Problem A: Parenthesis

Source: `parenthesis.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

To a computer, there is no difference between the expression `(((x)+(y))(t))` and `(x+y)t`; but, to a human, the latter is easier to read. When writing automatically generated expressions that a human may have to read, it is useful to minimize the number of parentheses in an expression. We assume expressions consist of only two operations: addition (+) and multiplication (juxtaposition), and these operations act on single lower-case letter variables only. Specifically, here is the grammar for an expression `E`:

```
E : P | P '+' E
P : F | F  P
F : V | '(' E ')'
V : 'a' | 'b' | .. | 'z'
```

The addition (`+`, as in `x+y`) and multiplication (juxtaposition, as in `xy`) operators are associative: `x+(y+z)=(x+y)+z=x+y+z` and `x(yz)=(xy)z=xyz`. Commutativity and distributivity of these operations should not be assumed. Parentheses have the highest precedence, followed by multiplication and then addition.

Input

The input consists of a number of cases. Each case is given by one line that satisfies the grammar above. Each expression is at most 1000 characters long.

Output

For each case, print on one line the same expression with all unnecessary parentheses removed.

Sample input

```
x
(x+(y+z))
(x+(yz))
(x+y(x+t))
x+y+xt
```

Sample Output

```
x
x+y+z
x+yz
x+y(x+t)
x+y+xt
```

Problem B: Ropes

Source: `ropes.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

When climbing a section or "pitch", the lead climber ascends first, taking a rope with them that they anchor to the rock for protection to ascend. Once at the top of a pitch, the lead climber has the second climber attach to the rope, so they can ascend with the safety of the rope. Once the second climber reaches the top of the pitch, the third attaches, and so on until all the climbers have ascended.

For example, for a 10 meter pitch and 50 meter rope, at most 6 climbers could ascend, with the last climber attaching to the end of the rope. To ascend safely, there must be at least 2 climbers and the rope must be at least as long as the pitch.

This process is repeated on each pitch of the climb, until the top is reached. Then to descend, the climbing rope is hung at its midpoint from an anchor (each half must reach the ground). The climbers then each rappel from this rope. The rope is retrieved from the anchor by pulling one side of the rope, slipping it though the anchor and allowing it to fall to the ground.

To descend safely, the rope must be at least twice as long as the sum of the lengths of the pitches.

For example, a 60 meter rope is required to rappel from a 30 meter climb, no matter how many climbers are involved.

Climbing ropes come in 50, 60 and 70 meter lengths. It is best to take the shortest rope needed for a given climb because this saves weight. You are to determine the maximum number of climbers that can use each type of rope on a given climb.

Input

The input consists of a number of cases. Each case specifies a climb on a line, as a sequence of pitch lengths as in:

$$N\ P_1\ P_2\ \ldots\ P_N$$

Here N is the positive number of pitches, with $1 \le N \le 100$, and P_k is the positive integer length (in meters) of each pitch, with $1 \le P_k \le 100$. The last line (indicating the end of input) is a single 0.

Output

Three numbers for each climb separated by a space, indicating the maximum number of climbers that could use the 50, 60, or 70 meter rope lengths, respectively. State 0 if the given rope length is not suitable for that climb.

Sample input

```
1 25
2 10 20
0
```

Sample Output

```
3 3 3
0 4 4
```

Problem C: Chain Code

Source: `chaincode.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

In a black and white (bi-level) image, collections of connected black pixels can be defined as foreground or objects, while white can be thought of as background. Each set of connected black pixels can be completely described by listing the positions of the pixels on its boundary in counterclockwise order, starting at some arbitrary point on the boundary. This list of pixels can, in turn, be represented simply as the direction to the next one in the list. This list of directions is called the chain code of the object, and describes the shape of the object precisely while being position independent.

There are 8 possible directions from one pixel to an adjacent pixel, and while assigning these numbers is arbitrary, **figure 1** shows the standard convention. The direction 0 means "to the right of", 2 "means immediately above", and 1 is at 45 degrees, bisecting 0 and 2, and so on.

Two black pixels are considered to be adjacent if the square of the distance between them is less than or equal to 2. This is based on a standard graphics coordinate system having a pixel at each integer coordinate. Two pixels are connected if a contiguous path of adjacent pixels can be traced between them. A connected region is a set of black pixels in which all members are connected to each other. A boundary pixel of a connected region (from now on just a region) is a pixel within the region that has at least one neighbor (in the four compass directions) that is not black. For this problem, you may assume that there are no “holes” in the region, so that there is only one boundary of the region.

