CCSC-CP Programming Contest

April 10, 2021

Official Problem Set

A: Ambiguous Codes

Some codes have variable length representations of characters. For example, Morse Code uses 1 to 6 'dits' and 'dahs' for the characters it represents. In the figure below they are replaced with 0's and 1's for programming convenience. When messages are constructed, it is necessary to separate consecutive characters with some additional space so that there is no confusion where they begin and end. For example, 'BUT' should be shown as 1000 001 1 rather than 10000011 because the latter could also be parsed as 10 00 00 11 which would spell 'NIIM'. Morse Code is therefore considered to be *ambiguous*.

International Morse Code

A	01	N	10	1	01111
B	1000	O	111	2	00111
C	1010	P	0110	3	00011
D	100	Q	1101	4	00001
E	0	R	010	5	00000
F	0010	S	000	6	10000
G	110	T	1	7	11000
H	0000	U	001	8	11100
I	00	V	0001	9	11110
J	0111	W	011	0	11111
K	101	X	1001	.	010101
L	0100	Y	1011	,	110011
M	11	Z	1100	?	001100

Your task is to examine a binary represented code, such as Morse code, and report if the code is definitely unambiguous. That is, in all cases of reading a continuous string of 0's and 1's (with no extra spacing between characters) of a message, one can immediately determine where each character ends (and consequently where the next character begins).

In the first sample input, you may observe that the code really is ambiguous because you could parse the string 10010011 as 1001 0011 or 1 0 0 1 0 0 1 1 or 1 0 0 1 0011 etc. In the second sample input, you could parse the string 00110011 as 00110 0011 or start with 0011 00..., but the third 0 in a row is a dead give-away that what may appear to be ambiguous, a little searching ahead shows that it is really unambiguous. Since it is not *immediately* obvious where the (first) character ends, simply allow that it *might* be ambiguous. The third sample input never has the code of one character be the prefix of the code for another character, and thus there can be no ambiguity for where one character ends and another begins. Simply said, the prefix property is necessary but not sufficient to confirm ambiguity.

Input

There may be multiple codes to examine. Each code will have from 1 to 39 characters that it can represent (note that Morse Code has 39 characters.) Each character represented in the code will have from 1 to at most 8 bits. Each code will be presented on multiple lines with the first line containing the number of characters, which is then followed with one code item per line. Following the last code will be a line with a single 0 which represents the end of the data rather than a new code with no characters.

Output

The output for each case will begin on a new line. The format must be as follows: The message 'Case n: msg' where **n** is the case number starting with 1, and **msg** is the phrase *might be ambiguous* or *definitely unambiguous*, with single spaces used as delimiters.

Sample Input

```
4
1001
0
0011
1
2
0011
00110
4
1001
00
101
01
0
```

Sample Output

```
Case 1: might be ambiguous
Case 2: might be ambiguous
Case 3: definitely unambiguous
```

B: Double or Nothing

In this problem you are given two non-negative integers **a** and **b**, each fitting a 32 bit **int**. The idea is to repeatedly divide **b** by 2 until it drops to 0. Each time **b** becomes odd, divide **a** by 2, and each time **b** becomes even, multiply **a** by 2. Perform the specified operation (halving or doubling) on **a** after each division of **b** that does not result in 0. You are to report the final value of **a**. Note that all divisions are integer division.

Input

There may be multiple rounds to process. For each round there is a new line containing the two integers **a** and **b** to use. The last round is followed by a line containing two 0's, as though there were a new case consisting of **a** and **b** both being 0.

Output

The output for each case will begin on a new line. The format must be as follows: The message 'Case n: m' where **n** is the case number starting with 1, **m** is the final value of **a**, with single spaces used as delimiters.

