UAE

National Programming Contest
29-30 March 2006

Etisalat University College

Problem Set Includes

- *No electronically stored information is allowed*
- *No disturbance in the lab*
- *Raise up your hand in case you need anything.*

Problem A

```
Source file: lucky{.c|.cpp|.java}
Input file: lucky.in
Output file: lucky.out
```

Lucky Names through Prime Numbers

An integer greater than one is called a prime number if its only positive divisors (factors) are one and itself. For example, the prime divisors of 8 are 2 and 4; and the first six primes are 2, 3, 5, 7, 11 and 13.

In this problem we introduce a prime character, which is a character with the corresponding ASCI code translates to a prime number. For example, the ASCI code of the character 'A' is 65, thus, 'A' is not a prime character. The ASCI code of the character 'C' is 67, therefore, 'C' is a prime character.

In this problem your task is to define the percentage of luck for each name provided to you. Names are constituted by characters with the ASCI code from 65 ('A') to 122 ('z'), including underscores.

We compute the percentage of lucky names by calculating the average weight of the characters constituting each name, where each character is given either the weight 100 if it is a prime or the weight 0 otherwise.

Given a name N of length l, the percentage of luck for N is computed as follow:

$W(N) = \sum w_i / l$ $\quad\quad$ w_i: the weight of character i ($i = 1..l$).

Example1:

$W(\text{"SISCO"}) = 100/5 + 100/5 + 100/5+ 100/5+ 100/5$

$= 100$ Thus, the word ***SISCO*** is ***100%*** Lucky

Example2:

$W(\text{"cake"}) = 0/4 + 100/4 + 100/4 + 100/4$

$= 75$ Thus, the word ***cake*** is ***75%*** Lucky

Input

The input file contains a number of lines. Each line contains a single name for which, you are asked to compute its percentage of luck. Input is terminated by end of file.

Output

For each line of input produce a line of output. The output should outline the name followed by its percentage of lack.

Sample Input

```
SISCO
cake
Summer
Virus
```

Sample Output

```
SISCO: 100% Lucky
cake: 75% Lucky
Summer: 66% Lucky
Virus: 0% Lucky
```

Problem B

```
Source file: depth{.c|.cpp|.java}
Input file: depth.in
Output file: depth.out
```

Depth Traversal

The manager of a watch manufacturing company hired Mr. Jamal to launch a campaign in order to publicize the new model of watches that the company is produced. Mr. Jamal decided to organize an exhibition in all the cities in the country to demonstrate the products to potential customers. Mr. Jamal is in need for a plan that allows him to visit the entire country, one city at a time and only once. Your task is to help Mr. Jamal to produce an exhaustive traversal plan. Assume the following requirements:

- Cities are numbered incresingly from 1 (i.e. 1, 2, 3…..).
- Mr Jamal's trip must always start from the city with the lowest number and in the case where you can reach more than one city from a particular city, Mr. Jamal would like to visit the cities starting from the city with the lowest number and so on.
- No city shall be visited more than one time.

Write a program that produces the list of nodes that Mr. Jamal needs to visit to cover a whole country.

Input

The first line in the input file is an integer `N` representing the number of cities in a country. The second line is also an integer `M` describing the number of routes between all the cities in the country. Each of the next `M` input lines contains two integers (representing cities) separated by a space to denote a route between the two cities. Note that there is no specific order for the `M` lines or for the routes in the `M` lines.

Output

The output consists of the list of cities to be traversed according to the above requirements.

Sample Input

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

Sample Output

```
1 5 2 6 3 4 7
```

Problem C

```
Source file: factorials{.c|.cpp|.java}
Input file: factorials.in
Output file: factorials.out
```

Factorials

Factorials are used in several areas of Science. Factorials are denoted as a number followed by an exclamation point. For instance, factorial of an integer *n*, written *n*!, is the product of all integers from 1 to *n* inclusive. The factorial quickly becomes very large; 15! is too large to be stored as an integer on most computer, and 35! is too large for a floating-point variable. Your task is to write a program to find the rightmost nonzero digit of *n*! For example: 5! = 1 * 2 * 3 * 4 * 5 * 6 = 120, therefore the rightmost non-zero digit of 5! is 2. Also, 7 = 1 * 2 * 3 * 4 * 5 * 6 * 7 = 5040, so the rightmost nonzero of factorial of 7! is 4. *Note* that 0! = 1! = 1.

Input

An integer *n*, between 1 and 10,000,000 inclusive.