The chain code of a region can start at any pixel on the boundary. It proceeds by finding the next adjacent pixel on the boundary in a counter-clockwise direction, saving the direction (0-7) in an output buffer, and then continuing the process from the new pixel. When we arrive at the starting pixel again, the chain code is complete. The output buffer contains a set of direction values which comprise the chain code itself, and from which the original set of pixels can be recreated starting at any pixel position in an image.

As an example, a chain code for the region in **figure 2** is `446567001232`. The chain code describes the shape of the region unambiguously, although its position is completely unknown. Shape related measures such as perimeter and area (number of pixels in the region) can be determined directly from the chain code description alone. You are to write a program that calculates the area of a connected region given only the chain code.

Input

The input will be a collection of chain code strings, one per line. Each chain code contains at most 1000000 characters. You may assume that each chain code describes a valid region, and does not describe a boundary that intersects itself.

Output

For each chain code in the input, the output will be the area of the region (i.e. the number of pixels belonging to it), each printed on its own line.

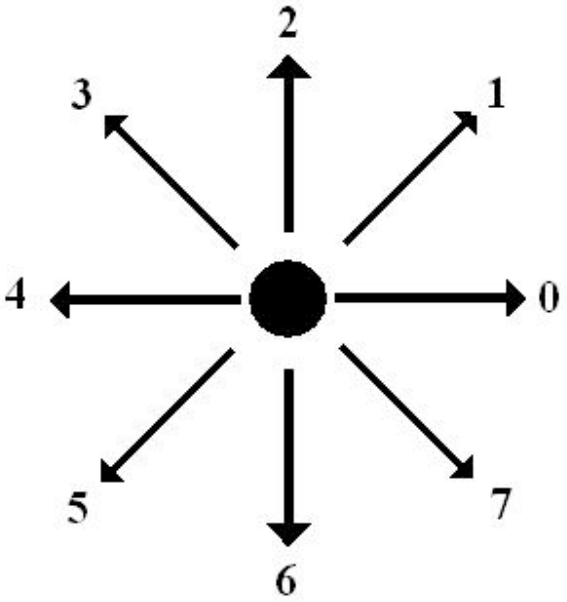

Figure 1: Directions.

