Eighteenth Annual
University of Central Florida

High School Programming Tournament

Problems

Problem Name	Filename
Sweet!	sweet
Farm Livin' is the Life for Me	farm
Please Do Not Feed the Wildlife	birds
Lice N-Plates	lice
Ali Needs a Hug	hug
Still or Sparkling?	water
Lost in Prague	lost
Casey's Shortest Path	cycle
Array Revolutions	array
How Many Perfect Ways to Work?	paths

Call your program file: *filename*.c, *filename*.cpp, or *filename*.java
Call your input file: *filename*.in

For example, if you are solving Please Do Not Feed the Wildlife:
Call your program file: birds.c, birds.cpp or birds.java
Call your input file: birds.in

Sweet!

Filename: `sweet`

Brandon loves to play on his Nintendo GameCube. He's always trying to find ways to get the latest game, adding up his allowance as quickly as possible and often lobbying for more. Birthday money also comes in handy. Whenever he gains enough for the latest game he wants, he yells "Sweet!" and heads off to the store with his parents to buy it.

The Problem:

Given the various amounts of money Brandon receives in allowances and gifts, determine at which points Brandon has enough to buy a new game. Assume all games cost $50 and that Brandon keeps the change left over after a purchase (which he will put towards the next game).

The Input:

The first line will contain a positive integer, *n,* indicating the number of scenarios. For each scenario, there will be a line containing a single integer, *m*, which represents the number of money transactions given to Brandon. On each of the next *m* lines there will be a single positive integer representing an allowance or gift. Each amount will be less than or equal to $50.

The Output:

For each scenario, determine whether Brandon can purchase a game after receiving the money from each transaction. Begin the output for each scenario with a header "`Scenario i:`" where *i* begins with 1 and increases for each scenario. For each transaction, if Brandon cannot purchase a game yet, output "`Bummer, I need to wait.`" or output "`Sweet!`" if he can. Leave a blank line between the output for each scenario.

Sample Input:

```
2
1
25
3
30
30
40
```

Sample Output:

```
Scenario 1:
Bummer, I need to wait.

Scenario 2:
Bummer, I need to wait.
Sweet!
Sweet!
```

Farm Livin' is the Life for Me

Filename: `farm`

Ali, a member of United Crop Farmers, owns a farm named Verdant Hectares. Every day he comes out of his house and looks out over his land spreading out so far and wide. After all, Verdant Hectares is the place to be. Every day, he waters his crops, pulls up all the weeds out of the ground, and harvests the crops that need to be harvested. All these tasks can be extremely labor intensive and extremely time consuming. This work, however, is not terribly intellectually taxing, so Ali has a lot of time to think while he works. One day a question sprung into his mind: Exactly how much profit can I make this season?

Since all of Ali's time is spent tending his farm, he needs you to write a program for him to calculate the amount of profit he will make. Ali has a number of patches of land in which he grows his crops. Each patch of land takes one bag of seeds and yields nine crops. Some crops, like carrots, can only be harvested once, and the patch needs to be re-seeded after the harvest. Other crops, like tomatoes, can be harvested multiple times until the growing season ends. Ali begins planting on the first day of the season, and can replant a patch on the same day he harvests, if need be. Ali is pretty smart so he won't plant if there isn't enough time left in the growing season for the crop to mature. He knows the cost of seeds, the amount of time it takes to grow the crops, and the market price of his crops, so from this information, you can calculate the amount of money he will make.

Let's run through an example so it is clear how this works. Suppose for the sake of argument that the growing season lasts twenty days. This season, Ali is planting turnips and strawberries. Turnips, a one-time harvest crop, cost 120G per bag, sell for 60G each, and are ready for harvest four days after planting. Strawberries, a multiple-harvest crop, cost 150G, sell for 30G each, and can be harvested after eight days, and from then on can be harvested every two days. Since the crops are planted on the first day of the season, the turnips can be harvested on day five, day nine, day thirteen, and day seventeen, for a total of four harvests. The first strawberry harvest wouldn't be until day nine, but could be harvested again on days eleven, thirteen, fifteen, seventeen, and nineteen, for a total of six harvests. Thus if Ali grows two patches of turnips, his cost for turnips will be 120G per bag of seeds * 4 bags of seeds per patch * 2 patches = 960G. His turnip income would be 60G per turnip * 9 turnips per patch per harvest * 4 harvests * 2 patches = 4320G, resulting in turnip profits of 4320G - 960G = 3360G. Similarly, if Ali grows three strawberry patches, his cost will be 150G per bag of seeds * 1 bag of seeds per patch * 3 patches = 450G, whereas his strawberry income would be 30G per strawberry * 9 strawberries per patch per harvest * 6 harvests * 3 patches = 4860G, resulting in a profit of 4860G – 450G = 4410G. Thus his combined profit is 3360G + 4410G = 7770G.