The input test data is stored in file "*factorials.in*". Each test case is on a separate line. The end of the input data is indicated by "-1". Note that 0! = 1! = 1.

Output

The output should be saved in a file "*factorials.out*" and the result should be the rightmost non-zero digit of *n*!

Sample Input:

```
3
5
9
7
-1
```

Sample Output:

```
6
2
8
4
```

Problem D

```
Source file:points.{.c|.cpp|.java}
Input File: points.in
Output File: points.out
```

Lines and Points

Given a set of points in two-dimensional (2D) coordinate space, each with integer *x* and *y* coordinates, find the largest number *K* of points lying exactly on the same line.

Input

The input file "points. in" contains an unknown number of data points sets. Every set is preceded by an integer *n* specifying the number of points in that set. If *n* is 0, then there are no more sets.

Each line in the file stores the *x* coordinate and the *y* coordinate (in this order) separated by a space.

Output

The output file "points.out" contains the number *K* for each set in a separate line.

Sample Input

```
6
2 6
4 10
6 14
8 20
10 22
9 30
7
2 9
4 11
15 22
20 32
6 15
8 15
12 19
0
```

Sample Output

```
4
5
```

Problem E

```
Source file:farmer.{.c|.cpp|.java}
Input File: farmer.in
Output File: farmer.out
```

Farmer and Corn Fields

A farmer harvests crops from different farmers' fields that he gets a percentage of the harvested corn as his payment. However every time he goes to a field, he has to measure the area of the field so that he can estimate if harvesting this field is profitable or not. Obviously this is not a straight forward task, as their shapes are usually irregular that any simple formula can't be applied; and the fields are filled with corn that farmer can't get in the fields. Therefore, the only method that he has to use is to walk around each field's periphery to record the distance and the direction of each move. This gives him a non-self intersecting polygon (none of the lines intersect with each other) representing the periphery of the fields. This smart farmer knows that if he gets an imaginary straight line that is not intersecting the polygon, he can draw two perpendicular lines to the imaginary line from staring and ending points of each move and parallel to each other (Figure 1). By this method he finds trapezoids with signed area (some positive if the move is away from the starting point, and some negative if the move is towards the starting point) (Figure 2). When he adds the area of these trapezoids, he can calculate the area of the corn fields. The farmer asks you to write a program that he will use for calculating the area of corn fields.

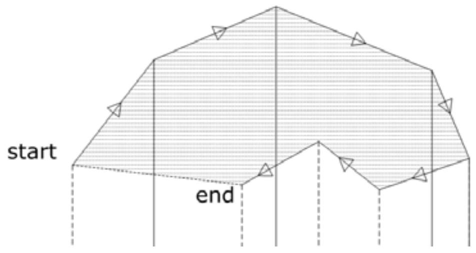

Figure 1. Polygon and parallel lines

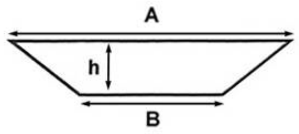

Figure 2. Trapezoid

`Area = [(A + B) ÷ 2] x h`

Input

The input starts with a number `M` describing the number of moves. Each line following `M` contains decimal pair *l* and *d* representing a move: *l* shows the length of the move and *d* shows the direction in degree angle counter clockwise. End of file is marked with 0 (zero).

Output

Display the area of each corn field in the input file.

Sample Input

```
3
50 45
100 0
50 225
5
50 45
50 0
50 315
50 225
50 180
5
10 90
10 0
10 90
10 0
28.284 315
0
```

Sample Output

```
Area for field 1 is 3535.5 unit-square
Area for field 2 is 6035.5 unit-square
Area for field 3 is 500.0 unit-square
```

Problem F

```
Source file: roman{.c|.cpp|.java}
Input file: roman.in
Output file: roman.out
```

Roman Numbers

A Roman numeral represents an integer using letters. For examples XVII represent 17. By contrast, ordinary numbers such as 17 or 1920 are called Arabic numerals. The following table shows the Arabic equivalent of all the single-letter Roman numerals:

M	1000	X	10
D	500	V	5
C	100	I	1
L	50		

When letters are joined together to form a string, the values of the letters are just added up, with the following exception. When a letter of smaller value is followed by a letter of larger value, the smaller value is subtracted from the larger value. For example, IV represents 5 - 1, or 4. And MCMXCV is interpreted as M + CM + XC + V, or 1000 + (1000 - 100) + (100 - 10) + 5, which is 1995. In standard Roman numerals, no more than three consecutive copies of the same letter are used. Following these rules, every number between 1 and 3999 can be represented as a Roman numeral made up of the following one- and two-letter combinations:

M	1000	X	10
CM	900	IX	9
D	500	V	5
CD	400	IV	4
C	100	I	1
XC	90		
L	50		
XL	40		

Write a program that reads Roman numerals and prints the corresponding Arabic numeral. The program should end when the user inputs an empty line.

