Contents

1 Web Search Engine **3**

2 Web Page Encryption **4**

3 Web Server Cache **5**

4 Finger-Friendly Domain Names **6**

5 Web Browser Window Placement **8**

6 Formatting HTML Text **10**

7 Web Reliable **12**

8 Wireless Interference Zones **14**

0 Word Extraction

You are applying for a job at a start-up company called *FindIt.com*. To determine whether you are a proficient programmer, they give you thirty minutes to write a short program in C++.

Your program must read in pairs of words and numbers. The number specifies how many characters to trim off the beginning of the word. You should print the remainder of each word on a separate line, printing a blank line if the word is completely trimmed.

Input Format

The input will consist of pairs of words and numbers. The input will be terminated by a line containing only -1.

Output Format

Your program should output each trimmed word on a separate line, printing a blank line if the word is trimmed completely.

Example

Input:

```
university 3 of 5
maryland 4
programming 3
contest 3
-1
```

Output:

```
versity

land
gramming
test
```

1 Web Search Engine

You have begun working at a start-up company called *FindIt.com*. Your company which sells customized web search engines and web servers. Your lead programmer caught the flu right before product ship deadline, and you have been called in to finish implementing a web search engine. The existing code *crawls* through the web, fetching web pages. Your job is to rank them, based on keywords.

For each web page, you need to score the page by finding the number of times a specific *keyword* appears in the page. Keywords are allowed to be part of other words, so the keyword **net** appears twice in the sentence "Networks and intranets". Keywords are allowed to overlap, so **ana** should be reported as appearing twice in "banana".

Input Format

The input will consist of a keyword on its own line, followed by words from a web page on one or more lines. The input will be terminated by a line containing only −1. You may assume the keyword and all input will be in lowercase.

Output Format

Your program should output the result of your search in the following form:
Found $< n >$ <*keyword*>

Example

Input:

```
net
surfing the net today to
learn etiquette about
networks and intranets
-1
```

Output:

```
Found 3 net
```

Input:

```
ana
bananas can be
found in anapolis
-1
```

Output:

```
Found 3 ana
```

2 Web Page Encryption

After successfully completing your search engine, you are put on a new job. It turns out a rival company is trying to steal your customers by providing direct links to your web pages from their web site. Your job is to *encrypt* your web pages so that only your web server can access them.

You remember that the ancient Romans used a *Caesar cipher*, where each letter is replaced by a letter k later in the alphabet (shifting it by k positions). Shifts occur in alphabetical order, between the letters "a" and "z". If a letter is shifted past "z", it starts back at "a" and continues shifting. For example, using a caesar cipher where $k = 2$, the word "car" is transformed into "ect", while the word "yaz" is shifted to "zcb".

You decide to go one better. Instead of shifting all letters by the same distance, you will shift each letter by a different amount, based on a code word. Each letter in the code word is converted into a shift amount, based on how many letters it is after "a". Text to be encrypted is then shifted by the amount of each letter in the keyword, repeating as necessary.

For instance, suppose your keyword is "bad". The shift amounts for the letters are then 1, 0, and 3, respectively. You would encrypt the string "carrot" as "dausow", shifting "c" by one, "a" by zero, and "r" by three letters, then repeating for "r", "o", and "t".

Input Format

The input will begin with your keyword on a single line. It will then consist of a series of words to be encrypted using the keyword, each on a separate line. The input will be terminated by a line containing only -1. You may assume that the keyword and all input will be in lowercase.

Output Format

For each word to be encrypted, output a line containing the encrypted word.

Example

Input:

```
bad
carrot
aaaaa
-1
```

Output:

```
dausow
badba
```

Input:

```
umd
mclj
rizq
-1
```

Output:

```
good
luck
```

3 Web Server Cache

While working at your start-up company, you are assigned the job of improving the performance of the company web server. You notice that users frequently browse the same web pages at your site. Being smart, you realize you can speed up the performance of your web server by keeping the most frequently accessed web pages in a *cache*, which can be uploaded more quickly than pages stored in memory.

Your job is to set up the web server for a hundred web pages numbered from 1 to 100. As requests come in, you must decide which web page to keep in the cache. At the end you should report on the total number of requests and the number of requests which were supplied from the cache.

The problem you must face is your cache is small, and can hold only four web pages at a time. You must thus decide which pages to keep in the cache instead of memory. You decide to apply the "least recently used" (LRU) algorithm, where you throw out the web page which was used furthest in the past whenever space is needed to store a new web page.