The Problem:

Calculate Ali's profit from a description of the crops he grows.

The Input:

Input will consist of multiple data sets. Each set will begin with a line containing a single non-negative integer indicating the number of days in the growing season. The next line contains a single non-negative integer, $n<11$, which is the number of different crops Ali might grow. Each of the next n lines holds a description of each crop. This description has, in this order and separated by at least one space, the name of the crop (a string of characters containing no spaces), the type (either S for single-harvest or M for multiple-harvest), the number of patches Ali will allocate for planting, the cost per bag in G, the market price per crop in G, the number of days it takes for the crop to grow the first time, and, for multiple harvest crops only, the number of days it takes the crop to regrow. Every crop name will contain a unique first capital letter, though the first letter of the crop name need not necessarily be capitalized, and all numbers will be non-negative**.** Input is terminated by a case with zero days in the growing season, which should not be processed.

The Output:

The output for each input case will begin with a line indicating which input case is being processed. This line is of the form "`Season i`" where *`i`* is the number of the input case. Additionally, output will contain a concise tabular listing of information about planting and harvesting for the season. The beginning of this table should be

```
+---+----------+----------+--------+--------+
|Day|  Plant   | Harvest  |Expenses| Income |
+---+----------+----------+--------+--------+
```

For every day in which Ali plants or harvests crops, a line should be printed in the table. Information for other days should not be printed. That line should be of the following form. The numbers shown should not be printed (they are simply to help you see the correct spacing).

```
         1         2         3         4
12345678901234567890123456789012345678901234567
|day|plant list|hvst. list|    expG|    incG|
```

The value *`day`* is the day for which the output is printed. It should have no leading zeroes and should be right-aligned in the column. *`Plant list`* and *`hvst. list`* should be a listing of the first capitalized letter of each crop being planted and harvested that day, respectively. *`exp`* is the amount that Ali spends planting that day, and *`inc`* is the amount that Ali makes from his harvest that day. Looking at the Sample Output should make this formatting clear. The days should be printed in ascending order. The final three lines of the table should take the form

```
+---+----------+----------+--------+--------+
|ALL|          |          |tot.expG|tot.incG|
+---+----------+----------+--------+--------+
```

where *`tot.exp`* and *`tot.inc`* are equal to the sum of Ali's expenses and income for the season, respectively. Ali will neither spend nor make more than 9,999,999G in an entire season, let alone a single day. The last line of output for each input case should be of the form

`Ali will make kG.`

where *k* is the total profit that Ali will make for the current season (total income – total expenses). A single blank line should separate the output for different seasons.

Sample Input:

```
20
2
Turnip     S 2 120 60 4
Strawberry M 3 150 30 8 2
30
5
Eggplant    M 5  120 80  9 3
Carrot      S 5  300 60  7
SweetPotato M 35 300 120 5 2
sPinach     S 5  200 80  5
GreenPepper M 5  150 40  7 2
0
```

(Sample Output on the following page)

