1991 ACM Scholastic Programming Contest Finals
sponsored by AT&T Computer Systems

Problem "A"
Firetruck

The Center City fire department collaborates with the transportation department to maintain maps of the city which reflects the current status of the city streets. On any given day, several streets are closed for repairs or construction. Firefighters need to be able to select routes from the firestations to fires that do not use closed streets.

Central City is divided into non-overlapping fire districts, each containing a single firestation. When a fire is reported, a central dispatcher alerts the firestation of the district where the fire is located and gives a list of possible routes from the firestation to the fire. You must write a program that the central dispatcher can use to generate routes from the district firestations to the fires.

Input and Output

The city has a separate map for each fire district. Streetcorners of each map are identified by positive integers less than 21, with the firestation always on corner #1. The input file contains several test cases representing different fires in different districts. The first line of a test case consists of a single integer which is the number of the streetcorner closest to the fire. The next several lines consist of pairs of positive integers separated by blanks which are the adjacent streetcorners of open streets. (For example, if the pair 4 7 is on a line in the file, then the street between streetcorners 4 and 7 is open. There are no other streetcorners between 4 and 7 on that section of the street.) The final line of each test case consists of a pair of 0's.

For each test case, your output must identify the case by number (case #1, case #2, etc). It must list each route on a separate line, with the streetcorners written in the order in which they appear on the route. And it must give the total number routes from firestation to the fire. *Include only routes which do not pass through any streetcorner more than once.* (For obvious reasons, the fire department doesn't want its trucks driving around in circles.) Output from separate cases must appear on separate lines. The following sample input and corresponding correct output represents two test cases.

Sample Input

```
6
1 2
1 3
3 4
3 5
4 6
5 6
2 3
2 4
0 0
4
2 3
3 4
5 1
1 6
7 8
8 9
2 5
5 7
3 1
1 8
4 6
```

Sample Output

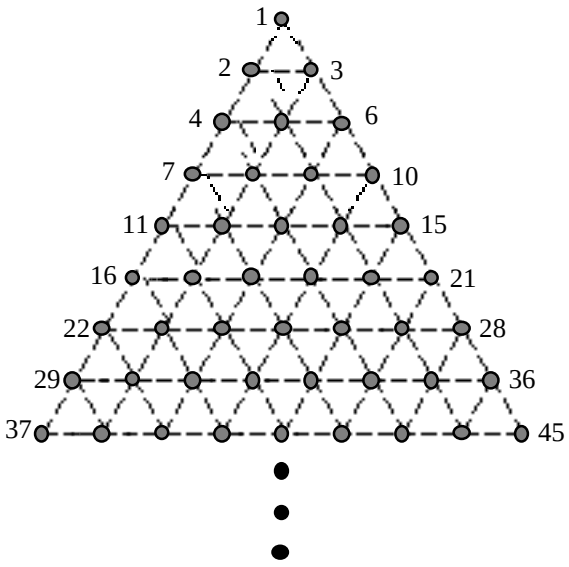
```
CASE 1:
   1   2   3   4   6
   1   2   3   5   6
   1   2   4   3   5   6
   1   2   4   6
   1   3   2   4   6
   1   3   4   6
   1   3   5   6
There are 7 routes from the firestation to streetcorner 6.
CASE 2:
   1   3   2   5   7   8   9   6   4
   1   3   4
   1   5   2   3   4
   1   5   7   8   9   6   4
   1   6   4
   1   6   9   8   7   5   2   3   4
   1   8   7   5   2   3   4
   1   8   9   6   4
There are 8 routes from the firestation to streetcorner 4.
```

continued on reverse side ∅

```
6 9
0 0
```

Problem “B”
Triangular Vertices

Consider the points on an infinite grid of equilateral triangles as shown below:



Note that if we number the points from left to right and top to bottom, then groups of these points form the vertices of certain geometric shapes. For example, the sets of points {1,2,3} and {7,9,18} are the vertices of triangles, the sets {11,13,26,24} and {2,7,9,18} are the vertices of parallelograms, and the sets {4,5,9,13,12,7} and {8,10,17,21,32,34} are the vertices of hexagons.

