Problem A

Quite Good Numbers

A "perfect" number is an integer that is equal to the sum of its divisors (where 1 is considered a divisor). For example, 6 is perfect because its divisors are 1, 2, and 3, and 1 + 2 + 3 is 6. Similarly, 28 is perfect because it equals 1 + 2 + 4 + 7 + 14.

A "quite good" number is an integer whose "badness" – the absolute value of the difference between the sum of its divisors and the number itself – is not greater than a specified value. For example, if the allowable badness is set at 2, there are 11 "quite good" numbers less than 100: 2, 3, 4, 6, 8, 10, 16, 20, 28, 32, and 64. But if the allowable badness is set at 0 (corresponding to the "perfect" numbers) there are only 2: 6 and 28.

Your task is to write a program to count how many quite good numbers (of a specified maximum badness) fall in a specified range.

Input

Input will consist of specifications for a series of tests. Information for each test is a single line containing 3 integers with a single space between items:

- **`start`** (**`2 <= start < 1000000`**) specifies the first number to test
- **`stop`** (**`start <= stop < 1000000`**) specifies the last number to test
- **`badness`** (**`0 <= badness < 1000`**) specifies the maximum allowable badness

A line containing 3 zeros terminates the input.

Output

Output should consist of one line for each test comprising the test number (formatted as shown) followed by a single space and the number of values in the test range with badness not greater than the allowable value.

Sample Input

```
2 100 2
2 100 0
1000 9999 3
0 0 0
```

Sample Output

```
Test 1: 11
Test 2: 2
Test 3: 6
```

Problem B

Primordial Values

What's the next number in the following sequence?

```
1, 11, 21, 1211, 111221, ...
```

Most people assume that the sequence has something to do with numeric values and struggle to find a pattern . In fact, the rule for generating the next value in the series is that each value is a *description* of the previous value. For example, 21 can be described as "one 2, one 1", leading to the next value of 1211. By this rule, the answer to the problem above is 312211 ("three 1, two 2, one 1").

The process can also be reversed to determine *previous* values in the sequence. For example, the value that comes before 2221 (two 2, two 1) would be 2211, and the value before that would be 221. However, not every value has a previous value. For example, 221 has no previous value because it isn't a valid description, and 2212 has no previous value because the value it describes (222) would have a next value of 32, not 2212.

The "primordial value" of some value is defined as the value that would generate a sequence containing the given value but that itself has no previous value. For example, if the starting value is 2221 the primordial value is 221, and if the starting value is 312211 the primordial value is 1. As a special case, the value 22 (for which the previous value would also be 22) is considered primordial.

Your task is to write a program that determines primordial values for given starting values.

Input

Input will consist of specifications for a series of tests. Information for each test is a single line containing a string of digits of length `1 <= n < 100` that specifies the *last* value in a sequence defined as above. A line containing the value `0` terminates the input.

Output

Output should consist of one line for each test comprising the test number (formatted as shown) followed by a single space and the primordial value of the input value.

Sample Input

```
2221
312211
22
0
```

Sample Output

```
Test 1: 221
Test 2: 1
Test 3: 22
```

Problem C

Fun With Fractions

A rational number can be represented as the ratio of two integers, referred to as the numerator (**n**) and the denominator (**d**) and written **n/d**. A rational number's representation is not unique. For example the rational numbers **1/2** and **2/4** are equivalent. A rational number representation is described as "in lowest terms" if the numerator and denominator have no common factors. Thus **1/2** is in lowest terms but **2/4** is not. A rational number can be reduced to lowest terms by dividing by the greatest common divisor of **n** and **d**.

Addition of rational numbers is defined as follows. Note that the right hand side of this equality will not necessarily be in lowest terms.

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

A rational number for which the numerator is greater than or equal to the denominator can be displayed in mixed format, which includes a whole number part and a fractional part. For example, $5\frac{1}{3}$ is a mixed format representation of the rational number $\frac{16}{3}$.

Your task is to write a program that reads a sequence of rational numbers and displays their sum.

Input

Input will consist of specifications for a series of tests. Information for each test begins with a line containing a single integer **1 <= n < 1000** indicating how many values follow. A count of zero terminates the input.

