10th Gulf Programming Contest

November 1 – 2, 2022

SQU, Oman

Duration: 10 am - 3 pm

Sponsored by

Problem Set

Problem #	***Problem Name***	***Balloon Color***
A	Dining Table	Light Blue
B	Square Counter	Red
C	Binary Play	Black
D	Virus Outbreak	White
E	Construction Frenzy	Orange
F	Date Pyramid	Dark Blue
G	Transaction frequency	Golden
H	Areej Makeup Show	Green
I	The League	Yellow
J	The Company Trip	Pink
K	Your Weight On Other Worlds	Purple

A. Dining Table

Program:	`table.(cpp|java|py)`
Input:	`table.in`

Description

Sara is a baby girl who likes to sit at the dining table. She just learned how to climb over small objects!! In order to reach the dining table, Sara learned to stack small side tables and use them as steps in order to reach the dining table, as shown in Figure 1. To stop her from doing so, her parents removed the smallest table, and kept it in the store. This worked, since she could not climb or step over to the middle one.

However, Sara has now grown and so you must solve this problem again. Given the height of the dining table in centimeters (H), the number of side tables (N), the height of every side table in centimeters L, and the maximum height Sara can climb S; your task is to find which table the parents should remove in order to stop her, where they prefer to remove the shortest table if more than one can do the job. Note that parents are willing to remove only one table.

Example

H=100 N=3, L = {40, 55, 70}, S = 50, the solution is: 40.
H=100 N=3, L = {30, 50, 70}, S = 50, the solution is: Not Possible.
H=100 N=3, L = {20, 40, 60}, S = 50, the solution is: 60.

Input

The first line of input starts with an integer **k** denoting the number of test cases.

For every test case, the first line contains 3 integers, **H, S, N**, where **H** is the dining table height, S is the step, and **N** is the number of side tables.

Nest line contains N integers representing the heights of side tables: **L_1, L_2, .. L_N**. where

$10 < H < 1000000$
$0 < S < H$
$0 < N < 1000$
$0 < L_i < H$,
$L_i \neq L_j$ for every $i \neq j$

and for every given case, the child can climb the side tables to reach the dining table.

Output

For each input case output one line starting with the case number (starting from one), a period and the size of the table that parents should remove, or “Not possible”, if removing one table will not solve the problem.

Sample Input / Output for sample input

table.in

```
3
100 50 3
40 55 70
100 50 3
40 20 60
100 50 4
45 30 60 70
```

```
1. 40
2. 60
3. Not possible
```

B. Square Counter

Program:	square.(cpp|java|py)
Input:	square.in

Description

A board game consists of a square array of dots, with two alternating players. Each player chooses to connect any two adjacent dots on their turn, vertically or horizontally. Every time a player makes a square, he/she gets points as per the size of the squares formed. Note that a single move may result in multiple squares. Given the state of the board, you are asked to count the number of squares, grouped by their sizes (A square's size is represented by the length of its side).

Example

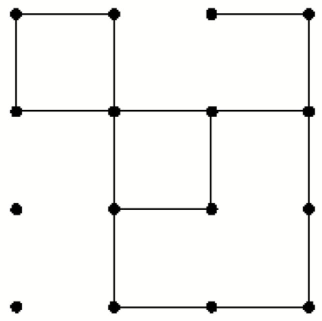


Figure 1: represented by case 1 in the sample input

For the board shown in figure one the output should be

size 1 (side length = 1): 2 squares

size 4 (side length = 2): 1 square

Input

The input consists of a number of test cases, with each case starting with an integer N ($1 < N \leq 50$) representing the length and width and of board. The input series is terminated with N = 0. N is followed by a matrix of zeros and ones, totaling N x N - 1 numbers (N rows each with N-1 columns). This matrix represents the horizontal lines between dots, 1 for a line, and 0 for no line. A second N-1 x N matrix of zeros and ones follows (N-1 rows each with N columns), that represents the vertical lines between dots, 1 for a line, and 0 for no line.

Output

For each test case, output should be of the following format:

Case k
length$_a$: s$_a$
length$_b$: s$_b$
…

Where **k** is the test case number, followed by all found square sizes in ascending order based on length. Output the length of the completed square, followed by a colon "**:**", a single space then the number of squares found. Test cases are separated by a new line. If there are no squares found, just output case number followed the new line.