Input

The data input file “roman.in” consists of Roman numerals where each numeral is on a single line.

Output

Corresponding Arabic numerals should be stored in a file “roman.out”.

Sample Input

```
MXCLVI
LCIIXCL
MXVLCII
```

Sample Outpu

```
1146
210
1157
```

Problem G

```
Source file: subsets{.c|.cpp|.java}
Input file: subsets.in
Output file: subsets.out
```

Sum of Subsets

The jewelry store TREASURE contains valuable items each of which has a label indicating its weight. The store manager is organizing a special promotion for all customers. Each customer is given a knapsack that can carry a fix predefined weight. Every customer who is able to fill his knapsack with the exact weight gets a 20% discount. You were asked by a friend to write a software program to help him choose the items that will fit exactly in any particular knapsack. Your friend will run the program on his portable computer and once inside the store, he will enter the weights of all the items and a desired target weight for a knapsack and your program will print out all possible solutions.

Input

The first line of the input file to this problem contains the number N (N <= 1000) of items available in the store on a separate line followed by the set of N positive integers representing the weights of the N items. The integers are organized in consecutive lines each of which contains 10 numbers separated by a tabulation character. The last line may contain less than 10 numbers depending of the value of N. After that, the input file contains the number M (M <= 1000000) of samples to be tested followed by M lines each of which contains a target weight.

Output

The output for each sample starts with the word "SAMPLE" followed by a single space and the sequence number of the sample (1, 3 …M) on a separate line. Each possible solution will follow on a separate line which contains the numbers (weights) separated by a single space. Your program should output all possible solutions or a message "No Solution" if none is possible. A solution is a set of weights that sums to the target weight.

Example: The example below shows fourteen (14) items with weights: 13, 17, 99, 71, 7, 1, 66, 73, 53, 529, 4, 71, 70 and 19 and three samples of target weights respectively 3, 25 and 35. The first sample (weight =3) has no solutions, the second sample (weight =25) has two solutions: 1 4 7 13 and 1 7 17 because 1+4+7+13=25 and 1+7+17=25. The last sample (weight =35) has one single solution 1 4 13 17 as 1+4+13+17=35.

Sample Input

```
14
13    17    99    71    7     1     66    73    53    529
4     71    70    19
3
3
25
35
```

Sample Output

```
SAMPLE 1
No Solution
SAMPLE 2
1 4 7 13
1 7 17
SAMPLE 3
1 4 13 17
```

Problem H

```
Source file: piratemap{.c|.cpp|.java}
Input file: piratemap.in
Output file: piratemap.out
```

Pirate and Sorted Treasures

An old pirate hides his gold coins by distributing them among different cities. As his final and encoded map, the pirate keeps a flat list of visited cities. In order to convert the list to a geographic map (G-map) he invented the following golden rules.

1. A city can have direct road to maximum three cities: a city that the pirate comes from, a city somewhere east or west of this city. Three cities that the pirate visits can have only one path. For example, if the city *Alg* has a direct road to the cities *Balg* and *Calg*; then *Balg* and *Calg* are not directly connected.
2. Each city name in the list is stored one-by-one and the pirate decides where to locate it on the G-map by the following logic:
    a. If it is the first city read, it is the harbor city.
    b. Each city read in sequence from the list is placed on the G-map by comparing to city names read before and located on the G-map. He adds up each letter's numeric value in the city name (such as A is 1 and Z is 26 for small or capital letters) and divides the result with the number of letters. If the value is smaller than or equal to the city name value being compared to, it is mapped on somewhere East of that city, else somewhere on West.
    c. Last located city on a direction (East or West) has a direct connection with the city being placed on the G-map.
3. The amount of gold hidden in each city is twice more than the gold in the city before. For example, if the pirate moves from the *harbor city* to *Alg*, and from *Alg* to *Balg*, he stores twice as much gold coins in *Balg* than *Alg*; and twice as much gold in *Alg* than the *harbor city*.

