University of Central Florida

2012 (Fall) Local Programming Contest

Problems

Problem#	Filename	Problem Name
1	change	Exact Change
2	hanging	Hanging Nests
3	fold	Fold the Paper Nicely
4	hiking	Happy Hiking Grounds
5	palindrome	Palindrome Maker
6	callme	Call Me Maybe
7	spaces	Spaces, the Final Frontier
8	rummy	Rummy Score
9	circles	Simi Circles
10	dungeon	Dungeon Trouble!
11	safepizza	Safely Stacking Leftover Pizzas

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

For example, if you are solving Call Me Maybe:

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

UCF Local Contest — September 1, 2012

Exact Change

filename: `change`

Whenever the UCF Programming Team travels to World Finals, Glenn likes having the exact amount of money necessary for any purchase, so that he doesn't have to count and receive change. Of course, most countries don't have many different denominations of coins, so Glenn creates different "packages" with him, each with some particular amount of money, in cents. Glenn would like to know which amount of money (in cents), is the smallest that he can't pay for exactly, with some combination of his packages.

The Problem:

Given a list of positive integers, determine the smallest integer that can't be represented as the sum of some subset of the integers on the list.

The Input:

The first input line contains a single positive integer, n $(1 < n \le 100)$, indicating the number of sets of coin packages to evaluate. Each of the n input sets follows. The first line of each input set contains only an integer, c $(1 \le c < 31)$, representing the number of different packages of coin for that input set. The following line contains exactly c positive integers, each separated by a single space, representing the value of each of the c packages in cents. The sum of these c integers is guaranteed not to exceed 10^9. Note that the package values are not necessarily distinct, i.e., there may be multiple packages with the same value.

The Output:

For each set of packages, first output "`Set #i: `" where `i` is the input data set number, starting with 1. Follow this with a single positive integer, the smallest value that can't be represented as a sum of the values of a subset of the packages given. Note that a package value can be used at most once in a subset unless there are multiple packages with that value (if there are m occurrences of a package value, up to m occurrences of that value can be used in a subset). Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
3
6
12 8 1 2 4 100
3
1 2 3
6
3 1 3 2 3 3
```

Sample Output:

```
Set #1: 28

Set #2: 7

Set #3: 16
```

UCF Local Contest — September 1, 2012

Hanging Nests

filename: `hanging`

The Turing Tern is a newly discovered species of bird that is known for its highly regular behavior patterns. Every member of a given flock is uniquely identifiable by the number of colored spots on its wings, i.e., no two members will have the same number of spots.

When a flock migrates to a new area, it finds a tall tree and builds a hook-like structure on the underside of the highest branch. Now each member will build a nest for itself. Every nest will hang from a hook, and will itself have up to two hooks on the underside – one on the far left, and the other on the far right.

Here are the rules used to build nests:

1) Birds always build nests in the order they arrive.
2) Every nest will hang off of exactly one of the hooks of another nest or the hook-like branch. A hook can hold at most one nest.
3) Every bird will try to hang its nest on the original hook-like branch. If that hook is being used, it will use rule (4).
4) If there is already a nest hanging on the hook that a bird is trying to use, it will compare its spots with the owner of that nest. If it has fewer, it will try to build its nest on the left hook of that nest, otherwise it will try to build it on the right hook. This procedure will repeat until it finds an empty hook.

The Problem:

The Turing Tern is endangered, and researchers have recently figured out why. When large flocks build their nests, the resulting structure is likely to become unstable. Inevitably a nest falls off the hook, taking along all the nests below it too. (While falling is rarely a problem for birds, flying is rather difficult when half a dozen nests and their occupants unexpectedly fall on your head in the middle of the night.)

We define a *hanger* of a nest as a sequence of nests connected by hooks, starting from that nest and moving one nest down at every step (never sideways) until it reaches a nest on the last level. The *hanger length* is just the number of nests in the hanger.

The *height* of a nest is defined as the length of its largest hanger. The *instability factor* of a nest is the absolute difference of the height of the nest on its left hook and the height of the nest on its right hook. (If there is no nest on a hook, the height of that 'null nest' is treated as 0.)

Your program, given a bird flock description, should output the number of spots on the bird which lives in the nest with the highest instability factor. If there are multiple such birds, output the one that arrived the earliest.

The Input:

The first input line contains a positive integer, *n*, indicating the number of flocks. Each input case takes up two lines. The first line is an integer, f $(1 \leq f \leq 5000)$, indicating the number of birds in the flock. The next line consists of *f* positive integers, each representing the number of spots on a bird, in the order they arrive. You may assume that no two birds have the same age. Each bird is at least 1-year old and at most 5000-year old.

The Output:

For each test case, output a single line of the form "`Flock #k: p`" where *k* is the flock number, and *p* is the number of spots on the bird living in the most unbalanced nest. Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
3
10
10 14 12 4 8 19 18 17 9 16
5
4 5 3 2 1
15
50 67 19 42 18 66 168 1 52 57 37 64 78 22 130
```