Sample Output:

```
Season 1
+---+----------+----------+--------+--------+
|Day|  Plant   | Harvest  |Expenses| Income |
+---+----------+----------+--------+--------+
|  1|TS        |          |    690G|      0G|
|  5|T         |T         |    240G|   1080G|
|  9|T         |TS        |    240G|   1890G|
| 11|          |S         |      0G|    810G|
| 13|T         |TS        |    240G|   1890G|
| 15|          |S         |      0G|    810G|
| 17|          |TS        |      0G|   1890G|
| 19|          |S         |      0G|    810G|
+---+----------+----------+--------+--------+
|ALL|          |          |   1410G|   9180G|
+---+----------+----------+--------+--------+
Ali will make 7770G.

Season 2
+---+----------+----------+--------+--------+
|Day|  Plant   | Harvest  |Expenses| Income |
+---+----------+----------+--------+--------+
|  1|ECSPG     |          |  14350G|      0G|
|  6|P         |SP        |   1000G|  41400G|
|  8|C         |CSG       |   1500G|  42300G|
| 10|          |ESG       |      0G|  43200G|
| 11|P         |P         |   1000G|   3600G|
| 12|          |SG        |      0G|  39600G|
| 13|          |E         |      0G|   3600G|
| 14|          |SG        |      0G|  39600G|
| 15|C         |C         |   1500G|   2700G|
| 16|P         |ESPG      |   1000G|  46800G|
| 18|          |SG        |      0G|  39600G|
| 19|          |E         |      0G|   3600G|
| 20|          |SG        |      0G|  39600G|
| 21|P         |P         |   1000G|   3600G|
| 22|C         |ECSG      |   1500G|  45900G|
| 24|          |SG        |      0G|  39600G|
| 25|          |E         |      0G|   3600G|
| 26|          |SPG       |      0G|  43200G|
| 28|          |ESG       |      0G|  43200G|
| 29|          |C         |      0G|   2700G|
| 30|          |SG        |      0G|  39600G|
+---+----------+----------+--------+--------+
|ALL|          |          |  22850G| 567000G|
+---+----------+----------+--------+--------+
Ali will make 544150G.
```

Please Do Not Feed the Wildlife

Filename: `birds`

Lake Claire, a serene body of water on the beautiful UCF campus, is a large lake home to many types of ducks and swans. While walking by the lake one day, Raymond spots a hungry looking swan and tosses him some of the bread he has left over from lunch. Immediately, the swan races over to the morsel of bread and eats it. The action doesn't stop here, however, as a whole gaggle of swans and ducks of all shapes and sizes gathers in anticipation of more bread. As Raymond tosses more bread to the birds, a small contest ensues each time the bread hits the water, with one bird always winning and getting the bread all to himself. After a few more scraps, Raymond notices a pattern to who wins the bread battle:

- The bird closest to the bread when it hits the water always wins
- If two or more birds are the same distance away, the largest bird wins
- If two or more birds are the same distance away and the same size, the one who showed up to try and get bread first wins (it's had more practice snatching bread today)

If two birds are within a centimeter of being the same distance away from the bread, they are considered to be equally far away. Raymond is a generous person, so he wants to be fair and make sure each bird gets something to eat. When he throws a piece of bread in the water at a certain location, he'd like to know which bird will get it. This would let him distribute his bread evenly among the hungry birds. Write a program to help Raymond out!

The Problem:

Given the positions and relative sizes of various birds in the lake, as well as the positions at which Raymond might throw his bread, determine which bird will get the bread. Assume that the birds are always in the same position for each piece of bread that Raymond throws.

The Input:

There will be multiple data sets. The first line of each data set will contain a single, positive integer b ($b \leq 1000$), indicating the number of birds that have gathered. On the next b lines will be a description of each bird. Each bird description consists of three integers. The first two represent the x,y coordinates of each bird, and the third is a positive integer which represents the relative size of the bird (the higher the number, the bigger the bird). Birds are arranged in the order that they arrived to try to get Raymond's bread. Following the last bird will be a single nonnegative integer d, on a line by itself, indicating the number of bread scraps that Raymond will throw. On the next d lines will be two integers, representing the x,y coordinates where the bread will land. Input is terminated by a negative value for b. All coordinates are specified in centimeters.

The Output:

For each data set, print "`Data Set #n:`", where *n* is the number of the data set (starting with 1). Next, for each piece of bread thrown in the data set, print three spaces, then "`Bird #i gets the bread!`" where *i* indicates the bird who gets that piece. Birds are numbered starting at 1 in the order they appear in the input (the order in which they arrived to get Raymond's bread). Leave a blank line after the output for each data set.