Sample input

```
446567001232
6024
```

Sample Output

```
19
4
```

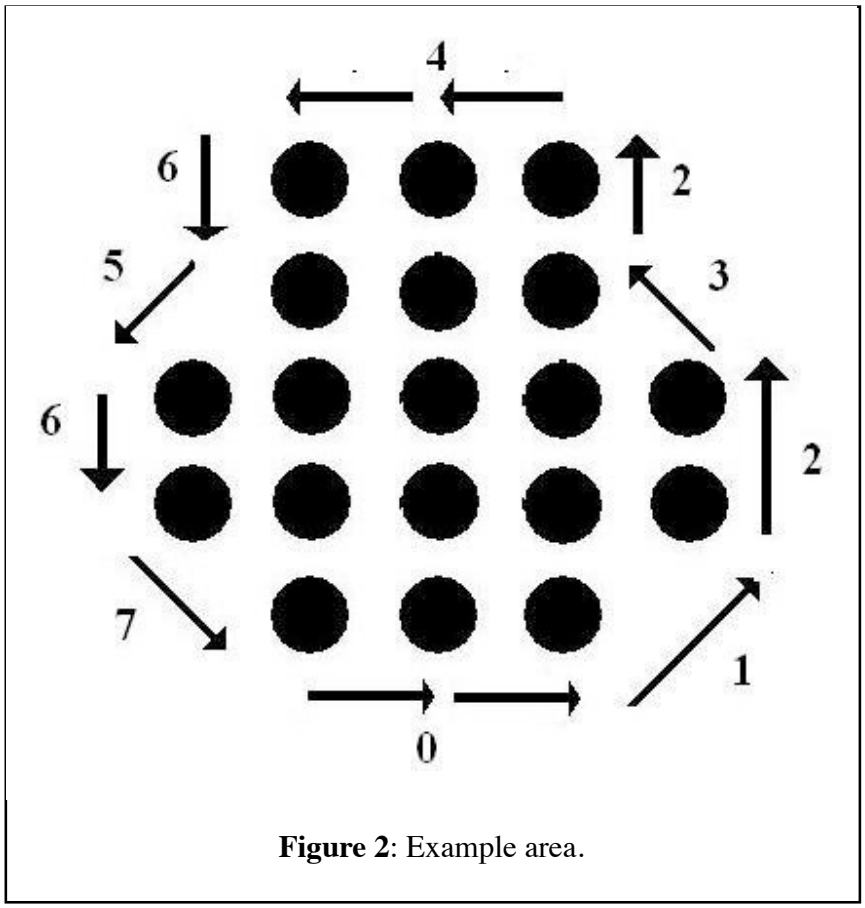

Figure 2: Example area.

Problem D: Task

Source: `task.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

In most recipes, certain tasks have to be done before others. For each task, if we are given a list of other tasks that it depends on, then it is relatively straightforward to come up with a schedule of tasks that satisfies the dependencies and produces a stunning dish. Many of us know that this can be solved by some algorithm called toplogical sort.

But life is not so easy sometimes. For example, here is a recipe for making pizza dough:

1. Mix the yeast with warm water, wait for 5 to 10 minutes.
2. Mix the the remaining ingredients 7 to 9 minutes.
3. Mix the yeast and the remaining ingredients together for 10 to 15 minutes.
4. Wait 90 to 120 minutes for the dough to rise.
5. Punch the dough and let it rest for 10 to 15 minutes.
6. Roll the dough.

In this case, tasks 1 and 2 may be scheduled after the first minute (we always spend the first minute to read the recipe and come up with a plan). The earliest task 3 may be started is at 8 minutes, and task 4 may start at 18 minutes after the start, and so on. This recipe is relatively simple, but if some tasks have many dependent tasks then scheduling can become unmanageable. Sometimes, the recipe may in fact be impossible to execute. For example, consider the following abstract recipe:

1. task 1
2. after task 1 but within 2 minutes of it, do task 2
3. at least 3 minutes after task 2 but within 2 minutes of task 1, do task 3

In this problem, you are given a number of tasks. Some tasks are related to another based on their starting times. You are asked to assign a starting time to each task to satisfy all constraints if possible, or report that no valid schedule is possible.

Input

The input consists of a number of cases. The first line of each case gives the number of tasks **n**, (**1 ≤ n ≤ 100**). This is followed by a line containing a non-negative integer **m** giving the number of constraints. Each of the next **m** lines specify a constraint. The two possible forms of constraints are:

```
task i starts at least A minutes later than task j
task i starts within A minutes of the starting time of task j
```

where **i** and **j** are the task numbers of two different tasks (**1 ≤ i, j ≤ n**), and **A** is a non-negative integer (**A ≤ 150**). The first form states that task **i** must start at least **A** minutes later than the start time of task **j**. The second form states that task **i** must start no earlier than task **j**, and within **A** minutes of the starting time of task **j**. There may be multiple constraints involving the same pair of tasks. Note that **at least** and **within** include the end points (i.e. if task 1 starts at 1 minute and task 2 starts at 4 minutes, then task 2 starts at least 3 minutes later than task 1, and within 3 minutes of the starting time of task 1).

The input is terminated by **n = 0**.

Output

For each case, output a single line containing the starting times of task 1 through task n separated by a single space. Each starting time should specify the minute at which the task starts. The starting time of each task should be positive and less than 1000000. There may be many possible schedules, and any valid schedule will be accepted. If no valid schedule is possible, print **Impossible.** on a line instead.

Sample input

```
6
10
task 3 starts at least 5 minutes later than task 1
task 3 starts within 10 minutes of the starting time of task 1
task 3 starts at least 7 minutes later than task 2
task 3 starts within 9 minutes of the starting time of task 2
task 4 starts at least 10 minutes later than task 3
task 4 starts within 15 minutes of the starting time of task 3
task 5 starts at least 90 minutes later than task 4
task 5 starts within 120 minutes of the starting time of task 4
task 6 starts at least 10 minutes later than task 5
task 6 starts within 15 minutes of the starting time of task 5
3
4
task 2 starts at least 0 minutes later than task 1
task 2 starts within 2 minutes of the starting time of task 1
task 3 starts at least 3 minutes later than task 2
task 3 starts within 2 minutes of the starting time of task 1
0
```

Sample Output

```
3 1 8 18 108 118
Impossible.
```

Problem E: Page Count

Source: `pagecount.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

When you execute a word processor's **`print`** command, you are normally prompted to specify the pages you want printed. You might, for example, enter:

```
10-15,25-28,8-4,13-20,9,8-8
```

The expression you enter is a list of print ranges, separated by commas.