Sample Input

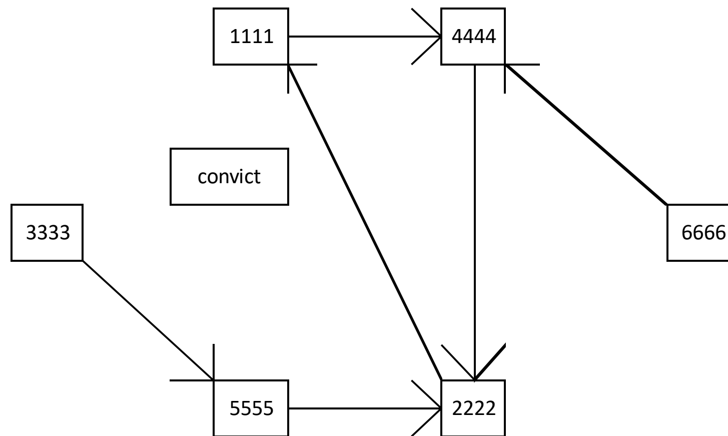
```
64 11
101 11
101 1
0  0
```

Sample Output

```
Case 1: 32
Case 2: 50
Case 3: 101
```

C: Firing Squad 1

In the stereotypical circular firing squad, we have a squad of incompetent guards who form a circle around the convict, but end up shooting each other! For this exercise you are given a squad of **n** guards along with a guard at whom each is (unwittingly) aiming at. Your task is to determine if any of the guards are safe from being shot. Assuming that the weapons are not fired simultaneously, there has to be at least one guard left standing simply because any guards aiming at him are shot first. (The one exception is if each and every guard shoots himself.) However, we might not be able to determine which guard that will be. So the only way to be truly safe is to not aim at oneself and not be the target of any other guard. We want to know if there are any guards guaranteed to be safe, not just lucky to survive. Let's not worry about the convict!



In the figure above (coming from the first sample data), guard 1111 aims at 4444 who aims at 2222 who aims back at 1111 in a cycle so that all three are at risk. Guard 6666 also aims at 4444 and guard 3333 aims at 5555 who in turn also aims at 2222. No guard aims at himself and none aim at the convict. That leaves guards 3333 and 6666 as guaranteed safe.

Input

There may be multiple firing squads to analyze. Each squad is detailed on **n**+1 lines. The first line specifies the size of the squad. This will be a positive integer that fits in an unsigned 16 bit **int**. Each of the other lines contains two integers which specify a guard and his unintended target. This listing may be in arbitrary order. The last case is followed by a line containing only the number 0, as though there are no guards in a new case.

Output

The output for each case will begin on a new line. The format must be as follows: The message 'Case n: msg' where **n** is the case number starting with 1, and **msg** is either 'safe' or 'unsafe' depending on if there is at least one identifiable guard left standing, regardless of the order of shooting, all with single spaces used as delimiters.

Sample Input

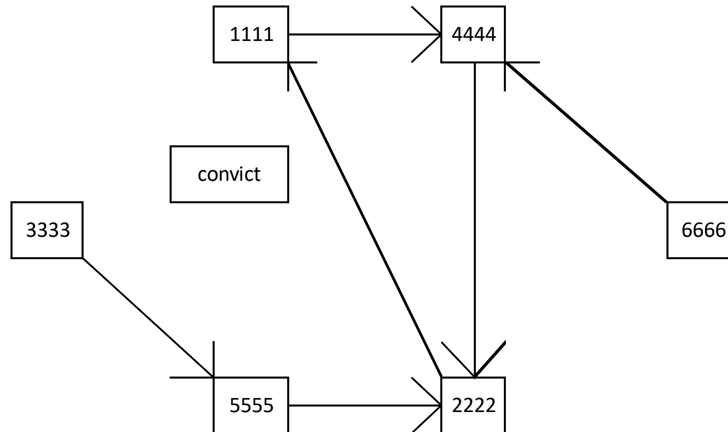
```
6
1111 4444
4444 2222
6666 4444
2222 1111
5555 2222
3333 5555
4
1234 0008
1111 1234
2424 2424
0008 1111
0
```

Sample Output

```
Case 1: safe
Case 2: unsafe
```

D: Firing Squad 2

In the stereotypical circular firing squad, we have a group of incompetent guards forming a circle around the intended convict so that they unwittingly shoot each other! For this exercise you are given a squad of **n** guards, each paired with a guard (possibly oneself) being aimed at. Your task is to determine the maximum number of guards who can possibly escape being shot. This will depend on the order they fire their weapons. Assuming that they do not shoot simultaneously, there has to be at least one guard left standing, unless each and every guard shoots himself. Let's not worry about the convict, since he sees what is happening and ducks in time!