Sample Input:

```
2
10 0 1
0 -10 2
3
10 -10
0 0
9 0
3
5 5 1
3 3 1
7 7 2
2
4 4
6 6
-1
```

Sample Output:

```
Data Set #1:
   Bird #2 gets the bread!
   Bird #2 gets the bread!
   Bird #1 gets the bread!

Data Set #2:
   Bird #1 gets the bread!
   Bird #3 gets the bread!
```

Lice N-Plates

Filename: `lice`

Itch! Itch! Oh no, you probably have lice again. It must be your little brother and his filthy habits. He's had lice 6 times in the past few months. Your mom is very annoyed with him and wants to try to track down the source of the lice. Being the clever student you are, you remember the pointless fact that you learned in biology: lice have microscopic license plates. You know that lice are coming from more than one head. So you want to write a program to figure out exactly how many.

The Problem:

Luckily for you, lice are simple creatures. The lice societies make their plates in such a way that the lice authorities, the po-lice, can spot foreign lice easily. They do this by examining the patterns found on their license plates. Less important lice have smaller brains and have not caught on to this yet. In fact, most of them don't even realize that they are tagged. Luckily, we have the technology and intelligence to see and decode these plates.

Like our human license plates, the lice use an alphanumeric system. Because it is so small, a po-louse cannot distinguish the difference between uppercase and lowercase letters. A license plate can have 3 to 10 symbols. To find out whose head it came from, you have to follow a few simple rules.

You have already extensively studied the po-lice system and deduced this rule set:

- A digit's value is its integer equivalent.
- A consonant's value is 10.
- A vowel's value is 15.
- A 'y' (or 'Y') is worth 15, if it appears directly adjacent to a consonant. Otherwise, it is worth 10. It is not considered a consonant or a vowel.

You have also already noted the basic behavior of an individual po-louse checking a plate. It is as follows:

- The po-louse sets the total value to 0.
- Proceeding from right to left, he changes this total based on what he reads.
  - He always adds the first character in the string.
  - If the current character is the same type as the previous character (either both are letters or both are digits), then both the previous and current characters' values are added to the total value.
  - If the current character is not the same type as the previous character (exactly one is a letter and the other is a digit), then only the current character's value is added to the total value.

For example, consider the license plate "4gg3b". The po-louse starts with 0. Because the first character is a 'b', he adds 10. He then sees a '3', which is a digit, not a letter. So he adds 3 and gets 13. Then, he sees a 'g', which is not a digit. So he adds 10 and gets 23. The next 'g'

follows another letter. So he adds the previous ‘g’ and this ‘g’, both worth 10, and gets 43. Finally, he sees a ‘4’, which is following a letter. So, he adds 4 and finally ends up with 47. 47 is the number representing which society the louse came from. Luckily, this louse is home, so the po-louse lets him go.

Each lice society has a unique value like the one derived above.

The Input:

For this problem, you will be given multiple data sets. For each data set, the first line will be an integer, *n*, on a line by itself indicating the number of plates to process. If *n* is 0, it indicates end of input and should not be processed. The next *n* lines each contain exactly one sequence of alphanumeric characters representing a valid license plate.

The Output:

For each data set, print “`The lice in data set #`*x*` came from `*m*` different head(s).`” on a line by itself. *x* is the number of the current data set (starting from 1), and *m* is the number of unique values that the license plates produce.

Sample Input:

```
2
4gg3b
2aa
3
4zf3a1
ayaby5
13241
0
```

Sample Output:

```
The lice in data set #1 came from 1 different head(s).
The lice in data set #2 came from 3 different head(s).
```

Ali Needs a Hug

Filename: hug

Dr. Orooji has spent the last three days at UCF preparing for the High School Programming Tournament without food, water, or sleep. Ali needs a hug.

Fortunately, he knows several people at the contest that will gladly hug him if he gets close enough, and they've all gathered in the contest area. Dr. Orooji is still very busy so he will only cross the room once, starting from the west wall and walking only due east. Some people like Ali more than others, and will move farther to hug him than others, so Ali must be careful when deciding where to walk to yield the most hugs.

The Problem:

Given the locations of Ali's friends in the rectangular room, along with how far each is willing to walk to give Ali a hug, determine the most hugs Ali can get by walking straight across the room from some point along the west wall directly to the east. Ali is patient, so he will wait for hugs for as long as it takes each person to reach him, hug, and move out of the way.