Write a program which will repeatedly accept a set of points on this triangular grid, analyze it, and determine whether the points are the vertices of one of the following “acceptable” figures: triangle, parallelogram, or hexagon. In order for a figure to be acceptable, it must meet the following two conditions:

1)Each side of the figure must coincide with an edge in the grid.

and 2)All sides of the figure must be of the same length.

Input and Output

The input will consist of an unknown number of point sets. Each point set will appear on a separate line in the file. There are at most six points in a set and the points are limited to the range 1..32767.

For each point set in the input file, your program should deduce from the number of points in the set which geometric figure the set potentially represents; e.g., six points can only represent a hexagon, etc. The output must be a series of lines listing each point set followed by the results of your analysis.

Sample Input	Sample Output
1 2 3	1 2 3 are the vertices of a triangle
11 13 29 31	11 13 29 31 are not the vertices of an acceptable figure
26 11 13 24	26 11 13 24 are the vertices of a parallelogram
4 5 9 13 12 7	4 5 9 13 12 7 are the vertices of a hexagon
1 2 3 4 5	1 2 3 4 5 are not the vertices of an acceptable figure
47	47 are not the vertices of an acceptable figure

11 13 23 25 11 13 23 25 are not the vertices of an acceptable figure

Problem "C"
Concurrency Simulator

Programs executed concurrently on a uniprocessor system appear to be executed at the same time, but in reality the single CPU alternates between the programs, executing some number of instructions from each program before switching to the next. You are to simulate the concurrent execution of up to ten programs on such a system and determine the output that they will produce.

The program that is currently being executed is said to be *running*, while all programs awaiting execution are said to be *ready*. A program consists of a sequence of no more than 25 statements, one per line, followed by an end statement. The statements available are listed below.

Statement Type	Syntax
Assignment	*variable* = *constant*
Output	**print** *variable*
Begin Mutual Exclusion	**lock**
End Mutual Exclusion	**unlock**
Stop Execution	**end**

A *variable* is any single lowercase alphabetic character and a *constant* is an unsigned decimal number less than 100. There are only 26 variables in the computer system, and they are shared among the programs. Thus assignments to a variable in one program affect the value that might be printed by a different program. All variables are initially set to zero.

Each statement requires an integral number of time units to execute. The running program is permitted to continue executing instructions for a period of time called its *quantum*. When a program's time quantum expires, another ready program will be selected to run. Any instruction currently being executed when the time quantum expires will be allowed to complete.

Programs are queued first-in-first-out for execution in a *ready queue*. The initial order of the ready queue corresponds to the original order of the programs in the input file. This order can change, however, as a result of the execution of **lock** and **unlock** statements.

The **lock** and **unlock** statements are used whenever a program wishes to claim mutually exclusive access to the variables it is manipulating. These statements always occur in pairs, bracketing one or more other statements. A **lock** will always precede an **unlock**, and these statements will never be nested. Once a program successfully executes a **lock** statement, no other program may successfully execute a **lock** statement until the locking program runs and executes the corresponding **unlock** statement. Should a running program attempt to execute a **lock** while one is already in effect, this program will be placed at the end of the *blocked queue*. Programs blocked in this fashion lose any of their current time quantum remaining. When an **unlock** is executed, any program at the head of the blocked queue is moved to the head of the ready queue. The first statement this program will execute when it runs will be the **lock** statement that previously failed. Note that it is up to the programs involved to enforce the mutual exclusion protocol through correct usage of **lock** and **unlock** statements. (A renegade program with no **lock**/**unlock** pair could alter any variables it wished, despite the proper use of **lock**/**unlock** by the other programs.)

Input and Output

The first line of the input file consists of seven integers separated by spaces. These integers specify (in order): the number of programs which follow, the unit execution times for each of the five statements (in the order

given above), and the number of time units comprising the time quantum. The remainder of the input consists of the programs, which are correctly formed from statements according to the rules described above.