In the figure above (coming from the first sample data), no one aims at guards 3333 and 6666 so they both survive. In a worst case 2222 shoots 1111 before being shot by 4444 or 5555, both of whom are subsequently shot by 6666 and 3333 respectively. But in the best case 2222 is shot before being able to shoot 1111, leaving a maximum of three surviving guards.

Input

There may be multiple firing squads to analyze. Each squad is detailed on **n**+1 lines. The first line specifies the size of the squad which is a positive integer that fits in a 16 bit unsigned **int**.. Each of the other lines contains two integers which specify a guard and his unintended target guard. This listing may be arbitrarily ordered. The last case is followed by a line containing only the number 0, as though there are no guards in a new case.

Output

The output for each case will begin on a new line. The format must be as follows: The message 'Case n: m' where **n** is the case number starting with 1, and **m** is the maximum number of surviving guards, all with single spaces used as delimiters.

Sample Input

```
6
1111 4444
4444 2222
6666 4444
2222 1111
5555 2222
3333 5555
4
1234 0008
1111 1234
2424 2424
0008 1111
0
```

Sample Output

```
Case 1: 3
Case 2: 1
```

E: Incrementing Bases

Incrementing numbers is generally viewed as a very basic operation, so let's make this a bit more interesting. Instead of using decimal numbers, let's allow a choice of bases such as binary, octal, etc.. Using only the numerals 0 through 9 limits the range of possible bases, so let's instead use uppercase letters. So if our chosen base is sixteen (hexadecimal), the digits will be A (zero), B (one), C (two), D (three), E (four), F (five), G (six), H (seven), I (eight), J (nine), K (ten), L (eleven), M (twelve), N (thirteen), O (fourteen) and P (fifteen), so that the decimal number thirty-six would be represented as CE rather than the more familiar 24_{16}.

The idea is to increment a non-negative integer in a chosen base. So given the input 16 (indicating hexadecimal) and CE (effectively 24_{16}), incrementing by 1 gives us CF (effectively 25_{16}).

Input

There may be multiple integers to increment. Each case will be given on two lines. The first line will contain the desired base which is a positive integer up to and including 26. The second line will contain the integer to increment (which is already in the desired base) using "numerals" A (zero) through Z (twenty-five). The length of the integer will be no more than 30 characters. The last case will be followed by two lines consisting of 0 and A respectively, as though they were a new case with base 0 and the (decimal) number 0.

Output

The output for each case will take three lines. The format must be as follows: On the first line is the message 'Case n: b' where **n** is the case number starting with 1 and **b** is the desired base, with single spaces used as delimiters. On the second line is the original number to increment, indented by two spaces. On the third line is the incremented number, with any leading 0's (really A's) removed, also indented by two spaces.