By using these golden mapping rules, find out how many gold coins the old pirate hides in each city and how much the total treasure is worth.

Input File

The input starts with a number M describing the number of cities. It is followed by another numeric value showing how much gold coins stored in the harbor city. Following each line refers to a city name.

Output File

Display each city name with the hidden gold coins sorted from lowest to highest. After the city list, display the total gold coins that this old pirate hid.

Sample Input

```
7
4
Calg
Balg
Dalg
Bolg
Alg
Egle
Tangle
```

Sample Output

```
Calg: 4
Balg: 8
Dalg: 8
Bolg: 16
Alg:  32
Tangle: 32
Egle: 64

Total hidden gold coins: 164
```

Problem I

```
Source file: words{.c|.cpp|.java}
Input file: words.in
Output file: words.out
```

Word Frequencies

Many information retrieval applications make use of document word frequencies in order to retrieve the most relevant documents from a database of texts according to queries from users. The calculation of frequencies of words in text documents is the first step to prepare a database for information retrieval.

Write a program that reads a text from an input file and counts the number of occurrences of each word. A word is defined by any sequence of characters delimited by a space, a new line or a punctuation mark. You should write the words to an output file along with their frequencies first in alphabetical order and in second arranged by the by their frequencies in descending order. In the later case, words with similar frequency should be arranged in alphabetical order.

Input

The input consists of a text file containing ASCII text.

Output

The output consists of the list of words in the input file and their frequencies arranged first in alphabetical order of the words and secondly by frequencies. The first line is the sentence "Words in alphabetical order:", then a set of output lines ordered in alphabetical order. Each line contains a word from the text followed by a space and then its frequency. Then the sentence "Words in order of frequency:" followed by a set of lines ordered according to the frequencies of the words as described above. Each line contains a word followed by a space and then the word frequency.

Sample Input

```
Constraint satisfaction problems are set of problems with a complete
solution. The problems consist of a set of variables, subject to the
particular constraints of the problems.
```

Sample Output

```
Words in alphabetical order:
a 2
are 1
complete 1
consist 1
constraint 1
constraints 1
of 4
particular 1
problems 4
satisfaction 1
set 2
solution 1
subject 1
the 3
to 1
variables 1
with 1
Words in order of frequency:
of 4
problems 4
the 3
a 2
set 2
are 1
complete 1
consist 1
constraint 1
constraints 1
particular 1
satisfaction 1
solution 1
subject 1
to 1
variables 1
with 1
```

Problem J

```
Source file: password{.c|.cpp|.java}
Input file: password.in
Output file: password.out
```

Check Password

A password is a sequence of characters that can be used for authentication purposes. Passwords are widely used in computer systems and networks; and they are often used to authenticate the identity of an automated data processing (ADP) system user.

A valid password is a personal password that will authenticate the identity of an individual when presented to a *password system*.

In this problem you are asked to check the validity of passwords through the development of some routines that will play the role of a password system. Basic factors shall be considered in the design, implementation, and use of a password system used to authenticate the identity of a person. In this problem we only focus on two main factors, comprising 4 distinct rules:

- **Length Range Factor**: Length Range is the set of acceptable lengths of passwords, expressed as a minimum length through a maximum length.
  - **Rule 1**: acceptable number of characters in a valid password must be more than 6 and less than 15.
- **Composition Factor**: Composition is the set of acceptable characters which may be used in a valid password.
  - **Rule 2**: valid password should not be only alphabetic characters
  - **Rule 3**: valid password should not be only numeric characters
  - **Rule 4**: valid password should not be a series of alphabetic characters ending with '99'

Passwords are alphanumeric characters with the ASCI code in the following ranges:

- 48 – 57 for numeric values (0, 1, …, 9)
- 65 - 90 : upper case alphabetic characters (A, B, …, Z)
- 97 – 122: lower case alphabetic characters (a, b, …, z)

Input

The input file contains a number of lines. Each line contains a single word (password to be checked). Input is terminated by end of file.

Output

For each line of input produce a line of output. The output should outlines if a password is valid or not, in case of invalid password the output should identify the violated rules.

Sample Input

```
NPC2006
CGI
a199
99
CGI99
```

Sample Output

```
valid Password
not a valid Password (rule(s): 1,2 violated)
not a valid Password (rule(s): 1 violated)
not a valid Password (rule(s): 1,3 violated)
not a valid Password (rule(s): 1,4 violated)
```