Input Format

The input will consist of a sequence of requests for web pages 1 to 100, each on a separate line. The sequence will be terminated by a line containing only -1.

Output Format

For each web page request, display the contents of your web cache. Print on a single line the numbers of the web pages in your cache, from most recent to least recently used. You should print 0 for empty pages in your cache. When all requests have been served, output a line displaying the number of service requests satisfied from cache and the total number of requests, in the form "X of Y requests found in cache".

Example

Input: **Output:**

```
1           1 0 0 0
2           2 1 0 0
1           1 2 0 0
-1          1 of 3 requests found in cache
```

Input: **Output:**

```
1           1 0 0 0
5           5 1 0 0
21          21 5 1 0
44          44 21 5 1
8           8 44 21 5
21          21 8 44 5
8           8 21 44 5
5           5 8 21 44
-1          3 of 8 requests found in cache
```

4 Finger-Friendly Domain Names

Your Internet startup needs a domain name. Your market research guru tells you that users tend to avoid sites with names that are hard to type. Thus, you must pick a name that is easy to type. For your research, you decide to use the idealized keyboard model suggested by Figure 1. Note that the keys are perfectly aligned both horizontally and vertically, unlike the skewed positioning of keys in typical keyboards. You decide to assign a *typing cost* to each domain name (or any word, in general) based on the following rules.

- Consecutive characters are easier to type if they use different hands. Therefore, we say that each pair of consecutive characters that uses the same typing hand contributes 1 unit to the typing cost. For example, typing `d` followed by `a` adds 1 to the this component of the typing cost because `d` and `a` are both typed using the left hand. On the other hand, typing `n` followed by `e` does not contribute anything to this component of the typing cost because `n` and `e` are typed using different hands. An *exception* to this rule is when the two consecutive characters are identical; such a pair of characters does not contribute anything to this component of the typing cost (e.g., the consecutive `n`s in `cnn.com`).
- We assume that the typist begins in the neutral touch typing position, with the fingers of the left hand over the keys `a`, `s`, `d`, and `f`, and the fingers of the right hand over the keys `j`, `k`, `l`, and `;`. In order to type characters that are not in this middle row of keys, the hands are moved to the top and bottom rows. For example, to type `r`, the left hand is moved to the top row. Similarly, to type `.` (dot), the right hand is moved to the bottom row. Each such movement costs twice the number of rows moved. We assume that after typing a character, the hand remains over that character's row. For example, after typing `r`, the left hand stays over the top row until we need to move it in order to type another character with the left hand. Thus, if the `r` is followed by characters `l` and `o`, which are typed using the right hand, the left hand remains on the top row. For example, in order to type `x` after typing `r` and `u`, the left hand must move from the top row (used to type `r`) to the bottom row (for `x`); at a cost of 4 units (twice the distance of two rows).
- Typing the characters `t`, `g`, `b`, `y`, `h`, and `n` requires stretching the index finger. Therefore, each occurrence of any of these characters adds 3 units to the typing cost. An *exception* to this rule is when any of these characters is repeated in succession; in such a case, only the first occurrence of the character is charged the cost of 3 units. For example, typing `bgth` incurs a cost of 12 units (3 times 4) due to stretching. However, typing `bbbbbhhhhh` incurs a cost of only 6 units (3 times 2) due to stretching (once for the `b`s and once for the `h`s). Each of the characters in `bhbhbh` incurs a stretching cost of 3 units (for a total of 18) because the `b`s and `h`s are not contiguous.

Your goal is to write a program that takes a list of words (presumably candidate domain names) as input and outputs their typing costs. If the word contains a character that is absent from the idealized keyboard suggested by Figure 1, you should replace that character with the `,` (comma) character for the purpose of calculating the typing cost. For example, the typing cost of `foo-bar_baz` is calculated by applying the above rules to `foo,bar,baz` (obtained by replacing the `-` and `_` characters with `,`). Similarly, if the input contains uppercase