The **n** following lines each contain a single string with no embedded whitespace . Each string represents a rational number, which could be in any of the following forms and will not necessarily be in lowest terms (**w**, **n**, and **d** are integers: **0 <= w,n < 1000**, **1 <= d < 1000**).

- **w,n/d**: a mixed number equivalent to the rational number **(w*d + n) / d**.
- **n/d**: a rational number with a zero whole number part
- **w**: a whole number with a zero fractional part

Output

Output should consist of one line for each test comprising the test number (formatted as shown) followed by a single space and the sum of the input number sequence. The sum should be displayed in lowest terms using mixed number format. If either the whole number part or the fractional part is zero, that part should be omitted. As a special case, if both parts are zero, the value should be displayed as a single **0**.

Sample Input

```
2
1/2
1/3
3
1/3
2/6
3/9
3
1
2/3
4,5/6
0
```

Sample Output

```
Test 1: 5/6
Test 2: 1
Test 3: 6,1/2
```

Problem D

The Queen's English

In the official British written word format for numbers, tens and units are separated by a hyphen, and hundreds are separated from tens or units by the word "**and**". For example the number **123** would be written "**one hundred and twenty-three**". Large numbers are written in "triads" (groups of three digits) followed by the appropriate suffixes. For example **123456** is "**one hundred and twenty-three thousand four hundred and fifty-six**". As a special case, if the final triad of a large number has a tens or units component but no hundreds component, it needs an "**and**". Thus 1001001 is "**one million one thousand and one**".

Your task is to write a program that produces correct British written format for numbers of up to 9 digits (less than one "short" billion). Note that the correct spellings are "**fifteen**" and "**fifty**" ('**f**' instead of '**v**'), and "**forty**" (no '**u**').

Input

Input will consist of specifications for a series of tests. Information for each test is a single line containing an integer **1 <= n < 1000000000** that specifies the value to process. A line containing the value **0** terminates the input.

Output

Output should consist of one line for each test comprising the test number (formatted as shown) followed by a single space and the correct British word form of the input value, with a single space between words.

Sample Input

```
123
1001001
900090009
0
```

Sample Output

```
Test 1: one hundred and twenty-three
Test 2: one million one thousand and one
Test 3: nine hundred million ninety thousand and nine
```

Problem E

Powerbase Format

Rather than the familiar right-to-left place-weighted system for printed numbers, powerbase representation writes digits from left to right with the rule that a digit **d** in position **p** has the value of **d** raised to the power **p**. For example, the integer that we would normally write as **100** would be represented in powerbase form as **983** because $9^1 + 8^2 + 3^3$ is one hundred.

Powerbase format has the advantage that there is no need to know (or assume) what radix (base) a number is written in. For example **100** in a place-weighted scheme could represent one hundred (if the radix is ten) or four (if the radix is two). But **983** in powerbase format can only ever represent one hundred.

The lack of a radix also frees up the matter of which digits are available. Powerbase numbers are frequently written without 0 or 1 digits, but could in principle be written with any particular digit set. However, restricting the available digits may mean that particular values have no powerbase representation, or at least none shorter than a small number of digits. For example if numbers are written in powerbase format using digits 2 to 8, all values from 2 to 10,000 can be represented using no more than 7 digits (with only 31 of them needing the full 7-digit form). However, if only digits 2 to 7 are used there is no powerbase representation (of any length) for the value 25.

Your task is to write a program to find integer values that do not have a powerbase representation shorter than a specified number of digits when a particular digit set is used.

Input

Input will consist of specifications for a series of tests. Information for each test begins with a line containing two integers, separated by a single space character, that specify the first and last value to check. Each value is **1 <= n <= 100000**, and the last value is not smaller than the first. A line containing two zeros terminates the input.

The second line for each test contains an integer **2 <= n <= 15** that specifies the maximum length of the powerbase representation that should be considered. The third and final line for each test contains a string that specifies which digits can be used in the powerbase representation. The contents of the digit string can include the numeric characters **0** to **9** and the uppercase letters **A** to **Z**, which represent digit values 10 to 35. The characters in the digit string are arranged in order of strictly increasing digit value.

For example the input specification

```
100 2000
5
1248GW
```

describes a test of the integers from 100 to 2000 (both inclusive), looking for values that can be represented in powerbase form using no more than 5 digits, where allowable digit values are 1, 2, 4, 8, 16, and 32.