Sample Output:

```
Flock #1: 14

Flock #2: 4

Flock #3: 66
```

UCF Local Contest — September 1, 2012

Fold the Paper Nicely

filename: `fold`

Dr. Orooji has a daily calendar (365 pages) on his desk. Every morning, he tears off one page and, as he is reading the notes on the next page, he folds (out of habit) the sheet in his hand. Dr. O noticed that he always folds the sheet (a rectangular paper) along the longer side, e.g., if one side is 80 and the other side is 60, he would fold along 80; this will make the paper of size 40 and 60; if he folds it again, he would fold along 60 since that's the longer side now.

The Problem:

Given a rectangular piece of paper and how many times Dr. O folds it, you are to determine the final sizes. When folding a side with an odd length, the fraction is ignored after folding, e.g., if a side is 7, it will become 3 after folding.

The Input:

The first input line contains a positive integer, *n*, indicating the number of data sets. The sets are on the following *n* input lines, one set per line. Each set contains three positive integers (each ≤ 10000), the first two providing the rectangle sides and the third providing the number of folds.

The Output:

At the beginning of each test case, output "`Data set:` *`v`*" where *`v`* is the input values. Then, on the next output line, print the final values for the rectangle (larger side of the final values first). Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
3
60 51 4
3 2 50
3 2 1
```

Sample Output:

```
Data set: 60 51 4
15 12

Data set: 3 2 50
0 0

Data set: 3 2 1
2 1
```

UCF Local Contest — September 1, 2012

Happy Hiking Grounds

filename: `hiking`

Arup likes hiking and, since he just taught trees in CS I, he has found a trail that is in the form of a tree. For simplicity, Arup always uses A to refer to the tree root. So, a sample trail might be:

```
        A
      / | \
     /  |  \
    /   |   \
   B    J    C
 / | \  |   / \
D  E  F I  H   G
```

Arup has decided to follow a Breadth First Search (BFS) to hike the trail and, for simplicity, Arup visits the children of a node in alphabetical order. The BFS traversal of the above sample tree is:

```
A B C J D E F G H I
```

But Arup quickly realizes that to go from one child to a sibling, he has to go back to the parent again (since there is no direct path between a child and its sibling). So, for the above tree, Arup will actually walk as follows:

```
A-B-A-C-A-J-A-B-D-B-E-B-F-B-A-C-G-C-H-C-A-J-I-J-A
```

Note that Arup returns to the root at the end of his hike. Well, this is a lot more walking than Arup had expected from BFS but Arup doesn't mind since his wife (a medical doctor) has convinced him of the benefits of exercise and good eating habits!

The Problem:

Given a trail description (a tree), determine the total distance (cost) of the described Breadth First Search hike of the trail including the distance (cost) of backtracking at each step.

The Input:

The first input line contains a positive integer, *t*, indicating the number of trails (trees) to check. Each trail is defined by a series of paths that connect junctions and points of interest (each of which is represented by a single uppercase letter). On the following input line there will be an integer, n $(1 \le n \le 26)$, providing the number of tree nodes (junctions and points of interest); assume the nodes will be the first *n* uppercase letters. On each of the next $n-1$ input lines, there will be two uppercase letters, indicating that there is an edge between those two locations (first node being the parent of the second node), and a positive integer representing the length of that edge (the distance between the two nodes); the three values on these input lines are separated from each other by a single space.

Arup will always start at the entrance, which is labeled as "A" (which also is guaranteed to appear in the input).

The Output:

At the beginning of each trail, output the message "`Trail #`*`i`*`:`" on a line by itself where *`i`* is the number of trail in the input (starting with 1). On the next output line, output the message "`BFS hike will cost` *`b`*" where *b* is the total distance of Arup's hike in his breadth first fashion (this must include the backtracking cost). Assume the total distance will fit in an integer.

Note that, at each location, Arup will always visit each of the next possible locations in alphabetical order. Leave a blank line after the output for each trail. Follow the format illustrated in Sample Output.

Sample Input:

```
2
3
A B 5
A C 3
8
A B 1
A C 7
B D 5
B E 3
B F 2
C G 4
C H 2
```

Sample Output:

```
Trail #1:
BFS hike will cost 16

Trail #2:
BFS hike will cost 64
```

UCF Local Contest — September 1, 2012

Palindrome Maker

filename: `palindrome`

A “palindromic integer sequence” is a sequence that is the same when written forwards or backwards, i.e., of the form {a1, a2, a3, …. , a3, a2, a1}. Some examples of palindromic integer sequences are {78, 91, 78}, {100}, {10, 20, 20, 10}, and {5, 5, 5, 5}. But {1, 2, 3, 1} and {10, 20} are not palindromic sequences.

You are given an integer sequence. You want to convert the sequence into a palindromic integer sequence by a series of insertions. You can convert {1, 2, 3, 1} to a palindromic sequence by inserting a 2 at the fourth position which will become {1, 2, 3, 2, 1}, and you can convert {10, 20} to palindromic sequence by inserting 20 at the first position or inserting 10 at the last position.

The Problem:

Given an integer sequence, determine the minimum number of insertions required to convert it into a palindromic sequence. It is guaranteed that the result (number of insertions) will always be less than 100.

The Input:

The first input line contains a positive integer, *n*, indicating the number of integer sequences to check. The sequences are on the following *2n* lines, two lines per sequence. First line contains *s* ($1 \leq s \leq 10{,}000$), the size of the sequence. Second line contains *s* integers (separated by a single space), providing the sequence. The integers in a sequence are between 1 and 50 inclusive.

The Output:

For each sequence, first output “`Sequence #i: `” where *`i`* is the sequence number, starting with 1. Then, output a single integer denoting the minimum number of insertions required to convert the sequence into a palindromic sequence. Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
3
4
1 2 3 1
4
1 2 3 4
3
1 2 1
```

Sample Output:

```
Sequence #1: 1

Sequence #2: 3

Sequence #3: 0
```

UCF Local Contest — September 1, 2012

Call Me Maybe

filename: `callme`

The popular song, “Call Me Maybe” by Carly Rae Jepson, has been spoofed with great comedic success on the internet recently. In particular, one of the YouTube videos splices words from different speeches by President Obama to string together the lyrics of the song. (Rumor has it that Angela Merkel is eagerly awaiting Mr. Obama’s call.) You have decided that this idea is comic gold. Given a library of a particular person’s speeches as well as the lyrics to a song, you want to spoof them by writing a program to automatically check the text of all of the speeches to make the appropriate word substitutions from the speeches to the song lyrics. For example, the first line of “Call Me Maybe” is “I threw a wish in the well.” Perhaps “I” appears as the 10th word in speech number 4, “threw” appears as the 54th word in speech number 12, etc. Of course, if the speaker never said “wish” in any of his speeches, then the spoof is not possible. With your program, you can spoof almost any song with any celebrity and get rich!

The Problem:

Given a list of speeches by an individual, as well as the text of a song to spoof by pulling words from those speeches, determine if the spoof is possible and, if so, give a listing of which words in which speeches to use as substitutes.

In order to obtain a unique answer, you must cycle through each occurrence of a particular word in the set of speeches, in order by speech and word. Thus, if the word, “I” appears five times in the song and appears in the following four locations: (1) speech 2, word 7, (2) speech 4, word 12, (3) speech 4, word 50, and (4) speech 5, word 1, then the first substitution would come from speech 2, the second substitution would be the 12th word of speech 4, the third substitution would be the 50th word of speech 4, the fourth substitution would be from speech 5 and the last substitution would be from speech 2 again. As this example illustrates, a word does not have to appear in speeches as many times as it appears in a song. It does, however, have to appear at least once. Note also that a word may appear several times in a speech and these occurrences are used in order and before using the first occurrence of that word in the next speech.