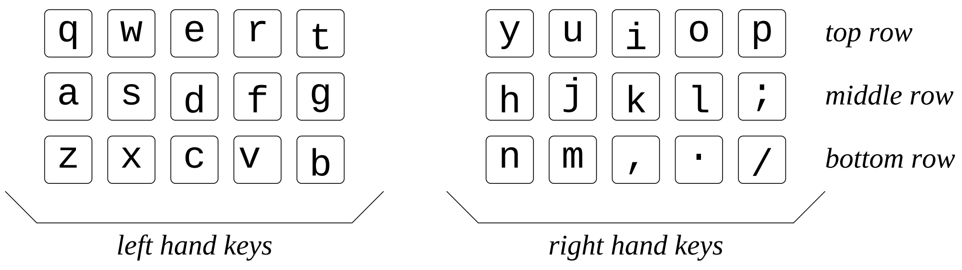


Figure 1: The idealized keyboard for Question 4

characters (e.g., `DiScO.oRG`), you should replace them with the corresponding lowercase characters (`disco.org`) for calculating the typing cost. You may assume a maximum word length of 256 characters.

Input Format

The input consists of a sequence of words (e.g., foo, bar, barney.com, abra.cadabra.gov.jp) separated by white space (space, tab, or newline). The end of the sequence is denoted using the reserved word "." (a single period). A sample input is displayed below:

```
UMD.edu  TheNextBigThing.com  abra.cadabra.gov.jp
database.org www.linux.org .
```

Output Format

Your program should output the typing costs of the input words (in the same order as the input words). It should print each cost on a separate line. The required output on the above input is displayed below:

```
16
81
56
36
46
```

Example

Consider the first domain name in our sample input: `UMD.edu`. First, we convert the uppercase letters to lowercase, yielding `umd.edu`. The pairs (`u`,`m`) and (`e`,`d`) are consecutive characters that must be typed using the same hand. Therefore, each pair contributes 1 unit to the typing cost, for a total of 2. Further, typing `u` requires moving the right hand from the middle row to the top row, at a cost of twice the distance (1 row): 2 units. Next, typing `m` requires moving the right hand from the top row to the bottom row, at a cost of twice the distance (2 rows): 4 units. Typing the next `d` does not require moving the left hand since it is over the middle row initially. Typing `.` does not require moving the right hand since it is already over the bottom row (from typing `m` earlier). Typing `e` requires moving the left hand from the middle row to the top row, at a cost of 2 units (twice the distance moved). Similarly, typing `d` requires moving the left hand back to the middle row, at a cost of 2 units. Typing the final `u` requires moving the right hand from the bottom row to the top row, at a cost of 4 units. Thus, the total cost of hand movements between rows is $2+4+2+2+4 = 14$ units. Since `umd.edu` does not contain any of the characters that require stretching the index finger for typing, there is no additional cost, and the final typing cost of `umd.edu` is 16 units (2 units due to pairs of characters and 14 units due to hand movements between rows).

5 Web Browser Window Placement

You are the chief programmer for *easyBrowse Inc.* — the company that will revolutionize the world by creating a better faster easier web browser. Now all the new-fangled web pages keep popping up more and more windows each time you visit their web-site: your job today is to find the best place on the screen to place these new windows so that they cause your customers the least pain. After talking to the marketing people, the psychology people, the human-computer interaction people, the advertising people, and the marketing people again, you have come up with the following rules for placement of windows for easy browsing (tm):

- All windows must be placed *completely* within the screen. For your test window placement code, you should assume the screen is 320×240 pixels wide. Assume that in the beginning, there are no windows on the screen. This also means that a single windows cannot be larger than 320×240; also, windows must be at least one pixel in both width and height.
- You want to place a window such that you occlude (overlap) the minimum number of *pixels* of existing windows. For example, suppose you have two legal choices (x, y), and $(x', y)'$ to place your window such that the complete window is visible on-screen. Suppose by placing your window at (x, y), you cover up 10 pixels of existing windows, while placing the window at (x', y'), you cover up 42 pixels: you should place the window at (x, y). You should minimize the total the number of pixels your placement obstructs — you do *not* have to worry about the number of windows.
- If you find two points on the screen such that the occlusion (overlap with number of pixels of other windows) is minimum, you should break the tie by choosing the point that has the smallest y coordinate. If the y coordinates are the same, then you should break the tie by choosing the point with the smaller x coordinate. This will ensure that windows are placed as close to the top (and left) of the screen as possible. Note that this implies that the first window you place will always be at $(0, 0)$.

Input Format

The input to your program is a list of window sizes:

```
10 10
320 20
319 230
319 230
120 23
-1  -1
```

The first number on a line is the width of the window, while the second number is the height of the window. Note that you can be multiple given windows of a given size and you don't a-priori know the maximum number of windows you may have to place. A `-1 -1` entry signifies end of input.