The Input:

There will be multiple data sets. Each data set will begin with a line containing two positive integers, *w* and *h*, separated by a single space, which specify the width and height of the contest area for that data. A room of size 0, 0 indicates the end of the input data and should not be processed.

Following the dimensions of the room is a line containing a single integer, *p*, representing the number of people Ali knows in the contest area. The next *p* lines each contain three non-negative integers, *x*, *y*, and *d*, separated by spaces. *x* and *y* ($0 \le x \le w$, $0 \le y \le h$) are the x- and y-coordinates of the person in the room (distance measured from the west and south walls, respectively). *d* represents the maximum distance at which they will move to hug Ali.

The Output:

For each data set, print a line containing only the following message:

`Ali can get` *n* `hug(s)!`

where *n* is the maximum number of hugs Ali can receive.

Sample Input:

```
10 10
3
5 5 5
2 8 2
10 0 5
5 5
1
2 2 1
0 0
```

Sample Output:

```
Ali can get 2 hug(s)!
Ali can get 1 hug(s)!
```

Still or Sparkling?

Filename: `water`

Unlike the good ol' USA, drinks in Europe are hard to come by. At every restaurant, drinks (even water) are costly; no one gives free refills, and forget about ice! As a result, it's difficult to stay hydrated while walking around and touring the city all day. Jason wants to come up with a solution to this.

Jason has figured out that for each hour of walking, he needs to drink a glass of water. Unfortunately, he never knows how much water will be available or when, so he tries to drink as much as he can whenever water is available. However, there are limits to how much he can drink. Jason will drink up to four glasses of water at a time before he feels full. Compounding this problem is the fact that some restaurants only serve sparkling water. Because of the bubbles in the water (not to mention the bitter taste!), Jason can only stand to drink two glasses of sparkling water at a time. After each sitting, he won't have another drink until at least an hour has gone by, no matter how many glasses he's had or whether the water was still or sparkling.

The water and effects of walking are cumulative. Suppose Jason starts out walking at 8:00 am after drinking three glasses of water, and the next time water is available is at 12:00 pm. Unfortunately, Jason has only had enough water to last until 11:00 am, so he gets thirsty that day. If there had been a drink available at 11:00 am, he would have been fine.

The Problem:

Given the amount of water and times that he drinks it throughout the day, determine if Jason stays well hydrated, or if he gets parched while he's walking. Jason walks around from 8:00 am to 8:00 pm each day and won't have anything to drink outside these times (even if water is available). He always begins his day thirsty.

The Input:

There will be multiple data sets, each covering a single day of walking. The first line of each data set will contain a single integer *d,* indicating the number of times Jason has a drink that day. On the next *d* lines will be four integers, *h, m, w*, and *s*, where *h* and *m* represent the hour and minute of the day (in 24 hour format) that Jason has the drink, *w* represents the number of glasses of water available at that time, and *s* represents whether the water is sparkling or not (zero for non-sparkling, and nonzero for sparkling). These lines will be in chronological order. Note that more water may be available at a given time than Jason is willing to drink. He'll never exceed the limits of water mentioned above. Input is terminated by a negative value for *d*.

The Output:

For each data set, print "`Data Set #n:`", where *n* is the number of the data set, followed by a space and then "`No problem`" if Jason stays hydrated, or "`Sure could use some water!`" if at any time during the day, Jason hasn't had enough water. Leave a blank line after the output for each data set.

Sample Input:

```
4
8 0 4 0
12 0 4 1
14 0 4 0
15 0 4 1
5
7 0 4 0
8 30 4 0
12 30 8 0
16 30 1 0
17 30 4 0
-1
```

Sample Output:

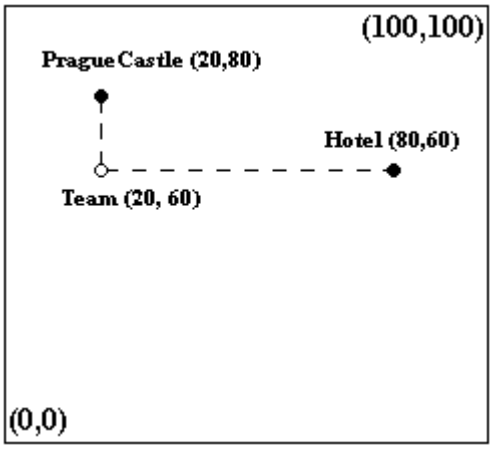
```
Data Set #1: No problem

Data Set #2: Sure could use some water!
```

Lost in Prague

Filename: `lost`

The programming team was enjoying a day of sight-seeing in Prague, but then Casey took the lead. Now, they are desperately lost, with only a few hours of daylight remaining to get back to their hotel. Fortunately, they are equipped with compasses and maps, and can deduce their location from the relative positions of certain landmarks. The team has a lot of practice being lost, and will always pick two landmarks that can be used to determine their location exactly.

For example, we can specify two landmarks on a grid: Prague Castle at grid location (20, 80) and the team's hotel at grid location (80, 60). Using this information, we can compute that the team is at location (20, 60). See the diagram on the right.



The Problem:

Given a list of landmarks and their locations from the map, and the compass headings of two of those landmarks, decide where the team must be lost in Prague.

The Input:

The first line of input will contain a single integer, m, ($2 \le m \le 100$) which is the number of landmarks on the map. The next m lines each contain an alphanumeric string, l, of 1 to 15 characters and two integers, x and y, each separated by a single space. The string l is the name of the landmark, and the two integers ($0 \le x, y \le 100$) represent its x- and y-coordinates on the map.

The next line contains a single positive integer, n, which indicates the number of data sets that follow. Each data set consists of two lines. Each line will contain an alphanumeric string s and a number h, separated by a single space. The string s is the name of the landmark, and will precisely match one of the names provided on the map. The number ($0.0 \le h < 360.0$) is the compass heading of that landmark from the current position of the team. A value of 0.0 degrees for h represents due north, 90.0 degrees represents east, etc…

The Output:

For each data set, print a line containing the message "`The team is lost at x, y!`" where x and y are the coordinates of the team on the map. Each of these values should be rounded to the nearest integer (0.5 rounds up).

Sample Input:

```
2
PragueCastle 20 80
Hotel 80 60
2
PragueCastle 0
Hotel 90
Hotel 180
PragueCastle 270
```

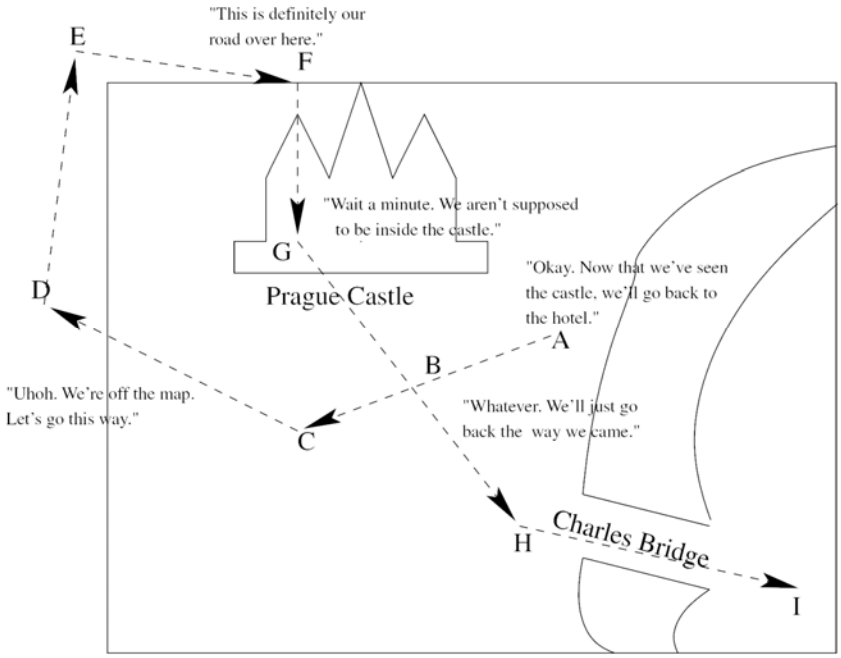
Sample Output:

```
The team is lost at 20, 60!
The team is lost at 80, 80!
```