The Input:

The first input line contains a single integer, *n* $(1 \le n \le 20)$, indicating the number of spoofs to potentially create. Each of the *n* input sets follows. The first line of each input set contains an integer, *m* $(1 \le m \le 100)$, representing the number of speeches for which there is text. Each of the speeches follows. Each speech is stored on a single line. The first token on each of these lines will be an integer, *t* $(1 \le t \le 1000)$, representing the number of words in the speech. Each of these words will consist of 1-20 lowercase letters. Words are separated by spaces and there are no other characters on these lines.

The input speeches are followed by a single line storing the lyrics of the song to be spoofed. The first token of this line will be an integer, *l* $(1 \le l \le 50)$, indicating the number of words in the lyrics. (Most songs aren't so long!) Each of these words will consist of 1-20 lowercase letters. The words in the lyrics are separated by spaces and there are no other characters on this line.

The Output:

For each spoof, first output "`Spoof #i: `" where *`i`* is the spoof number, starting with 1. If no spoof is possible, then simply output the string "NOT POSSIBLE".

If a spoof is possible, starting on the next line, output *w* lines, where *w* is the number of words in the song to spoof. Each line will contain two integers (separated by one space): the speech number and the word number, both of which are numbered, starting at 1.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
2
3
8 tom brady threw a pass in the pocket
5 i sung very well today
8 the genie wishes i had one wish left
7 i threw a wish in the well
1
15 let it be told to the future world that we came forth to meet it
8 do not ask me i will never tell
```

Sample Output:

```
Spoof #1:
2 1
1 3
1 4
3 7
1 6
1 7
2 4

Spoof #2: NOT POSSIBLE
```

UCF Local Contest — September 1, 2012

Spaces, the Final Frontier

filename: `spaces`

Captain Pick-Hard, the commanding officer of the starship *Venture,* is also a skilled xenoarchaeologist. On his few vacations, he can generally be found on some obscure planet digging up artifacts from long dead civilizations. On his last expedition, he found a series of inscriptions in the ancient language of Zagziggian. He requires your help in deciphering them.

A strange feature of Zagziggian writing is that it doesn't use spaces between words. Consequently, deciphering it is generally quite difficult, as one can never be sure when one word ends and the next word begins. This makes division of an inscription into its component words a difficult problem, because there can be a very large number of possible divisions.

Fortunately, the ever helpful Commander Theta has successfully reactivated the Zagziggian Codex, an artifact that, given an inscription, assigns a numerical value to a set of Zagziggian words. After studying these word-value pairs, Captain Pick-Hard hypothesizes that the *score* of a division is the sum of the values of its component words. And furthermore, inscriptions are written in such a way that the correct division is the one with the highest score. Note that the same word may occur multiple times in an inscription, and each occurrence contributes to the score. So if the word *ra* has a score of 5, and shows up 3 times, that counts as 5*3 = 15.

The Problem:

The captain wants a program that will find the highest scoring division of each inscription. Helping the captain is a step towards the UCF Programming Team and all the glory that goes with it, so let's help the captain.

The Input:

The first input line contains a positive integer, *n*, indicating the number of inscriptions to divide. This is followed by *n* descriptions of an inscription and the associated Codex.

The first line of each description is a single string *S* with length between 1 and 1000, inclusive. This string is guaranteed to contain only lowercase letters with no spaces.

The next input line contains an integer, c $(1 \leq c \leq 1000)$, indicating the number of word-value pairs generated by the Codex for this inscription. This is followed by *c* lines, each of which contains a word *w* (all lowercase letters with length between 1 and 1000, inclusive) and its value v $(1 \leq v \leq 1000)$, separated by a single space.

The Output:

For each of the descriptions in the input, output "`Inscription #k:`" where *k* is the number of the inscription. The next output line should contain the highest possible score for dividing that inscription. On the next line, you should print the highest scoring division by inserting spaces into the inscription string at the appropriate spots. You may assume that at least one valid division always exists.

In case there are multiple possible divisions with the highest score, choose the one where the first *differing* space is placed earlier in the string. (A differing space is one that is not present in both divisions.) See the Sample Output for any clarifications.

Leave a blank line after the output for each description. Follow the format illustrated in Sample Output.

Sample Input:

```
2
helloworld
7
he 5
hello 9
llo 2
lo 3
wo 6
wor 4
world 8
atatata
5
a 7
at 3
ta 6
t 4
att 5
```

Sample Output:

```
Inscription #1:
17
hello world

Inscription #2:
40
a t a t a t a
```

UCF Local Contest — September 1, 2012

Rummy Score

filename: `rummy`

Mack and Zack (Dr. Orooji's twins) play Rummy with a modified deck of playing cards. The card values are 1 through 13 (1 does not serve the dual purpose of 1 and 14; it is only 1) and there are no suits. Also, a deck may contain any number of each card value (unlike a standard deck of cards which has exactly four of each).

The Problem:

Given the seven cards in a Rummy hand (held by Mack or Zack), you are to determine the points lost by the hand. In Rummy, you group three (or more) cards together if they have the same value or they have consecutive values. Examples of a group include {4, 4, 4}, {6, 6, 6, 6, 6}, {9, 10, 11}, and {1, 2, 3, 4}. The "points lost by a hand" is the sum of values for cards that are not in a group.

When computing the points lost by a hand, you group the cards to minimize the loss, i.e., you want the smallest total sum of values for the ungrouped cards. Note that a given card could possibly belong to more than one group; you need to pick the group that minimizes the total points lost. For example, given the seven cards {2, 2, 2, 3, 4, 8, 10}, the grouping {2, 3, 4} is better than the grouping {2, 2, 2}. Note also that a card can be used in only one group, e.g., from the Rummy hand {2, 2, 2, 3, 4, 8, 10}, we can not create the two groups {2, 2, 2} and {2, 3, 4} since there are not a total of four cards with a value of 2 in the hand.

The Input:

The first input line contains only a positive integer, *n*, indicating the number of Rummy hands. The hands are on the following *n* input lines, one set per line. Each set contains seven integers, each integer between 1 and 13, inclusive (the values are separated by a single space).

The Output:

At the beginning of each test case, output "`Rummy Hand: `*`v`*" where *`v`* is the input values. Then, on the next output line, print the points lost by the hand. Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
7
5 2 2 6 2 10 4
11 11 11 11 1 2 3
11 2 3 4 5 13 1
4 5 6 6 6 6 13
2 2 2 3 4 8 10
12 13 1 4 4 4 4
10 10 10 8 9 8 9
```

Sample Output:

```
Rummy Hand: 5 2 2 6 2 10 4
10

Rummy Hand: 11 11 11 11 1 2 3
0

Rummy Hand: 11 2 3 4 5 13 1
24

Rummy Hand: 4 5 6 6 6 6 13
13

Rummy Hand: 2 2 2 3 4 8 10
22

Rummy Hand: 12 13 1 4 4 4 4
26

Rummy Hand: 10 10 10 8 9 8 9
10
```

UCF Local Contest — September 1, 2012

Simi Circles

filename: `circles`

Arup has a step-daughter, Simi, at home, so he's frequently in charge when she's swimming with her friends. Unfortunately, after she's done, Simi will run inside the house without completely drying herself. As she scurries through the house to the bathroom, she leaves drops of water that fall on the floor. Naturally, Arup's wife, Anita, blames him for not properly watching the kids and making sure they properly dry themselves before entering the house. Anita will say that the wood was ruined by the water damage. But, of course, Arup realizes that not all of the wood is ruined. Thus, he only wants to take responsibility for the area of the wood floor that has water on it.

When Simi runs through the house, each drop of water hits the ground and forms a perfect circle. Furthermore, she runs fast enough that any drop at most intersects with the previous drop and no others. Sometimes a drop does not intersect with the previous drop. This means that any individual drop can at most intersect with two drops: the one before it and the one after it. Assume that each drop (circle) covers some area of the wood floor. In particular, assume that a circle will not be completely encompassed (covered) by another circle.

The Problem:

Given a list of circles, where a circle in the list may only intersect the previous circle on the list and the subsequent circle on the list, determine the total area that the circles cover.

The Input:

The first input line contains a single positive integer, n $(1 \le n \le 100)$, indicating the number of Simi circle scenarios to consider. Each of the n input sets follows. The first line of each input set contains only an integer, c $(1 \le c \le 100)$, representing the number of drops for the input set. The following c lines contain information about each drop, one drop per line, in the order that the drops occurred. Each of these lines contains exactly three real numbers (each separated from the next by a single space): x $(-3000 < x < 3000)$, y $(-3000 < y < 3000)$, and r $(0 < r < 10)$, representing (respectively) the x and y coordinates of the center of the drop (in cm) and the radius of the drop (in cm).