Output Format

The output of your program is the placement of the windows using the following format:

```
New window of size  10 x  10: Final placement: [(  0   0), (  9   9)]
New window of size 320 x  20: Final placement: [(  0  10), (319  29)]
New window of size 319 x 230: Final placement: [(  0  10), (318 239)]
New window of size 319 x 230: Final placement: [(  1   0), (319 229)]
New window of size 120 x  23: Final placement: [(200 217), (319 239)]
```

Note that you are required to print the exact pixels that the top-left and bottom-right corners of each window occupy. Further, the top-left pixel is coordinate $(0, 0)$ and the bottom right pixel is coordinate $(319, 239)$.

Example

Lets walk through what happens when we try to place the set of windows given the previous section (we do not show the `-1 -1` delimiting input):

- In the beginning, there are no windows on the screen, so the first window takes is placed at $(0,0)$.
- The next window is a long thin window, and cannot be fit in the top ten "lines" without occluding the first window. This window is placed at $(0,10)$.
- The next two windows are truly massive: the first one is placed to use up the unused space at the bottom of the screen. The next window is the same size. It is placed at $(1,0)$ since this placement allows the placement algorithm to use the unused space in the top right part of the screen.
- The last window is placed aligned with the bottom right part of the screen since this is now the least used part of the screen.

More examples

```
      Input                                    Output
------------------------------------------------------------------------------------------
100 25         New window of size 100 x  25: Final placement: [(  0   0), ( 99  24)]
100 25         New window of size 100 x  25: Final placement: [(100   0), (199  24)]
100 25         New window of size 100 x  25: Final placement: [(200   0), (299  24)]
100 25         New window of size 100 x  25: Final placement: [(  0  25), ( 99  49)]
100 25         New window of size 100 x  25: Final placement: [(100  25), (199  49)]
100 25         New window of size 100 x  25: Final placement: [(200  25), (299  49)]
100 25         New window of size 100 x  25: Final placement: [(  0  50), ( 99  74)]
-1 -1
------------------------------------------------------------------------------------------
12 24          New window of size  12 x  24: Final placement: [(  0   0), ( 11  23)]
32 35          New window of size  32 x  35: Final placement: [( 12   0), ( 43  34)]
100 25         New window of size 100 x  25: Final placement: [( 44   0), (143  24)]
1 1            New window of size   1 x   1: Final placement: [(144   0), (144   0)]
33 2           New window of size  33 x   2: Final placement: [(145   0), (177   1)]
44 2           New window of size  44 x   2: Final placement: [(178   0), (221   1)]
55 124         New window of size  55 x 124: Final placement: [(222   0), (276 123)]
-1 -1
```

6 Formatting HTML Text

One of the things that a web browser needs to do is to display text in a nicely formatted manner. For this problem, you are to write text formatting program. Your program will input a paragraph of text. This text will consist of a series of words separated by spaces and/or newlines (you may assume that there are no tab characters). A word may contain punctuation characters. The text is terminated by special word "`$$$`", which appears alone on the last line. For example consider the text:

```
The quick brown-fox jump$
   over, the   ---               lazy

dog.
$$$
```

The words are "`The`", "`quick-brown`", "`fox`", "`jump$`", "`over,`", "`the`", "`---`", "`lazy`", and "`dog.`". Note that final "`$$$`" is not considered a word.

Your program will also be given an integer indicating the window width w. You are to output the words with a minimum number of spaces between them so that:

(1) No line has more than w total characters (excluding the newline character).

(2) For each two words on the same line there is at least one space between them.

(3) The leftmost character is nonblank and, except for the last line, the rightmost character is nonblank.

(4) The spaces are to be distributed as evenly as possible between the words. If you cannot generate exactly the same number of spaces between consecutive words, then place the larger number of spaces near the right end of the line. For example (as shown in the line with "jump$" below) if you have 4 words and hence 3 gaps between them, and you need a total of 5 spaces to fill up the line, then the spaces should be distributed 1–2–2 in the gaps.