All program statements begin in the first column of a line. Blanks appearing in a statement should be ignored. Associated with each program is an identification number based upon its location in the input data (the first program has ID = 1, the second has ID = 2, etc.).

Your output will contain of the output generated by the print statements as they occur during the simulation. When a print statement is executed, your program should display the program ID, a colon, a space, and the value of the selected variable. Output from separate print statements should appear on separate lines. A sample input and correct output are shown below.

Sample Input

```
3 1 1 1 1 1 1
a = 4
print a
lock
b = 9
print b
unlock
print b
end
a = 3
print a
lock
b = 8
print b
unlock
print b
end
b = 5
a = 17
print a
print b
lock
b = 21
print b
unlock
print b
end
```

Sample Output

```
1: 3
2: 3
3: 17
3: 9
1: 9
1: 9
2: 8
2: 8
3: 21
3: 21
```

Problem “D”
The Domino Effect

A standard set of Double Six dominoes contains 28 pieces (called bones) each displaying two numbers from 0 (blank) to 6 using dice-like pips. The 28 bones, which are unique, consist of the following combinations of pips:

Bone #	Pips	Bone #	Pips	Bone #	Pips	Bone #	Pips
1	0 | 0	8	1 | 1	15	2 | 3	22	3 | 6
2	0 | 1	9	1 | 2	16	2 | 4	23	4 | 4
3	0 | 2	10	1 | 3	17	2 | 5	24	4 | 5
4	0 | 3	11	1 | 4	18	2 | 6	25	4 | 6
5	0 | 4	12	1 | 5	19	3 | 3	26	5 | 5
6	0 | 5	13	1 | 6	20	3 | 4	27	5 | 6
7	0 | 6	14	2 | 2	21	3 | 5	28	6 | 6

All the Double Six dominoes in a set can he laid out to display a 7 x 8 grid of pips. Each layout corresponds at least one “map” of the dominoes. A map consists of an identical 7 x 8 grid with the appropriate bone numbers substituted for the pip numbers appearing on that bone. An example of a 7 x 8 grid display of pips and a corresponding map of bone numbers is shown below.

7 x 8 grid of pips	map of bone numbers
6 6 2 6 5 2 4 1	28 28 14 7 17 17 11 11
1 3 2 0 1 0 3 4	10 10 14 7 2 2 21 23
1 3 2 4 6 6 5 4	8 4 16 25 25 13 21 23
1 0 4 3 2 1 1 2	8 4 16 15 15 13 9 9
5 1 3 6 0 4 5 5	12 12 22 22 5 5 26 26
5 5 4 0 2 6 0 3	27 24 24 3 3 18 1 19
6 0 5 3 4 2 0 3	27 6 6 20 20 18 1 19

Write a program that will analyze the pattern of pips in any 7 x 8 layout of a standard set of dominoes and produce a map showing the position of all dominoes in the set. If more than one arrangement of dominoes yield the same pattern, your program should generate a map of each possible layout.

Input and Output

The input file will contain several of problem sets. Each set consists of seven lines of eight integers from 0 through 6, representing an observed pattern of pips. Each set is corresponds to a legitimate configuration of bones (there will be at least one map possible for each problem set). There is no intervening data separating the problem sets.

Correct output consists of a problem set label (beginning with Set #1) followed by an echo printing of the problem set itself. This is followed by a map label for the set and the map(s) which correspond to the problem set. (Multiple maps can be output in any order.) After all maps for a problem set have been printed, a summary line stating the number of possible maps appears. At least three lines are skipped between the output from different problem sets while at least one line separates the labels, echo printing, and maps within the same problem set. A sample input file of two problem sets along with the correct output are shown on the reverse of this page.