Each print range is either a single positive integer, or two positive integers separated by a hyphen. In the latter case we call the first integer **`low`** and the second one **`high`**. A print range for which **`low > high`** is simply ignored. A print range that specifies page numbers exceeding the number of pages is processed so that only the pages available in the document are printed. Pages are numbered starting from 1.

Some of the print ranges may overlap. Pages which are common to two or more print ranges will be printed only once. (In the example given, pages 13, 14 and 15 are common to two print ranges.)

Input

The input will contain data for a number of problem instances. For each problem instance there will be two lines of input. The first line will contain a single positive integer: the number of pages in the document. The second line will contain a list of print ranges, as defined by the rules stated above. End of input will be indicated by 0 for the number of pages. The number of pages in any book is at most 1000. The list of print ranges will be not be longer than 1000 characters.

Output

For each problem instance, the output will be a single number, displayed at the beginning of a new line. It will be the number of pages printed by the **`print`** command.

Sample input

```
30
10-15,25-28,8-4,13-20,9,8-8
19
10-15,25-28,8-4,13-20,9,8-8
0
```

Sample Output

```
17
12
```

Problem F: Soccer

Source: `soccer.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

In soccer, there are many different rewards (and punishments) depending on how you rank in the league at the end of a season. For example, in the British Premier League, the top 4 teams are eligible to play in the Champions League, the next team is eligible to play in the Europa League, and the bottom three teams are relegated to the lower division. It is now near the end of the soccer season, and there are still a number of games to be played. For any given team, we wish to know what is the highest and lowest rank it can have at the end of the season.

For each game played, a team wins if it scores more goals than its opponent. A team loses a game if it scores fewer goals. When both teams score the same number of goals, we call it a draw. A team earns 3 points for each win, 1 point for each draw and 0 point for each loss. Teams are ranked according to the number of points earned (more points result in a higher ranking). Teams that are tied are given the same rank. For example, if two teams are tied and have the next highest point total after the 3rd place team, then they are both ranked 4th (and the next team is ranked 6th). In real life, factors such as goal differences and goals scored are used to break ties, but we will not consider these for this problem.

You are given a list of soccer teams and a list of matches in a season. You may assume that every team will play the same number of games at the end. Some of the matches have been played and the results are known.

Input

The input consists of a number of cases. The first line in each case specifies two integers **n** and **m** (**2 ≤ n ≤ 20**, **1 ≤ m ≤ 1000**) indicating the number of teams in the league and the number of matches in the season. The next **n** lines contain the name of each team in its own line. The team names contain only alphabetic characters and have lengths at most 30 characters. This is followed by **m** lines each of the form

```
team1 vs team2: x y
```

with **team1** and **team2** being the names of two different teams, and **x** and **y** are non-negative integers (or both are -1), indicating that in the game between **team1** and **team2**, **team1** scores **x** goals and **team2** scores **y** goals. If both **x** and **y** are -1, then the game has not yet been played. At most 12 games will not have been played yet.

The input is terminated with **n = m = 0**.

Output

For each team in the same order as the team list in the input, print one line of the following form:

```
Team XXX can finish as high as nth place and as low as mth
place.
```

Use **st**, **nd**, and **rd** instead of **th** for first, second, and third place, respectively. Print a blank line between cases.

Sample input

```
4 6
ManUnited
Arsenal
Chelsea
Tottenham
ManUnited vs Arsenal: 3 1
Chelsea vs Arsenal: 2 2
ManUnited vs Chelsea: 1 0
Tottenham vs ManUnited: -1 -1
Tottenham vs Chelsea: 0 4
Tottenham vs Arsenal: -1 -1
0 0
```

Sample Output

```
Team ManUnited can finish as high as 1st place and as low as 1st place.
Team Arsenal can finish as high as 2nd place and as low as 4th place.
Team Chelsea can finish as high as 2nd place and as low as 3rd place.
Team Tottenham can finish as high as 1st place and as low as 4th place.
```

Problem G: Railroad

Source: `railroad.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

A train yard is a complex series of railroad tracks for storing, sorting, or loading/unloading railroad cars. In this problem, the railroad tracks are much simpler, and we are only interested in combining two trains into one.