Given a width of 21, the above text would be displayed as follows.

```
123456789012345678901          <--- This reference line is not printed
The  quick  brown-fox
jump$ over,  the  ---
lazy dog.
```

Input format

The first line of the input contains just the input width w. Starting with the next line, the text is given. The last line of the input contains the single word "`$$$`". This word appears nowhere else in the text. As a simplifying assumption you may assume no two consecutive words are so long that they cannot both fit on a single line with a single space between them. You may assume that each word has at most 50 characters, that there are at most 5,000 words total, and that w is at most 100.

Output format

The output consists of the words of text. There should be no leading spaces or trailing spaces.

Example

Input:

```
21
The quick brown-fox jump$
   over, the   ---                lazy

dog.
$$$
```

Output:

```
The  quick  brown-fox
jump$ over,  the  ---
lazy dog.
```

Input:

```
30
When in the Course of human events,
it becomes necessary for one people
to dissolve   the political bands
which have connected them with another,
and to   assume among the
powers of the earth, the separate
and equal station   to which
the Laws of Nature and of
Nature's   God entitle them, a decent
respect to the opinions of mankind
requires   that   they should
declare the causes which impel
them to the   separation.
$$$
```

Output:

```
When in the  Course  of  human
events, it  becomes  necessary
for one people to dissolve the
political  bands  which   have
connected them  with  another,
and to assume among the powers
of the earth, the separate and
equal  station  to  which  the
Laws of Nature and of Nature's
God  entitle  them,  a  decent
respect  to  the  opinions  of
mankind  requires  that   they
should  declare   the   causes
which  impel   them   to   the
separation.
```

7 Web Reliable

You are the chief network designer for *Web Reliable*, yet another new ISP that will revolutionize the web as we know it. *Web reliable* promises to transform the current world-wide-wait into a usable functioning information infrastructure. Their claim is simple: *Web reliable* provides the most reliable network paths. How? Well, thats where you come in.... The marketing people got a bit carried away and have started a huge ad. campaign without all the technical problems being completely solved. The research lab has come up with a way to find out how reliable each network link in the *Web reliable* network is. Your job is to use that information and deliver on the *Web reliable* promise, i.e., find the most reliable paths.

- Since you have signed all the non-disclosure agreements, *Web reliable* will give you a complete “map” of their network. As usual, this network will consist of a set of nodes (routers) and a set of edges (links). Of course, you will also have the information about the failure probabilities of each link.
- The *Web reliable* network consists of n routers, (numbered 0 ...n-1). All the *Web reliable* servers are connected to router 0. Since data comes from the *Web reliable* servers, you will only have to find the most reliable path from router 0 to all the other routers.
- If you find two paths with the same reliability, you should choose the path with the smaller number of hops (i.e. you should choose the path that uses lesser number of links). If you find paths with the same number of hops and the same reliability, you should produce the lexicographically smallest path. For example, if at any point, you could have taken a path through router a and router b, you should choose a if and only if $a < b$. (Recall that router ids are non-negative integers). An example of this tie-breaking rule is given later.

Input Format

The input to your program is a description of the *Web reliable* network

```
3 3
0 1 0.5
0 2 0.9
1 2 0.5
```

The network described by the input file corresponds to the network shown in Figure 2. The first line lists

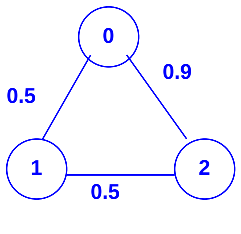


Figure 2: Sample network

the total number of nodes in the network, and the total number of edges. Each remaining line consists of two node numbers n m (representing a edge between n and m) and the probability the edge may fail.

Edges are bi-directional: they are specified by a pair of nodes and a probability of failure. The probability of failure is a floating point number in the range $(0, 1]$. Thus, the first line `0 1 0.5` means that there is a link between routers 0 and 1 with probability of link failure equal to 0.5.

The number of edges lines must be equal to the number of edges specified in the beginning. Also, the node identifiers must be in the range 0 .. (`NUM_NODES` - 1).

You are guaranteed that there is at least one path from node 0 to every other node. There are no self loops (i.e. an edge starting and ending at the same node).

Output Format

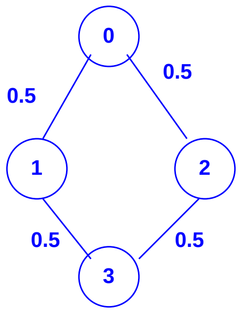
The output of your program is a set of paths and their associated success probabilities from node 0 in the following format:

```
From node 0
  to node 1: Prob.(Success) = 0.50, Path: 0 -> 1
  to node 2: Prob.(Success) = 0.25, Path: 0 -> 1 -> 2
```

Note that in this example, even though there was a direct link from 0 to 2, the most reliable path went through node 1.