Sample Input / Output

square.in

```
4
1 0 1
1 1 1
0 1 0
0 1 1
1 1 0 1
0 1 1 1
0 1 0 1
4
1 1 1
1 1 1
1 1 1
1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
3
1 1
1 1
1 1
1 1 1
1 1 1
0
```

```
Case 1
1: 2
2: 1

Case 2
1: 9
2: 4
3: 1

Case 3
1: 4
2: 1
```

C. Binary Play

Program:	`biplay.(cpp|java|py)`
Input:	`biplay.in`

Description

Given a 24-bit unsigned integer **d** in its decimal form, your task is to manipulate it – using a set of defined operators – into another 24-bit unsigned integer **r**.

To manipulate it, you can perform one of these operators:

RESET: turns all 24 bits to **0**

SET: turns all 24 bits to **1**

FLIP(i): flips a specific bit at index ***i*** (**0 <= *i* <= 23**) where 0 is the right-most bit

Example

Given d = 98 and r = 38

```
98 in binary is: 1100010
38 in binary is: 0100110
```

(14 insignificant left-side zeros are not shown)

r can be reached in many ways, but the shortest way uses two operations as follows:

```
1100010 → FLIP(6) → 0100010 → FLIP(2) → 0100110
```

Input

Your program will be tested on multiple test cases. Each test case is specified on a single line. Every input line consists of two integers **d** and **r** separated by one or more space(s) where $0 \le d, r < 2^{24}$.

The last input line has two zeros (which is not part of the test cases.)

Output

For each test case, print the smallest number of operations required to get **r** starting with **d.** Use the following format:

k.␣ans

where **k** is the sequence number of the test case (staring at 1,**)** followed by a full-stop (**.**), a single space (**␣**), and then the answer (**ans**).

Sample Input / Output

biplay.in

```
98 38
5 1048575
0 0
```

OUTPUT

```
1. 2
2. 5
```

D. Virus Outbreak

Program:	`virus.(cpp|java)`
Input:	`virus.in`

Description

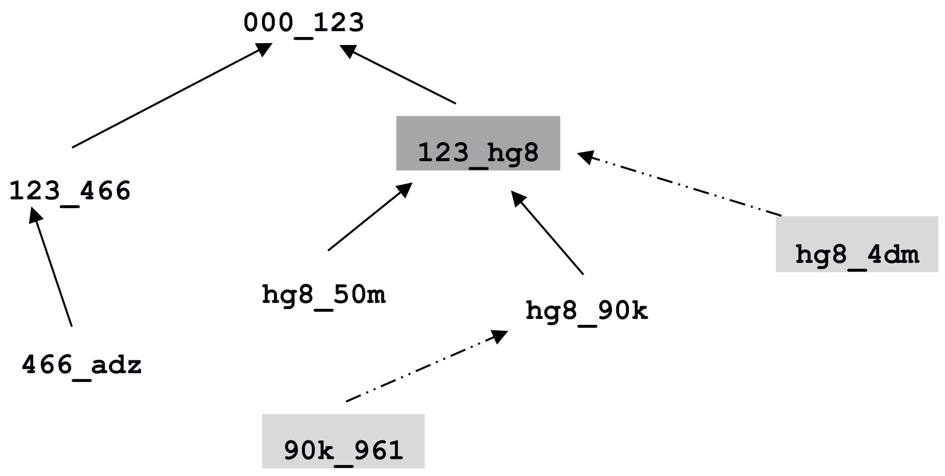
It's 2305 and a new form of virus outbreak takes place. The speed of which new strains are developing becomes alarming. Knowing that every new virus evolves from a prior existing strain, scientists soon discover that finding the *nearest common ancestor* (NCA) with a recent strain speeds up the process of containing and eliminating a new one.

The scientists have an extensive database of virus strains in their DNA codes. They need you to write a program that will determine the nearest common ancestor for any given two DNAs.

A virus DNA is coded as a pair of identifiers separated by an underscore character. Each identifier is 3 characters long. For example, **`000_abc`** is a description of a virus. When a virus evolves from another, it takes its second part of the DNA description as its first. For example, **`abc_x1z`** evolved from **`000_abc`**. The second (right) part of the DNA of any virus is always unique and is never repeated. If a virus is *original* (ie. It cannot be mapped into an existing strain,) **`000`** (three **zeros**) is chosen as the first (left) identifier. In the above example, **`000_abc`** is considered an original virus which cannot be mapped to any in the database.

Example

The following figure illustrates the genealogy of the first test case in the sample I/O in which the NCA of **`90k_961`** and **`hg8_4dm`** is **`123_hg8`**.