Output

Output should consist of one line for each test comprising the test number (formatted as shown) followed by a single space and the number of values in the test range that cannot be represented under the specified conditions for length and digit values.

Sample Input

```
2 10000
7
23456789
2 10000
6
23456789
25 25
15
234567
100 2000
5
1248GW
0 0
```

Sample Output

```
Test 1: 0
Test 2: 31
Test 3: 1
Test 4: 1363
```

Problem F

Jumble Match

For the purpose of this problem, a word matches a pattern if a substring of the word matches the pattern. A pattern can contain letters (which should be matched explicitly) and underscores (which match any single letter). Finally, a pattern is considered matched if any jumble (permutation) of the pattern characters matches.

For example, the pattern **`cat`** matches words such as **`cat`**, **`scat`**, and **`cater`** because **`cat`** is a substring of the word. However, the pattern also matches words that contain permutations of **`cat`** as a substring, such as **`tacit`**, **`latch`**, and **`fact`**. And the pattern **`cat_`** would match any word that contains a 4-character substring that is some permutation of the letters **`c`**, **`a`**, **`t`**, and any other letter, which would therefor match words such as **`track`**, **`cant`**, or **`crate`**.

Your task is to write a program that searches for a jumble match in a set of input words.

Input

Input will consist of specifications for a series of tests. Information for each test begins with a line containing a single string of length **`1 <= n < 100`** that specifies the pattern to search for. A pattern string consisting of a single dot terminates the input.

The lines following the pattern begin with an integer **`1 <= n <= 10`** that specifies the number of words on the line, followed by the words themselves, with a single space between items. Words are strings of alphabetic characters of length **`1 < n < 20`**. A line with a word count of 0 ends the input for each test.

Output

Output should consist of one line for each test comprising the test number (formatted as shown) followed by a single space and the number of words in the input set that match the pattern.

Sample Input

```
cat
10 act canto chest colt crest eject hutch scant tact track
10 bitch cat civet cotta cut evict notch stack tic tuck
2 cant chert
0
cat_
10 act canto chest colt crest eject hutch scant tact track
10 bitch cat civet cotta cut evict notch stack tic tuck
2 cant chert
0
```

```
c_t_
10 act canto chest colt crest eject hutch scant tact track
10 bitch cat civet cotta cut evict notch stack tic tuck
2 cant chert
0
.
```

Sample Output

```
Test 1: 4
Test 2: 6
Test 3: 14
```

Problem G

Road Trip

You are planning a road trip to visit your friends, each of whom live in different towns. Of course, you don't want to pay any more for fuel on the trip than you need to. However, the price of fuel in each of the towns is different, so you will need to carefully plan the trip to minimise the cost. For example, it might make sense to take on extra fuel at a town where the price is low so that you don't need to buy so much at a town where it is more expensive. Indeed, it may even make sense to sell excess fuel at some towns to recoup some of the costs. Of course, your car can only hold a certain amount of fuel, and you have to make sure you take on enough fuel at each town to reach the next (assume that it's OK to arrive with an empty tank).

Your task is to write a program to help you plan your trip.

Input

Input will consist of specifications for a series of journeys. Information for each journey will begin with a line containing an integer `0 < c < 100` that specifies the capacity of the car's fuel tank in litres and an integer `0 < t < 20` that specifies the number of towns to visit for that journey. A line containing two zeros indicates the end of input.

The following `t` lines contain information about successive stages on the journey: the price (in fixed point dollars-and-cents format, `0.01 <= p < 9.99`) of one litre of fuel (either to buy or to sell) in the town at the beginning of the stage, and the number of litres of fuel (an integer, 1 `<= n < 100`) needed to reach the next town.

Output

Output should consist of one line for each journey comprising the journey number (formatted as shown) followed by a single space and the minimum cost of completing that journey, formatted as a fixed-point number with 2 decimal places.