Examples

Example 0 In this example, the path to node 3 goes through node 1 (and not node 2) because $\{0, 1, 3\}$ is "lexicographically smaller" than $\{0, 2, 3\}$. Stated differently, the algorithm could choose either 1 or 2 at node 0: it chose 1 since $1 < 2$.



Input:

Figure 3: Example network

```
4 4
0 1 0.5
0 2 0.5
1 3 0.5
2 3 0.5
```

Output:

```
From node 0
  to node 1: Prob.(Success) = 0.50, Path: 0 -> 1
  to node 2: Prob.(Success) = 0.50, Path: 0 -> 2
  to node 3: Prob.(Success) = 0.25, Path: 0 -> 1 -> 3
```

8 Wireless Interference Zones

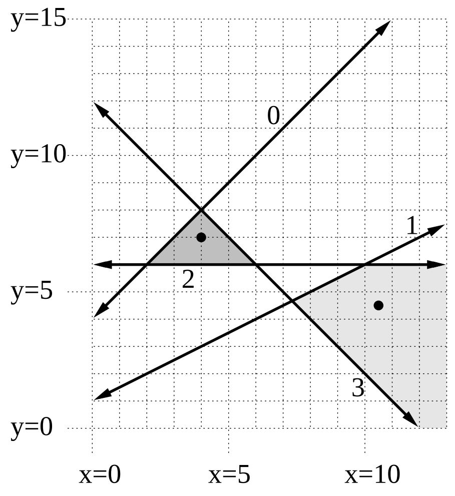
Wireless communication can be affected by regions of interference, due to effects such as the presence of magnetic fields near power lines. In this assignment you are give a set of lines describing where interference is present, and you are to determine where a user can roam without encountering interference. Each line is defined by the equation

$$y = ax + b,$$

where a and b are two real numbers. Think of each line as defining a *interference wall* of infinite length. This wall splits every point in the plane into one of three classes, those lying *above* the wall (having larger y-coordinate), those lying below (having smaller y-coordinate), and those lying on the wall. Altogether, these walls split the plane into regions, called *zones*.

A wireless *user* is given by a point (x, y), which we assume does not lie on one of the walls. Your task is to determine which zone contains the user. The zone is described by indicating those lines that make up the boundary of the zone.

For example, consider the walls shown in the figure below. The user at point $(4, 7)$ is in the zone (darker shaded) bounded by walls 0, 2, and 3 (note that wall 1 does not touch the zone). The user at point $(10.5, 4.5)$ is in the zone (lighter shaded) bounded by walls 1, 2, and 3.



Wall:	Equation:
0:	y = x + 4
1:	y = x/2 + 1
2:	y = 6
3:	y = -x + 12

Write a program which inputs a list of walls, given by their a and b values, and a list of users, given by their (x, y) coordinates. For each user, determine which zone it lies in. Output only the walls that bound this zone, and indicate whether the user is above or below each such wall.

Input format

The first line of input contains the number of walls, say n. After this there are n lines, each containing the a and b value for the coefficients of a wall. After this there is a list of users, each given by its x and y coordinates on one line. The list of users is terminated by the coordinates “999 999”. You may assume that there are at most 100 walls and 100 users. Point coordinates and line coefficients should be represented as doubles. You should not make any assumptions about the sizes of these numbers.

You may make the following assumptions about the positions of walls.

- There are no vertical walls (slope equals infinity).
- No user lies on a wall (it will either be strictly above or below).
- No two walls are parallel to each other and no two walls are collinear with each other.
- Three walls do not intersect in the same point.

Output format

For each user you should output the coordinates of the user on the first line. Then on separate lines give the wall indices (numbered from 0 to $n-1$) which bound the zone. The walls must be given in increasing order of wall index. Points should be output in the form "(x, y)" (note the space between the comma and the y coordinate).

Example

The example below shows the input and output for the situations shown in the figure above.

Input :

```
4
 1.0  4.0
 0.5  1.0
 0.0  6.0
-1.0 12.0

 4.0  7.0
10.5  4.5
999 999
```

Output:

```
The zone for (4, 7) is bounded by walls
    0
    2
    3
The zone for (10.5, 4.5) is bounded by walls
    1
    2
    3
```