Sample Input

```
5 4 3 6 5 3 4 6
0 6 0 1 2 3 1 1
3 2 6 5 0 4 2 0
5 3 6 2 3 2 0 6
4 0 4 1 0 0 4 1
5 2 2 4 4 1 6 5
5 5 3 6 1 2 3 1
4 2 5 2 6 3 5 4
5 0 4 3 1 4 1 1
1 2 3 0 2 2 2 2
1 4 0 1 3 5 6 5
4 0 6 0 3 6 6 5
4 0 1 6 4 0 3 0
6 5 3 6 2 1 5 3
```

Sample Output

```
Layout #1:

   5   4   3   6   5   3   4   6
   0   6   0   1   2   3   1   1
   3   2   6   5   0   4   2   0
   5   3   6   2   3   2   0   6
   4   0   4   1   0   0   4   1
   5   2   2   4   4   1   6   5
   5   5   3   6   1   2   3   1

Maps resulting from layout #1 are:

   6  20  20  27  27  19  25  25
   6  18   2   2   3  19   8   8
  21  18  28  17   3  16  16   7
  21   4  28  17  15  15   5   7
  24   4  11  11   1   1   5  12
  24  14  14  23  23  13  13  12
  26  26  22  22   9   9  10  10

There are 1 solution(s) for layout #1.


Layout #2:

   4   2   5   2   6   3   5   4
   5   0   4   3   1   4   1   1
   1   2   3   0   2   2   2   2
   1   4   0   1   3   5   6   5
   4   0   6   0   3   6   6   5
   4   0   1   6   4   0   3   0
   6   5   3   6   2   1   5   3

Maps resulting from layout #2 are:

  16  16  24  18  18  20  12  11
   6   6  24  10  10  20  12  11
   8  15  15   3   3  17  14  14
   8   5   5   2  19  17  28  26
  23   1  13   2  19   7  28  26
  23   1  13  25  25   7   4   4
  27  27  22  22   9   9  21  21

  16  16  24  18  18  20  12  11
   6   6  24  10  10  20  12  11
   8  15  15   3   3  17  14  14
   8   5   5   2  19  17  28  26
  23   1  13   2  19   7  28  26
  23   1  13  25  25   7  21   4
  27  27  22  22   9   9  21   4

There are 2 solution(s) for layout #2.
```

Problem "E"
Use of Hospital Facilities

County General Hospital is trying to chart its course through the troubled waters of the economy and shifting population demographics. To support the planning requirements of the hospital, you have been asked to develop a simulation program that will allow the hospital to evaluate alternative configurations of operating rooms, recovery rooms and operations guidelines. Your program will monitor the usage of operating rooms and recovery room beds during the course of one day.

County General Hospital has several operating rooms and recovery room beds. Each surgery patient is assigned to an available operating room and following surgery the patient is assigned to one of the recovery room beds. The amount of time necessary to transport a patient from an operating room to a recovery room is fixed and independent of the patient. Similarly, both the amount of time to prepare an operating room for the next patient and the amount of time to prepare a recovery room bed for a new patient are fixed.

All patients are officially scheduled for surgery at the same time, but the order in which they actually go into the operating rooms depends on the order of the patient roster. A patient entering surgery goes into the lowest numbered operating room available. For example, if rooms 2 and 4 become available simultaneously, the next patient on the roster not yet in surgery goes into room 2 and the next after that goes into room 4 at the same time. After surgery, a patient is taken to the available recovery room bed with the lowest number. If two patients emerge from surgery at the same time, the patient with the lower number will be the first assigned to a recovery room bed. (If in addition the two patients entered surgery at the same time, the one first on the roster is first assigned a bed.)

Input and Output

The input file contains data for a single simulation run. All numeric data in the input file are integers, and successive integers on the same line are separated by blanks. The first line of the file is the set of hospital configuration parameters to be used for this run. The parameters are, in order:

Number of operating rooms (maximum of 10)
Number of recovery room beds (maximum of 30)
Starting hour for 1st surgery of day (based on a 24-hour clock)
Minutes to transport patient from operating room to recovery room
Minutes to prepare operating room for next patient
Minutes to prepare recovery room bed for next patient
Number of surgery patients for the day (maximum of 100)

This initial configuration data will be followed by pairs of lines of patient data as follows:

Line 1: Last name of patient (maximum of 8 characters)
Line 2: Minutes required for surgery Minutes required in the recovery room

Patient records in the input file are ordered according to the patient roster, which determines the order in which patients are scheduled for surgery. The number of recovery room beds specified in any configuration will be sufficient to handle patients arriving from surgery (No queuing of patients for recovery room beds will be required). Computed times will not extend past 24:00.