Input

The input file will contain several test cases. Each case has a database of viruses, followed by several queries in the following format:

The first line per case contains two integers, **`d > 0`** and **`q ≥ 0`**, where **`d`** is the number of DNAs in the database, and **q** is the number of queries. The second line of each case has **d** DNA codes. The second line is followed by **q** lines, each representing a separate query. Each query has 2 DNA codes separated by one or more spaces.

Each identifier in the DNA is made of 3 characters where each character is a decimal digit, an uppercase letter, or a lowercase letter. Identifiers are case sensitive.

The viruses in the databases are not listed in any particular order. All viruses in the queries are guaranteed to be in the database.

The last line of the input has a dummy test case with two zeros.

Output

For every query, output the answer in the following format:

`k.q.␣ans`

where **`k`** is the test case number (starting at 1) and **`q`** is the query number in that case (starting at 1 for each case). **`ans`** is the DNA code of the nearest common ancestor per that query, "**`.`**" is a full-stop character and "␣" is a single space character.

If there's no common ancestry between two DNAs, print "**`000_000`**" (an underscore in the middle of six **zeros**, without the double quotes) as the output for that query.

Sample Input:

virus.in

```
8 2
000_123 123_466 123_hg8 466_adz hg8_50m hg8_90k hg8_4dm
90k_961
90k_961 hg8_4dm
466_adz 90k_961
2 1
000_123 000_adz
000_123 000_adz
0 0
```

Sample Output

```
1.1. 123_hg8
1.2. 000_123
2.1. 000_000
```

E. Construction Frenzy

Program:	`crane.(cpp|java|py)`
Input:	`crane.in`

Description

Business is booming in a sprawling city in the Middle-East and construction companies cannot keep up with the demand for office and living space. The city's authorities have commissioned a number of cranes to help speedup the construction process, but the problem is placing them in appropriate places for serving as many construction sites as possible.

A crane can serve the construction of a building as long as it is strictly lower in height than the crane. Otherwise, it cannot be used.

The construction plans involve a rectangular area that is X*Y blocks in size, with all blocks being of equal size. The top view of such a block is shown below:

	X					
Y	0	1	2	3	4	5
0	15	12	0	40	0	35
1	X	X	0	X	X	X
2	0	X	0	X	X	0
3	0	X	X	10	30	X
4	X	0	70	0	0	X

The blocks marked with 0 are free and can hold a crane, the ones marked with X have existing buildings, and the rest are destined to have a building erected with as many floors as the number in the corresponding block.

A crane can potentially serve only the eight blocks adjacent to the block of its placement, and only if they have a number of floors which is less or equal to the crane's height.

Your goal is to find the crane placement that will allow the maximum total number of floors (i.e. the sum of the floors of the buildings it can support) to be constructed. In the example shown above, if only one crane of 50-stories height was available, then the optimum placement would be position (4,0), allowing for 75 floor to be build.

If a building can be served by two or more cranes, its contribution to the sum of floors is singular.

Input

The input file starts with a number N<100 declaring the number of test cases. Each test case starts with a line containing three integers, X, Y and C, where X<=30 and Y<=30 are the dimensions of the construction area and C<=10 is the number of cranes. The following line contains C positive integers separated by white space, that represent the height of the cranes. The test case is completed by Y lines, each containing X tokens, each being either the letter 'X' or a non-negative integer representing the number of floors to be built. The available spots for cranes do not exceed 2*C. The maximum height of a building is 300 floors.

Output

For each test case you should output in a single line the test case number (starting from 1) followed by a dot and a space, and the total number of floors that can be constructed. E.g.

```
1. 75
```

Sample Input / Output

crane.in

```
1
6 5 1
50
15 12 0 40 0 35
X X 0 X X X
0 X 0 X X 0
0 X X 10 30 X
X 0 70 0 0 X
```

OUTPUT

```
1. 75
```

F. Date Pyramid

Program:	`pyramid.(cpp|java|py)`
Input	`pyramid.in`

Description

Date production has been exceptionally good this year in the Middle East. A farmer is facing the problem of how to stack and store his produce until it is time to move it to the market. The best way it to stack them as a pyramid of 3, 4 or 6 sides as shown below:

Three-sided pyramid with 4 levels:

```
     0           0           0          0
    00          00          00
   000         000
  0000
```

Four-sided pyramid with 4 levels:

```
     0           0           0          0
    00          00          00
   000         000           0
  0000          00
   000           0
    00
     0
```

Six-sided pyramid with 4 levels:

```
   0000        000          00          0
  00000       0000         000
 000000      00000          00
0000000       0000
 000000        000
  00000
   0000
```

The problem is that the pyramid has to be as complete as possible, i.e. all the levels have to exist, otherwise the dates will spoil. An incomplete pyramid has a fitness score which is equal to the number of dates needed to complete it. A complete pyramid has a fitness score of 0.

