 once	G
press 4 twice	H
press 4 three times	I
and so on...	

So to enter GIRAFFE, Melman must press 4444777233333333.

Melman hears that with T9-style messaging, he would only have to press one button for any letter. That means faster texting! Here is how it would work:

To enter GIRAFFE, Melman would only have to press one button for each letter. So Melman would press 4472333. The first digit, 4, could stand for either G, H, or I. Similarly, the third digit, 7, could stand for P, Q, R, or S. Given 4472333, the phone guesses that the most likely word to be created out of {G,H,I}{G,H,I}{P,Q,R,S}{A,B,C}{D,E,F}{D,E,F}{D,E,F} is G-I-R-A-F-F-E.

To determine which words are more likely than other words, the phone needs something called a frequency table. The table lists words. For each word, there is a count. The higher the count, the more common the word is.

Of course, sometimes you really want the less common word. In that case, users will hit a button that tells the phone to "try again."

Given a frequency table, write a program that implements T9-style messaging for Melman. Your implementation will interpret one word at a time. To tell your system to "try again," Melman will enter a zero.

Input/Output Format:

Input: Input will consist of a frequency table, followed by a line of 5 asterisks, and finally followed by a series of numeric strings. Each numeric string corresponds to a word. Assume all strings will consist of lower-case symbols.

Output: If a numeric string does not end in a zero, then the most common word that corresponds to the numeric string should be output. If a numeric string ends in i zeroes, where i > 0, then the (i+1)th common word should be output.

All words should be output on the same line.

Example:

Sample Input 1:

```
i 8
am 2
art 1
*****
4 26 278
```

Sample Output 1:

```
i am art
```

Sample Input 2:

```
i 8
that 7
is 6
an 4
am 2
artist 1
*****
4 260 26 278478
```

Sample Output 2:

```
i am an artist
```

5 Spy Hard with a Vengeance

The penguins, Skipper, Rico, Kowalski, and Private, who consider themselves to be *super spies*, start exploring Madagascar as soon as they arrive there from Antarctica. During their reconnaissance, they run across a locked door deep inside the island. There is no key to open the door, but there is a small altar in front of the door, and the inscription on a nearby wall tells them that they need to fill up a water jug with exactly 10 gallons of water, and put it on the altar. Though there is a stream nearby, the only jugs that are lying there are of sizes 4, 7 and 13 gallons. Rico suggests they put the 13 gallon jug on the altar and start adding water slowly till the door opens. This plan is however shot down by Kowalski, who notices further inscription that informs them that they have exactly one chance at trying to open the door, and if they get it wrong, the door would stay locked for another year. At this point, Skipper barks at Kowalski to find a way to achieve this, and goes off to find something to eat.

Kowalski realizes that to achieve his objective, he will have to perform a series of actions using the jugs. He first lists out the possible actions:

- Fill up one of the jugs with water.
- Empty one of the jugs.
- Pour from one jug into other till either the former is empty or the latter jug is full.

You are to write a program to help Kowalski in solving this and similar such problems. As an example, the following sequence of actions will solve the instance specified above:

1. Fill up the 13 gallon jug.
2. Pour 7 gallons from 13 gallon jug into the 7 gallon jug (at which point the 7 gallon jug would be full). This leaves us with 6 gallons in the 13 gallon jug.
3. Fill up the 4 gallon jug.
4. Pour from the 4 gallon jug into the 13 gallon jug.

At this point, the 13 gallon jug contains exactly 10 gallons of water. In general there may be many different ways to do such a task. You are required to write a program that finds the *minimum amount of water required to do the task* (in the above case, there exists a way to do it by using exactly 10 gallons of water).

Input/Output Format:

The input will contain several test cases, each on a single line. Each test case consists of 4 positive integers, the first 3 integers specifying the capacities (sizes) of the jugs that can be used (assume there are always exactly 3 jugs that can be used), and the fourth number specifying the *target* that needs to be achieved. Assume that the target is always less than or equal to the size of the largest jug. You can also assume that the capacities of the jugs are always less than 25.

If it is possible to achieve the goal, you must output the *minimum amount of water* that is needed (from the stream) in order to do so. For the above scenario, your output would be in the following format:

Given jugs with capacities 4, 7 and 13, target 10 is achievable using 17 gallons of water.

If the target is not achievable, you must output a message in the following format:

Given jugs with capacities 4, 7 and 13, target 10 is not achievable.