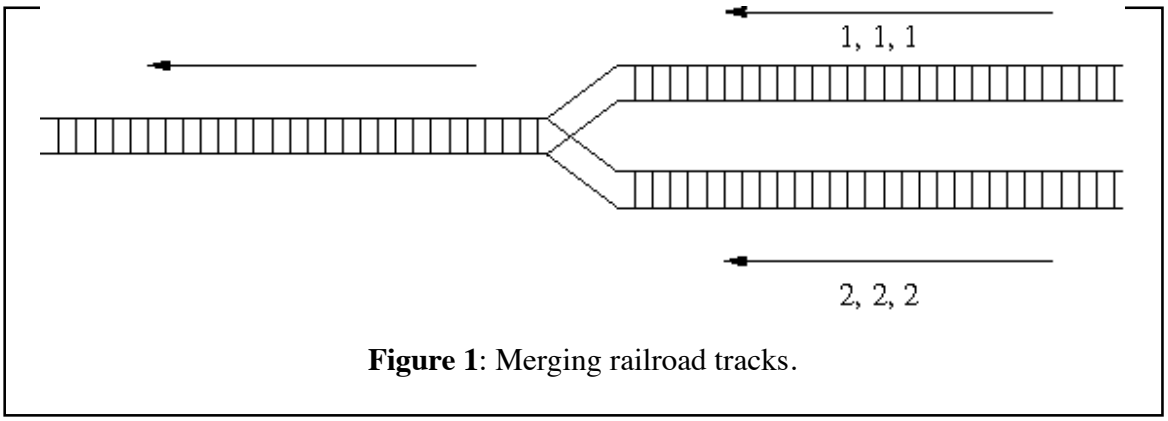

Figure 1: Merging railroad tracks.

The two trains each contain some railroad cars. Each railroad car contains a single type of products identified by a positive integer up to 1,000,000. The two trains come in from the right on separate tracks, as in the diagram above. To combine the two trains, we may choose to take the railroad car at the front of either train and attach it to the back of the train being formed on the left. Of course, if we have already moved all the railroad cars from one train, then all remaining cars from the other train will be moved to the left one at a time. All railroad cars must be moved to the left eventually. Depending on which train on the right is selected at each step, we will obtain different arrangements for the departing train on the left. For example, we may obtain the order 1,1,1,2,2,2 by always choosing the top train until all of its cars have been moved. We may also obtain the order 2,1,2,1,2,1 by alternately choosing railroad cars from the two trains.

To facilitate further processing at the other train yards later on in the trip (and also at the destination), the supervisor at the train yard has been given an ordering of the products desired for the departing train. In this problem, you must decide whether it is possible to obtain the desired ordering, given the orders of the products for the two trains arriving at the train yard.

Input

The input consists of a number of cases. The first line contains two positive integers N_1 N_2 which are the number of railroad cars in each train. There are at least 1 and at most 1000 railroad cars in each train. The second line contains N_1 positive integers (up to 1,000,000) identifying the products on the first train from front of the train to the back of the train. The third line contains N_2 positive integers identifying the products on the second train (same format as above). Finally, the fourth line contains N_1+N_2 positive integers giving the desired order for the departing train (same format as above).

The end of input is indicated by $N_1 = N_2 = 0$.

Output

For each case, print on a line **possible** if it is possible to produce the desired order, or **not possible** if not.

Sample input

```
3 3
1 2 1
2 1 1
1 2 1 1 2 1
3 3
1 2 1
2 1 2
1 1 1 2 2 2
0 0
```

Sample Output

```
possible
not possible
```

Problem H: Post Office

Source: `postoffice.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

Other than postcards, the post office department of some country recognizes three classes of mailable items: letters, packets, and parcels. The three dimensions of a mailable item are called length, height and thickness, with length being the largest and thickness the smallest of the three dimensions.

A letter's length must be at least 125mm but not more than 290mm, its height at least 90mm but not more than 155mm, and its thickness at least 0.25mm but not more than 7mm. (The unit millimeter is abbreviated by mm.)

All three of a packet's dimensions must be greater than or equal to the corresponding minimum dimension for a letter, and at least one of its dimensions must exceed the corresponding maximum for a letter. Furthermore, a packet's length must be no more than 380mm, its height no more than 300mm, and its thickness no more than 50mm.