Sample Input

```
10 3
2.00 7
1.50 8
1.00 3
50 6
1.50 20
4.20 5
1.15 35
1.41 27
1.92 30
2.21 15
0 0
```

Sample Output

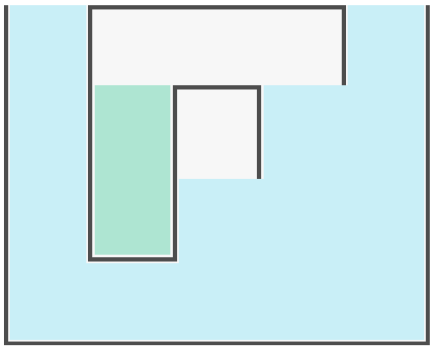
```
Journey 1: 29.00
Journey 2: 117.64
```

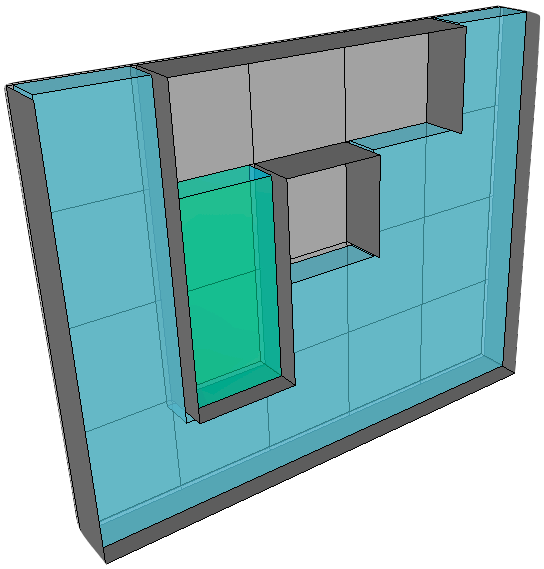
Problem H

Baffled!

Consider a thin rectangular tank with baffles that block the flow of water. The tank can be filled from one or more holes in its top, but the baffles prevent the water from reaching various parts, so that the tank may end up containing trapped pockets of air.

For example, the diagram below suggests a possible tank in profile and in perspective (with the front removed so you can see what's inside). The tank is "full", but the configuration of the baffles leaves two pockets of air. Note that because of the syphon effect, water will flow into the green "well" along the intervening baffle, even though the air pocket above will remain empty.





The tank dimensions and the positions of the baffles fall on a regular grid. For example, the tank above is 5 units wide, 4 units high, and 1 unit in thickness, for a total volume of 20 cubic units. However, the trapped pockets of air occupy 4 cubic units, so the amount of water that the tank can hold is only 16 cubic units. (For the purposes of this problem, you can assume that the baffles do not occupy any volume, and that air is incompressible.)

Your task is to write a program that can compute the amount of water that can be poured into a given tank before it becomes "full".

Input

Input will consist of specifications for a series of tanks. Information for each tank begins with a line containing two integers, separated by a single space, that specify the tank's width (`1 <= w < 100`) and height (`1 <= h <= 1000`). (All tanks have an implied thickness of 1.) A line containing `0 0` terminates the input.

The following lines contain strings that specify the location of baffles. For a tank of width `w` and height `h`, there will be exactly `h+1` lines, each containing exactly `2*w+1` characters. Each line specifies the baffles around the cells in one "row" of the tank. The first character represents the left-hand wall of the tank and is always `'|'` (except for first line, which

represents the "lid" row of cells above the real tank). The remaining characters in each line, taken in pairs, represent the cells on that line. If a cell has a baffle below it, the first character in the pair is '_'; if it is open below, the character is '.'. Similarly, if a cell has a baffle to its right, the second character in each pair is '|'; otherwise it is '.'.

The choice of characters is such that the lines create a simple ASCII picture of the tank profile. For example, the tank above would be specified as follows:

```
5 4
...._._._...
|.|..._..|.|
|.|.|.|.|...|
|.|_|.....|
|_._._._._|
```

Output

Output should consist of one line for each tank comprising the tank number (formatted as shown) following by a single space and the volume of water the tank can hold.

Sample Input

```
3 2
.......
|.....|
|_._._|
5 4
...._._._...
|.|..._..|.|
|.|.|.|.|...|
|.|_|.....|
|_._._._._|
6 3
._._._...._._.
|.._..|.|...|
|.|.|_..|.|.|
|_._|_|_._|_|
0 0
```

Sample Output

```
Tank 1: 6
Tank 2: 16
Tank 3: 8
```