Sample Input 1:

```
4 9 13 1
4 9 13 2
4 9 13 5
4 9 13 7
4 9 13 10
```

Sample Output 1:

```
Given jugs with capacities 4, 9 and 13, target 1 is achievable using 9 gallons of water.
Given jugs with capacities 4, 9 and 13, target 2 is achievable using 10 gallons of water.
Given jugs with capacities 4, 9 and 13, target 5 is achievable using 9 gallons of water.
Given jugs with capacities 4, 9 and 13, target 7 is achievable using 11 gallons of water.
Given jugs with capacities 4, 9 and 13, target 10 is achievable using 10 gallons of water.
```

Sample Input 2:

```
4 6 8 4
4 6 8 7
4 6 8 11
```

Sample Output 2:

```
Given jugs with capacities 4, 6 and 8, target 4 is achievable using 4 gallons of water.
Given jugs with capacities 4, 6 and 8, target 7 is not achievable.
Given jugs with capacities 4, 6 and 8, target 11 is not achievable.
```

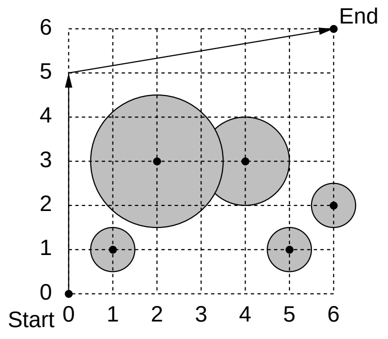
6 Penguins on Ice

The penguins have commandeered a ship, but in order to reach Madagascar they need to avoid a number of icebergs floating in the ocean. They have a map that indicates the locations of the icebergs as circles. Your job is to help them find the shortest path from the start to the goal, while avoiding the icebergs. Since the penguins have limited navigational skills, they can make *at most one turn* along the way, and this turn *must be at a grid vertex.*

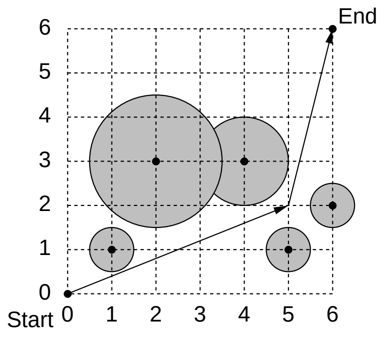
More formally, let *size* denote the height and width of the map. The set of *grid vertices* are given by (x, y) coordinates, where x and y are integers, where $0 \leq x, y \leq size$. The starting vertex is $(0, 0)$ and the ending vertex is $(size, size)$. Each iceberg is described by three numbers, its center coordinates (c_x, c_y) (integers) and its radius r (a double). The path taken by the ship is *valid* if it satisfies the following requirements:

- It starts at $(0, 0)$ and ends at $(size, size)$.
- It makes at most one turn at a grid vertex. (That is, the coordinates are integers in the range 0 to *size*.)
- It does not intersect any iceberg, that is for every point (x, y) along the path and for every iceberg we have $dist((x, y), (c_x, c_y)) > r$, where $dist((x, y), (c_x, c_y)) = \sqrt{(x - c_x)^2 + (y - c_y)^2}$.

  Hint: To compute the square root of a number z, use `Math.sqrt(z)`.



Valid path turning at (0,5)
of length 11.0827...



Shorter valid path turning at (5,2)
of length 9.5082...

Given the size of the grid and a set of obstacles, your task is to compute the shortest valid path.

Input/Output Format

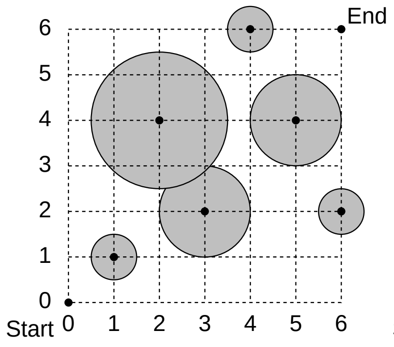
We have provided a main program for you that inputs the grid size and the center coordinates and radii of icebergs. It stores these values in the following global variables.

```
static int size;           // domain size
static int n;              // number of obstacles
static int[] obstX;        // obstacle x-coordinates
static int[] obstY;        // obstacle y-coordinates
static double[] obstRad;   // obstacle radii
```