All three of a parcel's dimensions must be greater than or equal to the corresponding minimum dimension for a letter, and at least one of its dimensions must exceed the corresponding maximum for a packet. Furthermore, the parcel's combined length and girth may not exceed 2100mm. (The girth is the full perimeter measured around the parcel, perpendicular to the length.)

Input

The input will contain data for a number of problem instances. For each problem instance, the input will consist of the three dimensions (measured in mm) of an item, in any order. The length and width will be positive integers. The thickness will be either a positive integer or a positive floating point number. The input will be terminated by a line containing three zeros.

Output

For each problem instance, your program will classify the item as **`letter`**, **`packet`**, **`parcel`** or **`not mailable`**. This verdict will be displayed at the beginning of a new line, in lower case letters.

Sample input

```
100 120 100
0.5 100 200
100 10 200
200 75 100
0 0 0
```

Sample Output

```
not mailable
letter
packet
parcel
```

Problem I: Aronson

Source: `aronson.{c,cpp,java}`
Input: `console {stdin,cin,System.in}`
Output: `console {stdout,cout,System.out}`

Aronson's sequence $\mathtt{a_k}$ is a sequence of integers defined by the sentence "t is the first, fourth, eleventh, ... letter of this sentence.", where the ... are filled in appropriately so that the sentence makes sense. The first few values are 1, 4, 11, 16, 24, 29, 33, 35, 39, ... Note the non-letter characters and spaces are not considered in the formulation of the sequence. When $\mathtt{k \le 100000}$, it turns out that $\mathtt{a_k \le 1000000}$.

To formulate the sequence, you must be able to write the ordinal numbers in English. The ordinal numbers are first, second, third, ..., while the cardinal numbers are one, two, three, ... It is easiest to define the ordinals in terms of the cardinals, so we describe these first.

A cardinal number less than twenty is written directly from the first two columns of **table 1** (3→three, 17→seventeen, etc.). A cardinal number greater than or equal to twenty, but less than one hundred is written as the tens part, along with a nonzero ones part (40→ forty, 56→fifty six, etc). A cardinal number greater than or equal to one hundred, but less than one thousand, is written as the hundreds part, along with a nonzero remainder (100→one hundred, 117→one hundred seventeen, 640→six hundred forty, 999→nine hundred ninety nine). A cardinal number greater than or equal to one thousand, but less than one million, is written as the thousands part, along with a nonzero remainder (12345→twelve thousand three hundred forty five). An ordinal number is written as a cardinal number, but with the last word ordinalized using the columns three and four of **table 1**.

Some example ordinal numbers are 3^{rd}→third, 56^{th}→fifty sixth, 100^{th}→one hundredth, and 12345^{th}→twelve thousand three hundred forty fifth.

Input

The input consists of a number of cases. Each case is specified by a positive integer **k** on one line (**1 ≤ k ≤ 100000**). The sequence of **k** values will be non-decreasing. The input is terminated by a line containing a single **0**.

Output

For each **k**, print the value of a_k on one line. The values of a_k will be at most **1000000**.

Sample input

```
1
3
9
0
```

Sample Output

```
1
11
39
```

n	cardinal	n^{th}	ordinal
1	one	1^{st}	first
2	two	2^{nd}	second
3	three	3^{rd}	third
4	four	4^{th}	fourth
5	five	5^{th}	fifth
6	six	6^{th}	sixth
7	seven	7^{th}	seventh
8	eight	8^{th}	eighth
9	nine	9^{th}	ninth
10	ten	10^{th}	tenth
11	eleven	11^{th}	eleventh
12	twelve	12^{th}	twelfth
13	thirteen	13^{th}	thirteenth
14	fourteen	14^{th}	fourteenth
15	fifteen	15^{th}	fifteenth
16	sixteen	16^{th}	sixteenth
17	seventeen	17^{th}	seventeenth
18	eighteen	18^{th}	eighteenth
19	nineteen	19^{th}	nineteenth
20	twenty	20^{th}	twentieth
30	thirty	30^{th}	thirtieth
40	forty	40^{th}	fortieth
50	fifty	50^{th}	fiftieth
60	sixty	60^{th}	sixtieth
70	seventy	70^{th}	seventieth
80	eighty	80^{th}	eightieth
90	ninety	90^{th}	ninetieth
100	hundred	100^{th}	hundredth
1000	thousand	1000^{th}	thousandth

Table 1: Translation table.