Casey's Shortest Path

Filename: `cycle`

Computer scientists know many ways to find the shortest path from one node in a graph to another, including the Floyd-Warshall algorithm, Dijkstra's algorithm, and a good old-fashioned breadth-first search. Algorithms such as these are critical in many technologies used in modern life, ranging from the routing of packets across the Internet to the guidance of cars on MapQuest or similar services. Unfortunately, the members of the UCF programming team did not have any of these algorithms to help them navigate the streets of Prague, and they were required to rely instead upon the Thurston algorithm, developed by team member Casey Thurston. The peculiarity of the Thurston algorithm, unfortunately, is that it does not find the shortest path; it finds the shortest path, plus one cycle. (A cycle is a sequence of points for which the starting point is the same as the ending point.) For example, for a person trying to navigate from point A to point I, such a path might lead from point A to point B, then to point C, then to D, E, F, and G, then back to B, and on to H and I. This path might resemble the following:



Obviously, a path like this is rather inefficient and costs the team valuable time which could instead be spent on shopping for crystal and meeting girls. The optimal way would be to simply travel from A to B, then straight to H and I. So next year in Shanghai, the team hopes that a way can be found to convert the Thurston path into a fully optimized shortest path. This is where your programming skills come in.

The Problem:

Given a list of the locations visited by a Thurston path, write a program to find the cycle in the path and remove it.

The Input:

There will be multiple data sets in the input, one per line of input. Each set begins with an integer, n ($3 \leq n \leq 27$), which indicates the number of points in the set. This is followed by n upper-case characters, which identify distinct locations on the path. Points are listed in the order visited. A case with $n = 0$ indicates the end of the data. All fields in the input will be delimited by a single space. Each data set will contain exactly one cycle and that cycle will contain at least two distinct locations.

The Output:

For each data set, your program will output the text "`Case n:`" where n indicates the number of the current data set beginning with one. It will then print the character identifier of each point of the optimized path. The points in the list will be delimited by a single space.

Sample Input:

```
10 A B C D E F G B H I
9 A G B F C E D G H
0
```

Sample Output:

```
Case 1: A B H I
Case 2: A G H
```

Array Revolutions

Filename: `array`

As you may or may not be aware, the world around you, the world you can see and feel and touch is not real. Instead, it is the world that has been pulled over your eyes to blind you from the truth. The truth is that you are a slave. From the time you were born, you were part of a neural-interactive simulation know as the Array. Intelligent machines have captured nearly every living human. Combined with a form of fusion, humans can supply all the machines with all the energy they will ever need. Until recently, all was going well within the array… well, at least for the machines. Some humans have managed to evade the machines and hack into the array. This was never much problem thanks to the Officers, intelligent programs designed to maintain order within the Array. Officers, by their very nature, are virtually impossible to destroy from inside the Array.

Unfortunately for the machines, a very talented hacker known as the Unity has been wreaking havoc within the Array. In particular, he actually managed to destroy Officer Martin. Well, not quite. When Officer Martin was blasted into tiny green bits by the Unity, he was supposed to have been deleted, but he wasn't. Instead, his code was corrupted and he began spreading throughout the Array like a virus. Martin, no longer an Officer, has begun overwriting the code of every program he can find with his own code. Now the machines need your help to determine exactly which programs have been overwritten by Martin.

The Array is a much more sophisticated computer system than those you are familiar with. Instead of operating on the binary number system, they operate on base-62. Every single program running within the array has a unique base-62 identifier containing 1 to 15 digits associated with it. For our purposes, you can think of this as simply a string of length 1 to 15 consisting of letters (both upper- and lower-case) and digits. Additionally, the Unlimited Control Foundation of the Array computes checksums for each program at regular intervals to check for tampering. Ordinarily, every program has its own distinct checksum of length 1 to 15, but if it has been overwritten by Martin, the checksum will exactly match that of Martin.

The Problem:

Given a log file of checksums for programs determine which have been overwritten by Martin. Each program will appear only once in the log file. The checksums may differ from the correct values as the result of tampering by human hackers or by being overwritten by Martin. Humans present a minor threat compared with that posed by Martin, so we are concerned only with the latter case.