Sample Input

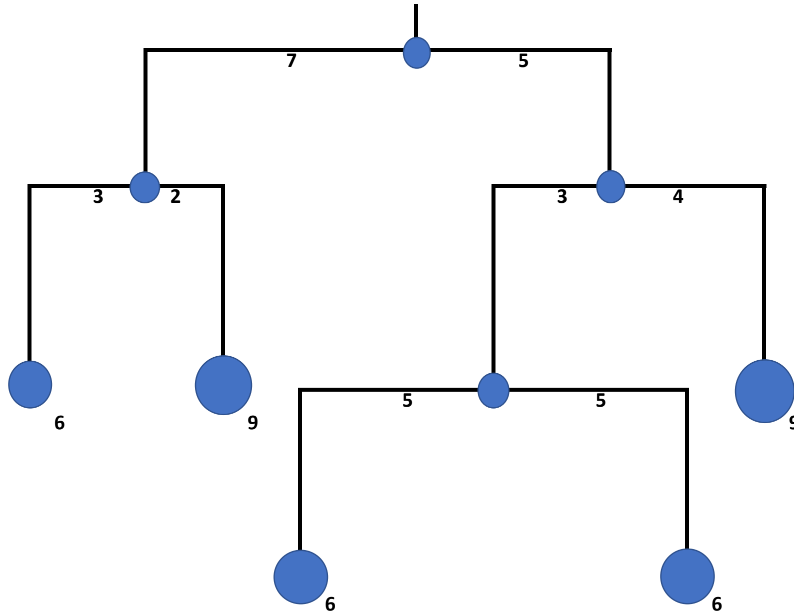
```
16
CE
8
DCH
0
A
```

Sample Output

```
Case 1: 16
  CE
  CF
Case 2: 8
  DCH
  DDA
```

F: Mobile Computing

Mobile computing has been attracting a lot of attention in recent years, thus motivating this little problem. Given a partial description of a mobile presented as a two-dimensional structure with horizontal arms of varying lengths, each arm suspended from some intermediate point, and a collection of weights to be hung from the ends of the lower arms, your task is to determine if there is some distribution of the weights that brings the entire mobile into balance.



In the example above, which comes from the first sample input, there are four arms having total lengths of 12, 5, 7, and 10 units, and points of suspension at 7, 3, 3, and 5 units from the left ends respectively. The weights are 6, 6, 6, 9, and 9 units. The distribution of the weights as shown does bring the entire mobile into balance. This can be seen applying the relationship $W_1L_1 = W_2L_2$ to the weights and lengths on either side of each arm.

Input

There may be multiple mobiles to process. Each mobile is described on two lines with the first line specifying the arm lengths and connections as nested parenthesized nodes, and the second line listing in arbitrary order the weights to be used. Each parenthesized node consists of four elements: the length of the left side, the length of the right side, the (nested) node or the weight suspended from the left side, and the (nested) node or the weight suspended from the right side. The places for weights are designated by periods (.'s). For example, the image above would be represented as (7 5 (3 2 . .) (3 4 (5 5 . .) .)). Spaces are used to delimit numbers and are optional between parentheses and periods. All values are positive integers.

The last mobile description will be followed by a line consisting of a parenthesized string of two zeros and

two periods and perhaps spaces as in (0 0 . .), basically using the same syntax as would be consistent with a mobile having one arm of zero lengths and two weights.

Output

The output for each case will begin on a new line. The format must be as follows: The first line should contain the message 'Case n: Can be balanced' or 'Case n: Can not be balanced' where **n** is the case number starting with 1 and with single spaces used as delimiters.

Sample Input

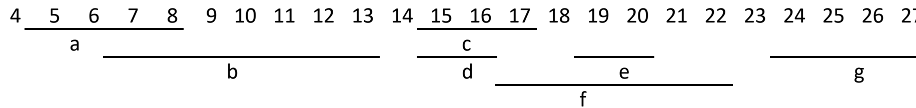
```
( 7 5 ( 3 2 . . ) ( 3 4 ( 5 5 . . ) . ) )
6 9 6 9 6
( 4 6 . . )
5 5
( 4 6 . . )
2 3
( 0 0 . .)
```

Sample Output

```
Case 1: Can be balanced
Case 2: Can not be balanced
Case 3: Can be balanced
```

G: Paving Jobs

The road used to be all gravel. Over the years some sections were paved, and sometimes repaved as portions deteriorated. In time there was less and less gravel and more and more pavement. You are to determine how many remaining sections of the road are still gravel. Assume that no pavement ever reverts to gravel.