Your goal is to find the pyramid with the best (minimum) fitness score.

Input

The input file starts with a number N<10000 declaring the number of test cases. Each test case is given as a separate line, containing the number of dates as a 32-bit positive integer.

Output

For each input you should output the result in the following format:

```
n. s f
```

where `n` is the case number (starting from 1), `s` is the size of the pyramid chosen and `f` is the fitness score. If multiple pyramids with the same score exist, you should pick the one with the most sides.

Sample Input / Output

pyramid.in

```
3
1
10
100
```

OUTPUT

```
1. 6 1
2. 3 3
3. 3 8
```

G. Transaction frequency

Program:	`transaction.(cpp|java|py)`
Input:	`transaction.in`

Description

You are given a set of transactions done in a supermarket, where each transaction consists of a set of items bought by one customer. You are given the task to find in how many item-sets of a certain length *k* appear in at least *S* (also known as the minimum support) of the transactions. An item-set is a set of items bought together.

Example

Consider the following transaction sequence in Table 1, where each row represents a transaction, and each letter corresponds to an item bought in that transaction. For example, in the first transaction (Id = 1), three different items were bought: item A, item B, and item C.

Table 1: The transactions

Transaction Id	**Items bought**
1	ABC
2	AC
3	ABD
4	BEF

Assuming minimum support *S* = 2, the question is, *how many item-sets of length k appear in at least S transactions?*

If *k*= 1, then the candidate item-sets of length *k* are all possible 1 item-sets, i.e., {A}, {B}, {C}, {D}, {E}, and {F}. The following table 2 shows the frequency of each item-set in the transactions, i.e., in how many transactions the item-set appears.

Table 2: item-sets frequencies for *k*=1

Item-set	**Frequency**	**Explanation**
{A}	3	Appears in transaction# 1, 2 and 3
{B}	3	Appears in transaction# 1, 3, and 4
{C}	2	Appears in transaction# 1 and 2
{D}	1	Appears in transaction# 3
{E}	1	Appears in transaction# 4
{F}	1	Appears in transaction #4

Therefore, there are three 1-item-sets, namely, {A}, {B}, and {C} appear in at least 2 transactions. So, the answer to the above question is 3.

If *S* = 3, then the answer to the question would be 2 (because only {A} and {B} have frequency 3)

Now, if $k = 2$, and $S = 2$, then the possible candidate k-item-sets are {AB}, {AC}, {AD}, {AE}, {AF}, {BC}, {BD}, {BE}, {BF}, {CD}, {CE}, {CF}, {DE}, and {DF}. If we count their frequencies, we see that {AB} appears in 2 transactions, {AC} appears in 2 transactions, but all other 2-item-sets appear in 1 or zero transactions. Therefore, the answer to the question would be 2 (only {AB} and {AC}). If $k = 2$ and $S = 3$, then the answer to the question would be 0 because no 2-item-sets appear in at least 3 transactions. Similarly, for $k = 3$, the k-item-sets would be {ABC}, {ABD}, {ABE}, {ABF}, {BCD}, …, {DEF}; for k=4, the k-item-sets would be {ABCD}, ..., etc.

Input

The input may consist of several test cases. Each test case starts with a line containing N ($0 < N < 1000$), the number of transactions. Then N lines follow, each line containing one transaction. Each transaction is a represented as a string (having only uppercase letters 'A'-'Z'), where each letter represents a particular type of item (e.g. A is bread, B is orange etc.). There will be maximum 15 items in a transaction, and each item will appear only once. After the transactions, there is one line containing two integers S ($0 < S < N$) and k ($0 < k < 8$), separated by space. The input ends with a value of 0 for N.

Output

For each test case, you are to output the number of k item-sets satisfying the minimum support S.