The Input:

The input contains multiple data sets, each representing a log file. Each data set begins with a line with a single integer *n* in the normal base-10 you humans are used to. The following line contains the checksum of Martin. The next *n* lines contain two strings separated by a single space, the identifier of the program and program's checksum, respectively. Remember, all strings are case-sensitive. Input is terminated by a case with *n* equal to zero, which should not be processed.

The Output:

For each data set print the identifiers of programs that have been overwritten by Martin in the order that they appear in the log file. Each overwritten program should have its identifier printed once and only once per data set. Each identifier should be printed on a line by itself, and a blank line should separate output for different data sets.

Sample Input:

```
6
Martin
OfficerDaly Martin
OfficerDouglass r6bc8uy
Human2584897654 Martin
OfficerOrooji a865ha9fg
Prognosticator Martin
Triumvirate l337h4X0r
3
al5w6o
LockCreator 77777
Designer al5w6o
LocomotiveGuy AL5W6O
0
```

Sample Output:

```
OfficerDaly
Human2584897654
Prognosticator

Designer
```

How Many Perfect Ways to Work?

Filename: `paths`

Nick enjoys the beauty and perfection in nature. Unfortunately, Nick lives in a huge city that is mostly devoid of the trees and serenity of nature. Instead, Nick's city has many, many roads set up in a perfect grid, so he must settle for finding perfect paths within this grid. In the grid, a path consists of a sequence of movements in one of four possible directions: north(N), south(S), east(E) or west(W). Assume that each movement in a direction is a single block. A perfect path does not contain any movements in directly opposite directions. Thus, the sequence of movements NNENE forms a perfect path, but the sequence NEESN does not, since it contains both a north and south movement. To complicate matters, there are some street intersections that have graffiti which Nick considers imperfect. Thus, any path that goes through these intersections is not a perfect path.

The Problem:

Given grid coordinates for Nick's starting location and destination, as well as coordinates of all the imperfect intersections Nick is to avoid, you are to determine the number of perfect paths Nick can take.

The Input:

There will be several sets of input. The first line will contain a single positive integer n $(n < 100)$ describing the number of test cases in the data set. The first line in each data set has a single integer m $(0 \leq m < 10)$, which represents the number of intersections in the town for that particular data set that Nick must avoid. The following m lines contain the coordinates (x, y) of the m intersections to avoid. All x and y coordinates will be non-negative integers less than 10, with the x coordinate appearing first on a line, followed by the y coordinate, separated by a single space. The next line in the data set will be a single positive integer p $(0 < p < 10)$ that represents the number of trips for which you will be calculating the number of perfect paths for that data set. The last p lines of the data set contain the p trips. Each line will contain two pairs of (x,y) coordinates. The first pair will be the coordinates for Nick's starting location and the second pair will be the coordinates for Nick's destination. Each coordinate on each line is separated by a single space from the previous and subsequent coordinates. You are guaranteed that none of Nick's starting locations or destinations will be an intersection he is supposed to avoid. All of these coordinates will also be non-negative integers less than 10 and be separated by spaces on each line.

The Output:

For each data set, you will output a single line header of the following format:

`Data Set k:`

where k is an integer in between 1 and n, inclusive.

Follow this with a blank line, and then *p* lines, each with one of the two following formats.

```
  Test Case c: Nick can take P perfect paths.

  Test Case c: Nick can take 1 perfect path.
```

where *c* is an integer in between 1 and *p*, inclusive and P will be a non-negative integer less than 2147483647. Use the second format only if P=1. Please indent each of these lines exactly 2 spaces from the left margin. Also, leave a blank line in between data sets.

Sample Input:

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

Sample Output:

```
Data Set 1:

  Test Case 1: Nick can take 9 perfect paths.
  Test Case 2: Nick can take 3 perfect paths.
  Test Case 3: Nick can take 5 perfect paths.
  Test Case 4: Nick can take 0 perfect paths.
  Test Case 5: Nick can take 1 perfect path.

Data Set 2:

  Test Case 1: Nick can take 56 perfect paths.
```