Problem 1: Booklet Printing

Source file: `booklet.{c, cpp, pas}`
Input file: `booklet.in`
Output file: `booklet.out`

When printing out a document, normally the first page is printed first, then the second, then the third, and so on until the end. However, when creating a fold-over booklet, the order of printing must be altered. A fold-over booklet has four pages per sheet, with two on the front and two on the back. When you stack all the sheets in order, then fold the booklet in half, the pages appear in the correct order as in a regular book. For example, a 4-page booklet would print on 1 sheet of paper: the front will contain page 4 then page 1, and the back will contain page 2 then page 3.

```
Front             Back
-------------     -------------
|     |     |     |     |     |
|  4  |  1  |     |  2  |  3  |
|     |     |     |     |     |
-------------     -------------
```

Your task is to write a program that takes as input the number of pages to be printed, then generates the printing order.

The input file contains one or more test cases, followed by a line containing the number 0 that indicates the end of the file. Each test case consists of a positive integer *n* on a line by itself, where *n* is the number of pages to be printed; *n* will not exceed 100.

For each test case, output a report indicating which pages should be printed on each sheet, exactly as shown in the example. If the desired number of pages does not completely fill up a sheet, then print the word `Blank` in place of a number. If the front or back of a sheet is entirely blank, do not generate output for that side of the sheet. Output must be in ascending order by sheet, front first, then back.

Example input:

```
1
14
4
0
```

Example output:

```
Printing order for 1 pages:
Sheet 1, front: Blank, 1
Printing order for 14 pages:
Sheet 1, front: Blank, 1
Sheet 1, back : 2, Blank
Sheet 2, front: 14, 3
Sheet 2, back : 4, 13
Sheet 3, front: 12, 5
Sheet 3, back : 6, 11
Sheet 4, front: 10, 7
Sheet 4, back : 8, 9
Printing order for 4 pages:
Sheet 1, front: 4, 1
Sheet 1, back : 2, 3
```

Problem 2: Finding Rectangles

Source file: `rect.{c, cpp, pas}`
Input file: `rect.in`
Output file: `rect.out`

Consider the point sets in figures 1a, 2a, and 3a. Using only those points as vertices, figures 1b, 2b, and 3b show all the rectangles that can be formed with horizontal and vertical sides. No rectangles can be formed from the points in figure 4.