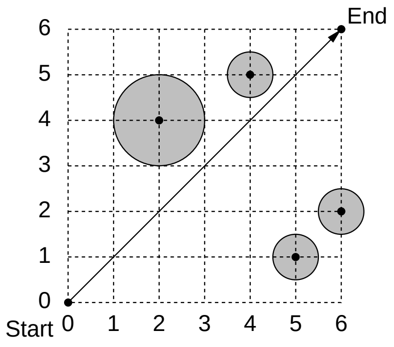
All you need to do is to provide the body for the function `findTurningPoint`, which uses the above values to compute the turning point of the shortest path. It has the following prototype:

```
private static int[] findTurningPoint();
```

Once you have computed the optimum turning point (x, y) in your `findTurningPoint`, this value can be returned by creating a *new* 2-element integer array, storing these coordinates in the array, and returning the array. If no turn is needed, return $(0, 0)$. If no valid path exists, return $(-1, -1)$. Examples of these special cases are shown in the figure below.



Turn at: (−1,−1)
(no solution)



Turn at: (0,0)
(direct path possible)

On return your program *must* call the function `validateTurningPoint`. The argument is the 2-element array you returned from `findTurningPoint`. This function tests the correctness of your answer and outputs a message either indicating that your result is correct or why your result is wrong.

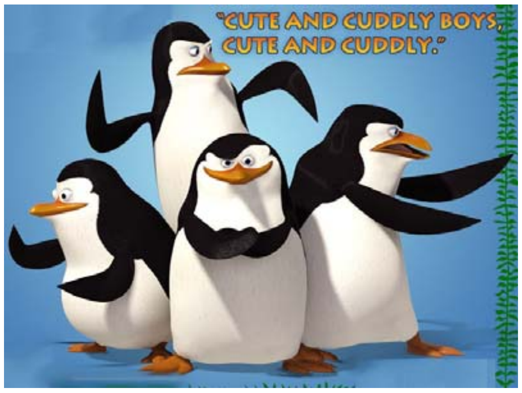
A sample input and output are shown below. (If you use our skeleton code, you do not need to do any input or output.) The first line of the input contains the grid size and the second line is the number of obstacles n. This is followed by n lines, containing the center coordinates and radii of the icebergs. You may assume that the iceberg center coordinates all lie on the grid vertices (e.g. between 0 and *size*) and that the radii are all positive. Note that the icebergs may extend outside the grid, as shown in the figure.

Sample Input 1:

```
6
5
1   1   0.5
2   3   1.5
4   3   1.0
5   1   0.5
6   2   0.5
```

Sample Output 1:

```
Grid size: 6
Number of obstacles: 5
  (1, 1) radius = 0.5
  (2, 3) radius = 1.5
  (4, 3) radius = 1.0
  (5, 1) radius = 0.5
  (6, 2) radius = 0.5
Your program returned the turning point (5, 2).
Correct.
```



7 Ramsey's Parties

When Alex the Lion, Marty the Zebra, Melman the Giraffe, and Gloria the Hippopotamus arrive at Madagascar, the Lemurs are overjoyed and host a series of parties ostensibly in honor of them (not that they needed a reason). Our heroes however don't know too many people at these parties, and mostly stick together watching the hosts celebrate. At one of these parties, Marty counts the number of lemurs to be 18, and remarks: "You know, considering how many lemurs are here, I bet there is a group of at least 9 of them who all know each other". Melman doesn't buy this, and claims that since there are a fair number of lemurs who clearly don't know each other, there is a group of at least 9 lemurs, none of whom know each other. At this point, Gloria, the smart one, intervenes and tells them that, according to Ramsey theory (named after Mathematician Frank Ramsey), it is only guaranteed that there would be a group of either 4 lemurs who *all* don't know each other[2], or 4 lemurs who *all* know each other[3]; further if there were fewer than 18 lemurs at a party, even this cannot be guaranteed. As usual, Marty and Melman don't believe her, and decide to test it out for themselves by attending as many parties as they can. You are to help Marty and Melman in their quest to prove or disprove the Ramsey theorem.