The example above comes from the sample input, with the lettering indicating the sequence of construction. The road being represented is 23 miles in length, extending from mile marker 4 to mile marker 27. There were 7 sections that were paved or repaved. Only two sections of gravel (from mile markers 13 to 14 and from 22 to 23) remain.

Input

There may be multiple roads to evaluate. Each road is described on multiple lines. On the first line are three non-negative integers, the first two representing the beginning and ending mile markers, and the third being the number of pavings. Each paving job is detailed on a separate line with its starting and ending mile markers. The last road to be checked is followed by a single line containing three 0's, as though a new road's specifications were all 0. You may ssume that all mile marker pairs have the starting mile marker less than the ending mile marker.

Output

The output for each case will begin on a new line. The format must be as follows: The message 'Case n: m' where **n** is the case number starting with 1, **m** is the number of remaining stretches of gravel, with single spaces used as delimiters.

Sample Input

```
4 27 7
4 8
6 13
14 17
14 16
18 20
16 22
23 27
0 100 1
10 90
0 0 0
```

Sample Output

```
Case 1: 2
Case 2: 2
```

H: Quad Pictures

There are a variety of data compression techniques. One that can be used for compressing images involves recursively dividing up a rectangle into four 'panes' as in a window. The recursion continues on any pane that contains pixels of brightnesses that differ by more than some maximum amount. In this way, each of the resulting panes very closely approximates the brightness of all the pixels it contains. For example, a picture consisting of open sky in the upper right, open sea in the lower left, sea gulls and sky in the upper left, and some waves in the lower right might be represented (in very low resolution) by the figure below. You will be given an image in this compressed quad format. Your task is to use the brightnesses to compute the amount of ink needed to print the image.

In this example we have an 8 x 8 pixel image (based on the finest granularity of any pane or quad). The numbers represent the average brightness of the pixels in each quad. The numbers are integers ranging from 0 (white) to 255 (black), the reverse of what is typical (which is being done for convenience in computation). The amount of ink in each quad is the average pixel brightness multiplied by the area (number of pixels). You can verify that in the example, this totals 106 units of ink.

Input

There may be multiple images to process. Each image is represented on a single line of no more than 100 characters. The four quads are represented as a sequence of integers bounded by parentheses. Any recursively divided quads are shown as being nested. The ordering of the quads in the input is as upper-left, upper-right, lower-left, and lower-right. For example, the image above would be shown as ((2 5 (3 1 2 0) 4) 1 0 (1 2 3 4)). Spaces are used to delimit numbers. Spaces are optional around parentheses. The shape is always square and the resolution (pixels on a side) is determined by the maximum nesting of the quads, so that the smallest panes are 1 x 1 pixel. The last case will be followed by a line consisting of (0 0 0 0) instead of the usual quad image representation.

Output

The output for each case will begin on a new line. The format must be as follows: The message 'Case n: m' where **n** is the case number starting with 1, **m** is the quantity of ink needed for the picture, with single spaces used as delimiters.

Sample Input

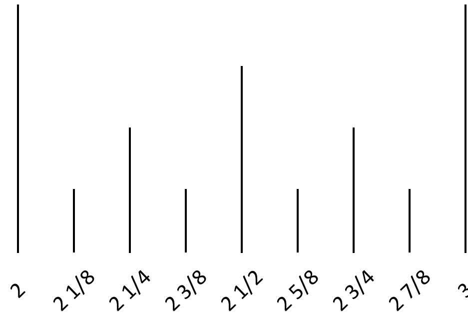
```
( ( 2 5 ( 3 1 2 0 ) 4 ) 1 0  ( 1 2 3 4 ) )
( 1 2 3 4 )
( 0 0 0 0 )
```

Sample Output

```
Case 1: 106
Case 2: 10
```

I: Ruler Divisions

Rulers, particularly ones marked in inches, are commonly divided into fractions with denominators being powers of 2. You will be given specifications for a section of a ruler which is to be divided up thusly, with the fractions listed out.