Sample Input / Output

transaction.in

```
4
ABC
AC
ABD
BEF
2 2
4
ABC
AC
ABD
BEF
3 2
0
```

OUTPUT

```
2
0
```

H. Areej Makeup Show

Program:	`areej.(cpp|java|py)`
Input:	`areej.in`

Description

Areej is TV announcer in Oman national TV. She has a weekly program that provides the audience with the latest makeup products in the market. The show offers a plenty of gift vouchers to promote the show and the channel. Areej requests her clients to send an SMS message to vote for their preferred makeup item for the day. Every few minutes, Areej needs to know the current favorite makeup item from the show viewers perspective. Once that is known, the next caller gets the gift voucher for that particular makeup item.

Example

There are three items in today's show I1, I2 and I3. Assume that at this moment, there are 25 votes for I1, 45 votes for I2, 15 voted for I3. If Areej decided to check the system now, then the next caller will win I2 voucher, which is the post popular item at this moment.

Input

The input consists of several test cases. For each test case, the first line consists of two integers separated by spaces: ***pCount*,** the number gift vouchers in today's show, and ***nCount*,** the number of items advertised today (0< *pCount* < *nCount* <= 10000). For each gift voucher, the input starts with the number of votes ***vCount*** (> 0) before gift voucher announcement, followed by ***vCount*** lines for the votes. Each vote line contains the unique item id. The item ids are expressed as strings in the following format: capital I followed by the item number (item number can be any integer between 1 to *nCount*), for example, "I1", "I2", … "I100", etc). The test cases end with a *pCount* value = 0.

Output

For each test case, output the item id for each gift voucher given. Once a gift voucher is given, the voting starts all over again. It is possible for the same item to appear in more than one vouchers. If there is a tie on the most popular item, then the item that first received the highest vote will be gifted. You should separate two test cases with a blank line.

Sample Input / Output

areej.in

```
3 4
6
I1
I2
I3
I2
I2
I1
4
I1
I1
I2
I2
5
I3
I2
I1
I3
I3
0 20
```

OUTPUT

```
I2
I1
I3
```

I. The League

Program:	`league.(cpp|java|py)`
Input:	`league.in`

Description

Most countries have a football league system consisting of a number of divisions that could be in series or interconnected. The divisions have a hierarchical format where teams could be promoted to a higher division or relegated to a lower division. For Example, the English football league system contains over 80 divisions across the country. This allows even the smallest teams and clubs to climb all the way to the top division – or level 1 (Premier League). Although this would be indeed a very difficult task.

The most common structure of the system allows the top 3 teams in a league division at the end of a season to be promoted to the next higher division in the following season, and in turn the 3 teams finishing at the bottom of the league division would be relegated to the next lower division. For example, if a team is playing in division 9 and finishes the season in the 2nd place, the team will be promoted to division 8 for the next season. If instead the team finishes in the last spot of the table, the team would be relegated to division 10. Teams that finish in the middle (neither top nor bottom 3) remain in the same division.

If a team is already in the top division (division 1) and finishes a season in the top 3, they will of course remain in the same division as no higher division exists. The rational is the same for a team playing in the lowest division and cannot be demoted to a lower one. In most leagues, selections from the top division are made for regional competition. For instance, the European Champions League is a competition where the top teams in the top division of a national league play. The number of teams selected for this type of regional leagues varies for each country and follows a specific system.

We need your help to design a system to let the organizers know which division a team should be in. You are given the number of divisions in a national league, the number of seasons played, and the number of times a specific team finishes a season in a position causing promotion, and the number of times the team finishes a season in a position causing relegation. Your task is to calculate the division that the team will play in for the next season, assuming the team started in the lowest division.

Input

The input starts with a number T ($1 \leq T \leq 1{,}000$) that represents the number of test cases in the file. Each test case is represented on a line that contains four integers D ($1 \leq D \leq 10^9$), S ($1 \leq S \leq 10^9$), P, and R ($0 \leq P + R \leq S$ *and* $0 \leq P - R \leq D$), representing the number of divisions, number of seasons, number of promotions for the team, and number of relegations for the team, respectively.

Output

The output for each test case is in this form:

```
k. Q
```

where ***k*** represents the test case number (starting at 1), and ***Q*** is the division the team will play in for next the next season.

Sample Input / Output

league.in

```
2
10 10 5 5
10 10 6 4
```

OUTPUT

```
1. 10
2. 8
```

J. The Company Trip

Program:	`trip.(cpp|java|py)`
Input:	`trip.in`

Description

Your company has done so well last year management has decided to plan a trip abroad for all employees. Little did they know, finding a destination country for the trip that all employees can get an on-arrival visa or free visa for was not an easy task. So as one of the greatest programmers of your generation, you come to the rescue.