Input/Output Format:

The first line of the input will contain the number of lemurs at a party (N), and a positive integer K ($< N$). We will assume that the lemurs are numbered from 0 to $N-1$. The next N lines of the input will contain the information about who knows who. The first line contains a list of lemurs that lemur 0 knows, the second line contains a list of people that lemur 1 knows, and so on. This is a symmetric relationship; so if X knows Y, then Y knows X. Each line ends with a -1.
An example follows:

```
5 3        - 5 lemurs at the party, and K = 3.
1 2 -1     - lemur 0 knows lemur 1 and lemur 2.
0 3 5 -1   - lemur 1 knows lemur 0, lemur 3 and lemur 5.
0 3 5 -1   - lemur 2 knows lemurs 0, 3, and 5.
1 2 -1     - lemur 3 knows lemurs 1 and 2.
1 2 -1     - lemur 4 knows lemurs 1 and 2.
```

Your task is to find the *first* group of K lemurs (in lexicographical order)[4] such that they all know each other, or no two of them know each other. You are to output this group on one line. Assume N is less than or equal to 20, and K is less than or equal to 7. If there is no such group of K lemurs, the output should simply contain a line:

```
No such group exists.
```

For the example input above, possible such groups are:

Groups where everyone knows each other: $(0,1,2)$, $(1,2,3)$, and $(1,2,4)$.
Groups where no one knows each other: $(0,3,4)$.

The first of these in the lexicographical order is $(0,1,2)$ and the output will be:

```
Lemurs 0, 1, 2 know each other.
```

If instead the answer was a group of lemurs who didn't know each other, you should output:

```
Lemurs 0, 1, 2 don't know each other.
```

[2]An independent set

[3]A clique

[4]$(x_1, x_2, \ldots, x_K)$ appears before $(y_1, y_2, \ldots, y_K)$ in lexicographical order, if (1) $x_1 < y_1$, **or** (2) $x_1 = y_1$ and $x_2 < y_2$, **or** (3) $x_1 = y_1$ and $x_2 = y_2$ and $x_3 < y_3$ and so on.

Sample Input 1:

```
5 3
3 -1
4 -1
4 -1
0 4 -1
1 2 3 -1
```

Sample Output 1:

```
Lemurs 0, 1, 2 don’t know each other.
```

Sample Input 2:

```
6 4
1 4 -1
0 2 -1
1 3 4 -1
2 5 -1
0 2 -1
3 -1
```

Sample Output 2:

```
No such group exists.
```

Sample Input 3:

```
18 6
1 3 4 5 7 8 9 12 -1
0 2 3 4 5 6 8 9 11 12 13 15 16 17 -1
1 5 7 8 10 11 13 15 16 -1
0 1 4 5 6 9 10 12 13 15 -1
0 1 3 5 6 10 11 12 14 15 17 -1
0 1 2 3 4 6 11 12 15 -1
1 3 4 5 8 9 10 11 13 14 15 -1
0 2 8 11 12 13 14 16 17 -1
0 1 2 6 7 12 17 -1
0 1 3 6 15 16 -1
2 3 4 6 11 13 14 16 -1
1 2 4 5 6 7 10 12 13 15 17 -1
0 1 3 4 5 7 8 11 13 15 17 -1
1 2 3 6 7 10 11 12 15 -1
4 6 7 10 -1
1 2 3 4 5 6 9 11 12 13 16 17 -1
1 2 7 9 10 15 -1
1 4 7 8 11 12 15 -1
```

Sample Output 3:

```
Lemurs 0, 1, 3, 4, 5, 12 know each other.
```

8 Making Change for Melman

Melman the Giraffe wants to buy a tissue. He asks Alex the Lion for the 26 cents he needs. Alex asks him how he wants it. Melman considers how many ways there are to make 26 cents using pennies, nickels, dimes, and quarters. He realizes that there are 13 ways to do this:

$$\begin{array}{rll}
1: & 26 = & 25+1 \\
2: & 26 = & 10+10+5+1 \\
3: & 26 = & 10+10+1+1+1+1+1+1 \\
4: & 26 = & 10+5+5+5+1 \\
5: & 26 = & 10+5+5+1+1+1+1+1+1 \\
6: & 26 = & 10+5+1+1+1+1+1+1+1+1+1+1+1 \\
7: & 26 = & 10+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 \\
8: & 26 = & 5+5+5+5+5+1 \\
9: & 26 = & 5+5+5+5+1+1+1+1+1+1 \\
10: & 26 = & 5+5+5+1+1+1+1+1+1+1+1+1+1+1 \\
11: & 26 = & 5+5+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 \\
12: & 26 = & 5+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1 \\
13: & 26 = & 1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+ \\
 & & \quad 1+1+1+1+1+1
\end{array}$$

Mason the Chimpanzee realizes that if they go to Madagascar the answer might not be right anymore since they use a different currency. However, Mason does not know what currency they use. **He needs your help!**

Write a program to solve the following problem: Given a currency system $(a_1, a_2, a_3, \ldots, a_k)$ (that is, there are coins worth a_1 cents, a_2 cents, …, a_k cents) and given a number N, output the number of ways to obtain N cents using that currency system. Note that we only want to know *how many* ways there are to make change, not what those ways are. (Some of our test inputs involve very high numbers of combinations, so it will not be possible to do this by brute-force enumeration.)