In the example above, which comes from the first sample input, the range (in inches) is 2 to 3 with divisions marked every 1/8 inch. You won't have to print out the varying length lines, but you will need to print all the fractions in the specified range, in lowest terms as shown.

Input

There may be multiple cases to process. Each case is given on one line as three integers **b**, **e**, and **d** where **b** and **e** are the beginning and ending marks of the interval, and $1/d$ is the size of each division. **d** will be a power of 2 ranging from 1 to at most 256. The last case will be followed by a line consisting of three 0's instead of the specifications for a new case. The numbers may be separated by more than one space.

Output

The output for each case will begin on a new line. The format must be as follows: The first line should contain the message 'Case n: b e d' where **n** is the case number starting with 1, **b** and **e** are the beginning and ending marks, and $1/d$ is the size of each division (just as in the input), with single spaces used as delimiters. The following lines contain each of the fractions in the specified interval, preceeded by two spaces.

Sample Input

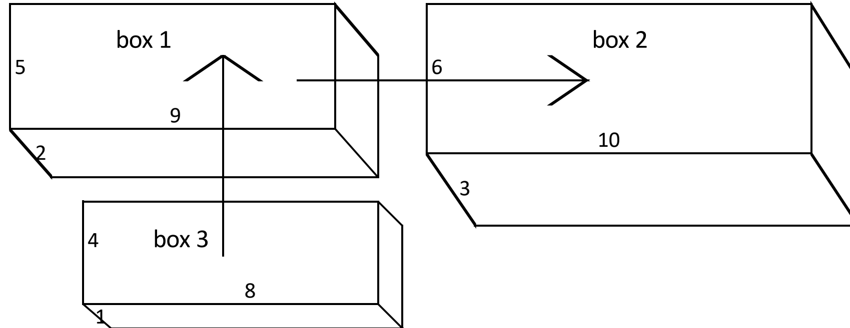
```
2␣␣3␣␣8
2␣␣4␣␣1
0␣␣0␣␣0
```

Sample Output

```
Case␣1:␣2␣3␣8
␣␣2
␣␣2␣1/8
␣␣2␣1/4
␣␣2␣3/8
␣␣2␣1/2
␣␣2␣5/8
␣␣2␣3/4
␣␣2␣7/8
␣␣3
Case␣2:␣2␣4␣1
␣␣2
␣␣3
␣␣4
```

J: Three Boxes

You are given three rectangular boxes of various dimensions. In order to save space, we want to know if the three boxes nest with the smallest inside the middle-sized which is inside the largest. You are given the outside dimensions in arbitrary order for each of the three boxes, which are themselves listed in arbitrary order. There is a small thickness to the box material - we can simply say 0.1 unit - so that the interior dimensions are somewhat smaller. Fortunately, we can make this a bit less complicated by stating that the solution must be rectilinear (meaning nothing is tipped or turned at anything other than right angles.)



In the example above (case 3 from the Sample Input), you can see that by turning the boxes appropriately, you will be able to fit box 3 into box 1 which then fits into box 2.

Input

There may be multiple cases to process. Details of each set of three boxes will be contained on a single line. This will consist of 9 non-negative integers representing the three dimensions for each box in turn. The last case will be followed by a line consisting of nine 0's instead of the usual dimensions for a new set of three boxes. There may be arbitrary spaces in the input.

Output

The output for each set of boxes should begin on a new line. The format must be as follows: The message 'Case n: msg' where **n** is the case number starting with 1, **msg** is the message 'fit' or 'no fit', with single spaces used as delimiters.

Sample Input

```
2 2 2   1 1 1   3 3 3
10 10 10   2 2 2   2 2 2
2 5 9   6 10 3   4 1 8
3 3 3   3 3 3   3 3 3
0 0 0   0 0 0   0 0 0
```

Sample Output

```
Case 1: fit
Case 2: no fit
Case 3: fit
Case 4: no fit
```