Correct output shows which operating room and which recovery room bed is used by each patient, and the time period that the patient uses the room and bed along with a summary of the utilization of hospital facilities for that day. The output file consists of a set of two tables describing the results of the simulation run. The first table is in columnar form with appropriate column labels to show the number of each patient (in the order the patient

roster), the patient’s last name, the operating room number, the time surgery beings and ends, the recovery bed number and the time the patient enters and leaves the recovery room bed.

The second table will also be in columnar form with appropriate column labels summarizing the utilization of operating rooms and recovery room beds. This summary indicates the facility type (room or bed), the facility number, the number of minutes used and percentage of available time utilized. Available time is defined as the time in minutes from the starting time for 1st surgery of day to the ending time of the last patient in a recovery room bed. A sample input file and corresponding correct output are shown below.

Sample input

```
5 12 07 5 15 10 16
Jones
28 140
Smith
120 200
Thompson
23 75
Albright
19 82
Poucher
133 209
Comer
74 101
Perry
93 188
Page 13Bush       1    9:49   10:15     10   10:20   12:21
111 223
Roggio
69 122
Brigham
42 79
Nute Facility Utilization
22 71
Young
38 140
Bush Room  2     248   44.60
26 121
Cates
120 248
Johnson
86 181
White
92 140
```

Sample output

```
                    Patient          Operating Room          Recovery Room
                    #  Name     Room#  Begin   End      Bed#  Begin    End
                    ------------------------------------------------------
                    1  Jones      1    7:00    7:28      3    7:33    9:53
                    2  Smith      2    7:00    9:00      1    9:05   12:25
                    3  Thompson   3    7:00    7:23      2    7:28    8:43
                    4  Albright   4    7:00    7:19      1    7:24    8:46
                    5  Poucher    5    7:00    9:13      5    9:18   12:47
                    6  Comer      4    7:34    8:48      4    8:53   10:34
                    7  Perry      3    7:38    9:11      2    9:16   12:24
                    8  Page       1    7:43    9:34      6    9:39   13:22
                    9  Roggio     4    9:03   10:12      9   10:17   12:19
                   10  Brigham    2    9:15    9:57      8   10:02   11:21
                   11  Nute       3    9:26    9:48      7    9:53   11:04
                   12  Young      5    9:28   10:06      3   10:11   12:31
                   14  Cates      3   10:03   12:03      8   12:08   16:16
                   15  Johnson    2   10:12   11:38      4   11:43   14:44
                   16  White      5   10:21   11:53      7   11:58   14:18

                    Type  # Minutes  % Used
                    ------------------------
                    Room  1     165   29.68
                    Room  3     258   46.40
                    Room  4     162   29.14
                    Room  5     263   47.30
                    Bed   1     282   50.72
                    Bed   2     263   47.30
                    Bed   3     280   50.36
                    Bed   4     282   50.72
                    Bed   5     209   37.59
                    Bed   6     223   40.11
                    Bed   7     211   37.95
                    Bed   8     327   58.81
                    Bed   9     122   21.94
                    Bed  10     121   21.76
                    Bed  11       0    0.00
                    Bed  12       0    0.00
```

Problem “F”
Message Decoding

Some message encoding schemes require that an encoded message be sent in two parts. The first part, called the header, contains the characters of the message. The second part contains a pattern that represents the message. You must write a program that can decode messages under such a scheme.

The heart of the encoding scheme for your program is a sequence of “key” strings of 0’s and 1’s as follows:

0,00,01,10,000,001,010,011,100,101,110,0000,0001,. . .,1011,1110,00000, . . .