Input/Output Format

We have provided a main program for you that inputs the coefficients and stores them in an array, and prints the final output. All you need to do is to provide the body for the function `countCombinations` which computes the number of combinations and returns this value. The function has the following prototype:

```
private static int countCombinations(int[] coins, int total)
```

where

coins is an array containing the coin values in increasing value. (The number of coins can be computed by `coins.length`.) In the above example, this array would contain $[1, 5, 10, 25]$.

total is the desired total value. In the above example this would be 26.

If you want to do your own input and output, the input values are entered on a single line, separated by white space. The first number is the number of coins, followed by the various coin values (in increasing order), and finally the desired total amount. The output consists of echoing the input values and outputting the number of combinations.

You may assume that the coin values and total value are all positive. You may assume that the given coin values are distinct from one another. Note that it may **not** be possible to generate the given total with the given coin values (and if so, the answer is 0). Some examples are shown below:

Sample Input 1:

```
4   1  5 10 25  26
```

Sample Output 1:

```
Coins: 1 5 10 25
Total: 26
Number of combinations: 13
```

Sample Input 2:

```
3   2 10 11      22
```

Sample Output 2:

```
Coins: 2 10 11
Total: 22
Number of combinations: 4
```

Sample Input 3:

```
3   2 10 11      9
```

Sample Output 3:

```
Coins: 2 10 11
Total: 9
Number of combinations: 0
```

Practice 1 Zebra Poker

Marty the Zebra and his friends are playing poker on the boat when he suggests they play a game called Razz. In Razz Each player receives 7 cards. Instead of trying to get the highest 5-card poker hand, the winner is the player with the *lowest* 5-card hand (using any 5 cards out of the 7 cards the player holds). Straights (5 cards in sequence) and flushes (5 cards of the same suit) do not affect the low hand, only pairs. Aces are treated as low cards.

Since Marty is a very timid poker player, he decides to bet only when he has the best possible Razz hand, which is Ace, 2, 3, 4, and 5 (in any order). Your job is to help Marty determine whether he has this 5-card hand out of his 7 cards.

Input/Output Format:

Cards are treated as numbers, with Aces, Kings, Queens, and Jacks represented as 1, 13, 12, 11, respectively. 7-card hands are read in as 7 numbers on each line. For every hand, your program should output on each line "Bet" if the lowest possible 5-card hand is possible, and "Fold" if it is not.

Example:

Sample Input 1:

```
1 2 3 4 5 6 7
1 2 3 4 6 7 8
1 2 3 3 4 4 4
11 1 5 4 12 2 3
5 1 3 2 6 7 9
```

Sample Output 1:

```
Bet
Fold
Fold
Bet
Fold
```

Practice 2 Lion Encoder

Alex the Lion is in Madagascar and decides he wants to send a message to his friends back at the zoo. Tapping into a satellite network he find he can send digital messages electronically, but bandwidth is limited. Doing some research, Alex finds out about about Run-Length Encoding (RLE), a simple form of data compression in which runs of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run.

Your job is to help Alex build a run-length encoder for the following byte-oriented run-length encoded format. Runs of the same, repeated byte in the original file are replaced with a count of the number of repeated bytes, stored as a byte, followed by the repeated byte itself. For example, the string "HELLO", would be encoded as follows:

original	H	E	L	L	O			
compressed (encoded)	1	H	1	E	2	L	1	O

Each letter is actually the ascii value of the character. (For example 'h' is 0x68 in base 16 or 104 in base 10). Note that every other byte is a count of how often to repeat the byte that follows.

Note that there is no reason to use a count of zero (0), so we can assume a count of 0 actually represents 256. Also note that if you receive more than 256 letters in a row, you will need to use multiple counts since bytes only store values up to 255. In such cases, all but the final count values should 0. See the example below, which involves encoding a string of 1000 H's.

original	H	H	H	H	H	H	H	...
compressed (encoded)	0	H	0	H	0	H	232	H

Example:

Sample Input 1:
H E L L O

Sample Output 2:
1 H 1 E 2 L 1 O

Sample Input 2:
H ... (1000 times)

Sample Output 2:
0 H 0 H 0 H 232 H