Given the list of countries each employee can visit visa free or with visa on-arrival, your task is to find the list of countries that all employees can visit visa free or with visa on-arrival.

Input

The input starts with a number T ($1 \leq T \leq 100$) that represents the number of test cases in the file. Each test case starts with a line that contains one integers E ($1 \leq E \leq 1{,}000$), representing the number employees in the company. E lines follow, each starting with an integer N ($1 \leq N \leq 10{,}000$), representing the number of countries employee Ei can visit visa free or with visa on-arrival (Let's assume a future where we have that many countries. Maybe we also live on other planets by then). N numbers follow Ni ($1 \leq Ni \leq 100{,}000$), representing the countries employee Ei can visit visa free or with visa on-arrival (countries are represented as numbers).

Output

The output for each test case is in this form:

```
k. C1,C2,..Ci
```

where ***k*** represents the test case number (starting at 1), and ***Ci*** is the list of countries all employees can visit visa free or with visa on-arrival, comma separated, in ascending order. If no such country exists, print "No Trip" instead.

Sample Input / Output

trip.in

```
2
3
3 1 2 3
5 2 3 5 7 9
4 2 3 9 11
3
3 1 2 3
4 4 5 6 7
5 8 9 10 11 12
```

```
1. 2,3
2. No Trip
```

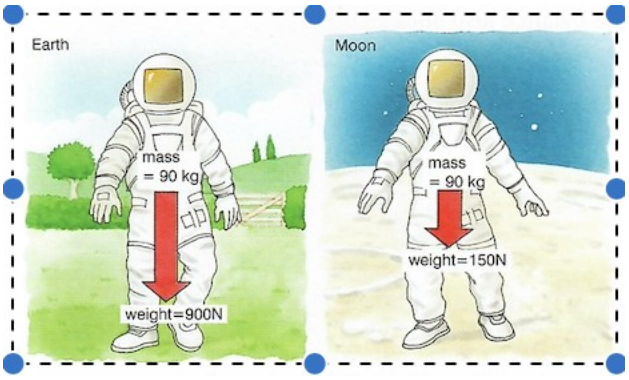
K. Your Weight On Other Worlds

Program:	`gravity.(cpp|java|py)`
Input:	`gravity.in`

Description

If you stood on the Moon, you would feel the gravity of the Moon pulling you down. Your mass would be the same as on earth, but your weight on the moon would be less. This is because the gravitational field strength on the Moon is about 1/6 of that on the earth, and so the force of the attraction of an object on the Moon is about one-sixth of that on the earth.

For example, if my mass is 80kg and we multiply it by the earth's gravitational field strength (10N/kg), my weight is 800N. Now, if I go to the moon, my mass will be the same, 80kg. We multiply that by the moon's gravitational field strength, 1.6 N/kg. That means my weight on the moon is 128N. That's why astronauts can jump high into the air on the moon because they're lighter up there. On the other hand, Jupiter is a very large planet with a strong gravitational field strength of 25 N/kg. If my mass is 80kg, then on Jupiter, my weight will be 25 x 80 = 2,000N. That means I wouldn't be able to get off the ground or stand up straight! I would probably be lying down all the time there. So weight varies depending on which planet you are on.



Input

The input starts with ***tCount***, the number of test cases, followed by *tCount* lines. Each test line contains the weight on earth in kg (a 32-bit integer) followed by a string representing a place in the universe (up to 100 characters long) and a floating point number that stands for its gravitational field strength. Everything is separated by a single space.

Output

For each case, output the weight on Earth and the weight on the other celestial body in Newtons. The format of the output should follow the example below:

```
Earth␣[force on Earth]N␣[name of celestial body]␣[force on other body]N
```

where "␣" is a single space. The first number should be an integer and the second number should be given with an accuracy of one decimal point.

Sample Input / Output

gravity.in

```
9
107 Moon 1.6
46 Sun 293.0
89 Saturn 10.5
77 Sun 293.0
106 Neptune 11.7
71 Venus 8.8
48 Uranus 9.0
104 Neptune 11.7
50 Sun 293.0
```

OUTPUT

```
Earth 1070N Moon 171.2N
Earth 460N Sun 13478.0N
Earth 890N Saturn 934.5N
Earth 770N Sun 22561.0N
Earth 1060N Neptune 1240.2N
Earth 710N Venus 624.8N
Earth 480N Uranus 432.0N
Earth 1040N Neptune 1216.8N
Earth 500N Sun 14650.0N
```