The first key in the sequence is of length 1, the next 3 are of length 2, the next 7 of length 3, the next 15 of length 4, etc. If two adjacent keys have the same length, the second can be obtained from the first by adding 1 (base 2). Notice that there are no keys in the sequence that consist only of 1’s.

The keys are mapped to the characters in the header in order. That is, the first key (0) is mapped to the first character in the header, the second key (00) to the second character in the header, the kth key is mapped to the kth character in the header. For example, suppose the header is:

AB#TANCnrtXc

Then 0 is mapped to A, 00 to B, 01 to #, 10 to T, 000 to A, ..., 110 to X, and 0000 to c.

The encoded message contains only 0’s and 1’s and possibly carriage returns, which are to be ignored. The message is divided into segments. The first 3 digits of a segment give the binary representation of the length of the keys in the segment. For example, if the first 3 digits are 010, then the remainder of the segment consists of keys of length 2 (00, 01, or 10). The end of the segment is a string of 1’s which is the same length as the length of the keys in the segment. So a segment of keys of length 2 is terminated by 11. The entire encoded message is terminated by 000 (which would signify a segment in which the keys have length 0). The message is decoded by translating the keys in the segments one-at-a-time into the header characters to which they have been mapped.

Input and Output

The input file contains several data sets. Each data set consists of a header, which is on a single line by itself, and a message, which may extend over several lines. The length of the header is limited only by the fact that key strings have a maximum length of 7 (111 in binary). If there are multiple copies of a character in a header, then several keys will map to that character. The encoded message contains only 0’s and 1’s, and it is a legitimate encoding according to the described scheme. That is, the message segments begin with the 3-digit length sequence and end with the appropriate sequence of 1’s. The keys in any given segment are all of the same length, and they all correspond to characters in the header. The message is terminated by 000. Carriage returns may appear anywhere within the message part. They are *not* to be considered as part of the message.

For each data set, your program must write its decoded message on a separate line. There should not be blank lines between messages. Sample input and corrresponding correct output are shown below.

Sample input

```
TNM AEIOU
0010101100011
1010001001110110011
11000
$#**\
0100000101101100011100001000
```

Sample output

```
TAN ME
##*\$
```

Problem “G”
Code Generation

Your employer needs a backend for a translator for a very SIC machine (Simplified Instructional Computer, apologies to Leland Beck). Input to the translator will be arithmetic expressions in postfix form and the output will be assembly language code.

The target machine has a single register and the following instructions, where the operand is either an identifier or a storage location.

L	load the operand into the register
A	add the operand to the contents of the register
S	subtract the operand from the contents of the register
M	multiply the contents of the register by the operand
D	divide the contents of the register by the operand
N	negate the contents of the register
ST	store the contents of the register in the operand location

An arithmetic operation replaces the contents of the register with the expression result. Temporary storage locations are allocated by the assembler for an operand of the form “$n” where n is a single digit.

Input and Output

The input file consists of several legitimate postfix expressions, each on a separate line. Expression operands are single letters and operators are the normal arithmetic operators (+, -, *, /) and unary negation (@). Output must be assembly language code that meets the following requirements:

1. One instruction per line with the instruction mnemonic separated from the operand (if any) by one blank.
2. One blank line must separate the assembly code for successive expressions.
3. The original order of the operands must be preserved in the assembly code.
4. Assembly code must be generated for each operator as soon as it is encountered.
5. As few temporaries as possible should be used (given the above restrictions).
6. For each operator in the expression, the minimum number of instructions must be generated (given the above restrictions).

A sample input file and corrresponding correct output are on the reverse of this paper.

Sample input

```
AB+CD+EF++GH+++
AB+CD+-
```

Sample output

```
L A
A B
ST $1
L C
A D
ST $2
L E
A F
A $2
ST $2
L G
A H
A $2
A $1

L A
A B
ST $1
L C
A D
N
A $1
```