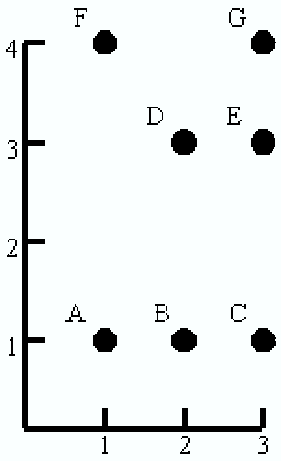


Figure 1a

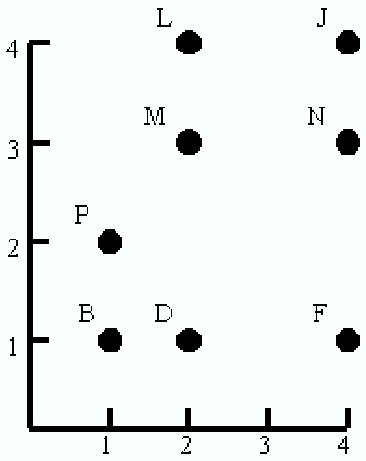


Figure 2a

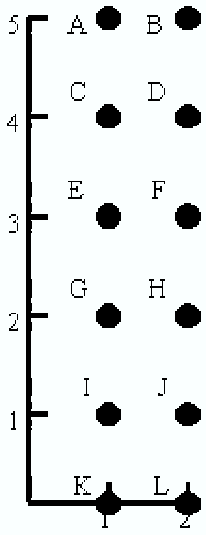


Figure 3a

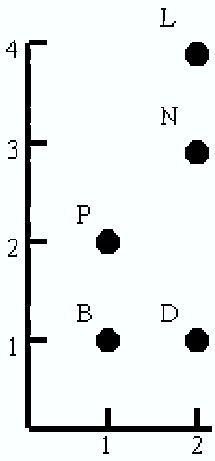


Figure 4

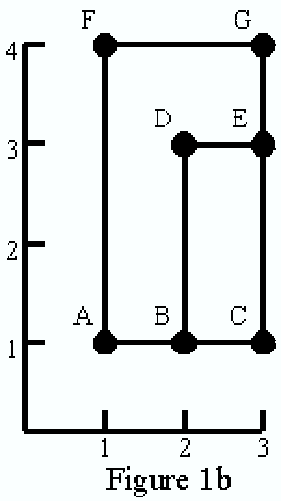


Figure 1b

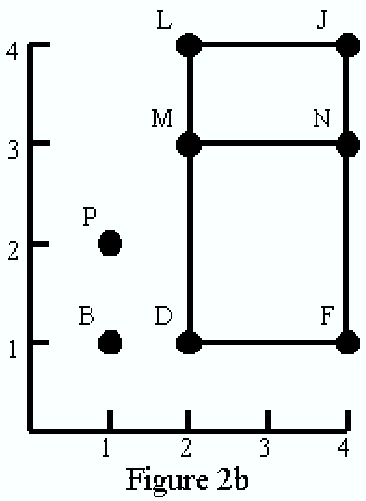


Figure 2b

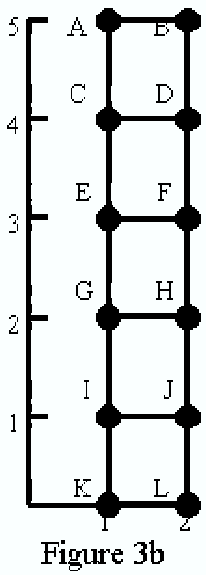


Figure 3b

Your task is to write a program that can find all rectangles that can be formed from a given set of points. The example input and output given below correspond to the figures above.

The input file contains one or more point sets, followed by a line containing the number 0 that signals the end of the file. Each point set begins with a line containing ***n***, the number of points, and is followed by ***n*** lines that describe the points. Each point description contains a capital letter that is the label of the point, then a space, the horizontal coordinate, a space, and the vertical coordinate. Within each set, points labels occur in alphabetical

order. Note that since each point is labelled with a capital letter there can be at most 26 points. All coordinates are nonnegative integers less than 50. Points within a set are unique.

The output for each point set starts with "`Point set `", followed by the number of the point set and a colon. If there are no rectangles, "`  No rectangles`" appears after the colon. If there are rectangles, they are listed starting on the next line. A blank precedes each rectangle. Each rectangle is given by its vertex labels, in clockwise order from the upper left, so the order is upper left, upper right, lower right, lower left. The rectangles are listed ten per line, except for the last line, where there may be as few as one. The rectangles are listed in alphabetical order.

Example input:

```
7
A 1 1
B 2 1
C 3 1
D 2 3
E 3 3
F 1 4
G 3 4
8
B 1 1
D 2 1
F 4 1
J 4 4
L 2 4
M 2 3
N 4 3
P 1 2
12
A 1 5
B 2 5
C 1 4
D 2 4
E 1 3
F 2 3
G 1 2
H 2 2
I 1 1
J 2 1
K 1 0
L 2 0
5
B 1 1
D 2 1
L 2 4
N 2 3
P 1 2
0
```

Example output:

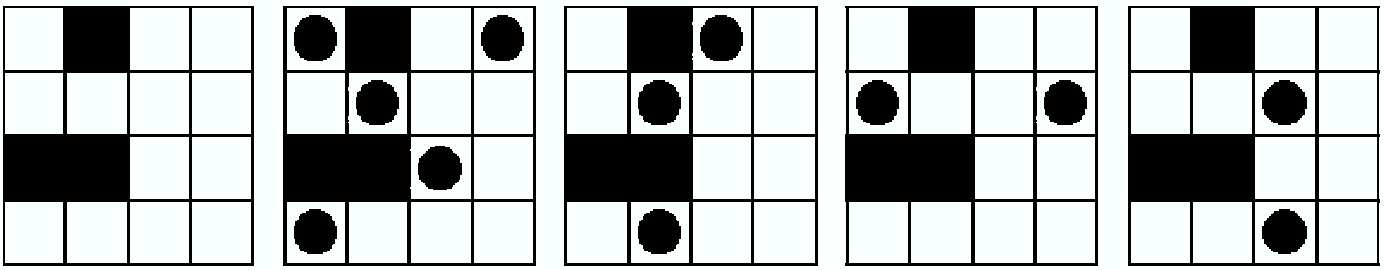
```
Point set 1:
 DECB FGCA
Point set 2:
 LJFD LJNM MNFD
Point set 3:
 ABDC ABFE ABHG ABJI ABLK CDFE CDHG CDJI CDLK EFHG
 EFJI EFLK GHJI GHLK IJLK
Point set 4: No rectangles
```

Problem 3: Don't Get Rooked

Source file: `rook.{c, cpp, pas}`
Input file: `rook.in`
Output file: `rook.out`

In chess, the rook is a piece that can move any number of squares vertically or horizontally. In this problem we will consider small chess boards (at most 4x4) that can also contain walls through which rooks cannot move. The goal is to place as many rooks on a board as possible so that no two can capture each other. A configuration of rooks is ***legal*** provided that no two rooks are on the same horizontal row or vertical column unless there is at least one wall separating them.

The following image shows five pictures of the same board. The first picture is the empty board, the second and third pictures show legal configurations, and the fourth and fifth pictures show illegal configurations. For this board, the maximum number of rooks in a legal configuration is 5; the second picture shows one way to do it, but there are several other ways.



Your task is to write a program that, given a description of a board, calculates the maximum number of rooks that can be placed on the board in a legal configuration.

The input file contains one or more board descriptions, followed by a line containing the number 0 that signals the end of the file. Each board description begins with a line containing a positive integer *n* that is the size of the board; *n* will be at most 4. The next *n* lines each describe one row of the board, with a '`.`' indicating an open space and an uppercase '`X`' indicating a wall. There are no spaces in the input file.

For each test case, output one line containing the maximum number of rooks that can be placed on the board in a legal configuration.

Example input:

```
4
.X..
....
XX..
....
2
XX
.X
3
.X.
X.X
.X.
3
...
.XX
.XX
4
....
....
....
....
0
```

Example output:

```
5
1
5
2
4
```

Problem 4: Self Numbers

Source file: `self.{c, cpp, pas}`
Input file: *none*
Output file: `self.out`

In 1949 the Indian mathematician D.R. Kaprekar discovered a class of numbers called self-numbers. For any positive integer ***n***, define ***d***(***n***) to be ***n*** plus the sum of the digits of ***n***. (The ***d*** stands for ***digitadition***, a term coined by Kaprekar.) For example, ***d***(75) = 75 + 7 + 5 = 87. Given any positive integer ***n*** as a starting point, you can construct the infinite increasing sequence of integers ***n***, ***d***(***n***), ***d***(***d***(***n***)), ***d***(***d***(***d***(***n***))), ... For example, if you start with 33, the next number is 33 + 3 + 3 = 39, the next is 39 + 3 + 9 = 51, the next is 51 + 5 + 1 = 57, and so you generate the sequence

33, 39, 51, 57, 69, 84, 96, 111, 114, 120, 123, 129, 141, ...

The number ***n*** is called a ***generator*** of ***d***(***n***). In the sequence above, 33 is a generator of 39, 39 is a generator of 51, 51 is a generator of 57, and so on. Some numbers have more than one generator: for example, 101 has two generators, 91 and 100. A number with *no* generators is a ***self-number***. There are thirteen self-numbers less than 100: 1, 3, 5, 7, 9, 20, 31, 42, 53, 64, 75, 86, and 97.

Write a program to output all positive self-numbers less than 10000 in increasing order, one per line.

Output:

```
1
3
5
7
9
20
31
42
53
64
 |
 |       <-- a lot more numbers
 |
9903
9914
9925
9927
9938
9949
9960
9971
9982
9993
```