The Output:

For each Simi circle scenario, first output "`Set #i: `" where `i` is the input set number, starting with 1. Follow this with a single positive real number rounded to 2 decimal places, representing the total area (in cm^2) covered by all of the drops for the Simi circle scenario. Thus, if the actual area is 31.674 cm^2, output 31.67 and if the actual area is 31.675 cm^2, output 31.68.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
3
2
0 0 1
0 2 1
2
0 0 5
6 0 5
1
0.1 0.1 0.1
```

Sample Output:

```
Set #1: 6.28

Set #2: 134.71

Set #3: 0.03
```

Note: If you need to use pi in your program, use the value 3.1415926535898.

UCF Local Contest — September 1, 2012

Dungeon Trouble!

filename: `dungeon`

Ash and Hazel used to play a game called “Dungeon Trouble!” when they were young. The mechanics of the game are quite simple. You start with a map of unnumbered dungeons in a cave, some of which are connected to one another via secret passage. The goal is to assign a number between 1 and 5, inclusive, to each dungeon so that no two adjacent dungeons share the same number. The last person to be able to successfully number a dungeon without violating the rule is declared to be the winner of that game.

Ash loves statistics and kept meticulous records of every game played. For each game played, he would write down a graph describing the connection of dungeons in the cave and the number that got assigned to each dungeon. Note that some dungeons may not have gotten numbered due to one player winning before they were all numbered.

Recently, Ash input all of these old games into his super awesome computer, “The Prime Analyzer 9000”. Unfortunately, his computer was too specialized and only recognized the prime numbers he input for dungeon numbers; all non-prime dungeon numbers were lost! (For some strange reason, the computer only lost the non-prime dungeon number assignments. All other prime input was preserved as Ash originally input it.)

The Problem:

Ash wants your help to analyze the games to see what information can be gleaned from the data that is left. You still have a perfect mapping of the original dungeon and connections, and the dungeon numbers that were assigned 2, 3, or 5 (the primes). However, you do not know the dungeon numbers of any dungeon which originally had 1 or 4 assigned, nor do you know which dungeons were not originally assigned a number.

Ash wants you to analyze each game and tell him whether the game could possibly have been played to “completion” or not. A game is considered played to “completion” if all dungeons are assigned a number between 1 and 5. Note that a player may “win” a game before all dungeons are assigned a number but we consider a game is played to “completion” if all dungeons are assigned a number. Ash doesn’t care who won; he just wants to know whether he and his sister were able to play the game to completion, assuming they both played optimally and each had the goal of completing the game. Note that, for the purpose of this program, the goal is not to win the game; rather to play the game to completion.

The Input:

The first input line contains a positive integer, *t*, indicating the number of dungeons to process. Each dungeon begins with a line containing two positive integers, *n* ($1 \le n \le 26$), which is the number of dungeons in the map, and *m* ($1 \le m \le 325$), which is the number of secret passageways connecting the dungeons. This will be followed by *m* lines, giving the connections between the dungeons. Each line will be of the form *i j* (‘A’ ≤ *i* < *j* ≤ ‘Z’), indicating a secret passage (connection) between dungeon *i* and *j* (secret passages can be traversed in either direction). Following this will be a line containing an integer *k* (0 ≤ *k* ≤ *n*), giving how many dungeons already have numbers. This will be followed by *k* lines of the form *i c* (‘A’ ≤ *i* ≤ ‘Z’ and *c* = {2,3 or 5}). It is guaranteed that each of these *k* lines will give a unique dungeon which exist in the given map, and the existing dungeon numbering will not violate any rules of the game. It is also guaranteed that there is a path between any two dungeons in the map, i.e., one can go from any dungeon to any other dungeon using the passages (connections) in the cave.

The Output:

For each dungeon in the input, output a line “`Dungeon #`*`x`*`: `*`s`*” where *x* is the dungeon number, starting with 1, and *s* is either the message “`Ash and Hazel played this game to completion`” or “`Ash and Hazel did not complete this game`” for the two possible outcomes. Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
3
3 3
A B
A C
B C
1
A 3
3 3
A B
A C
B C
0
4 5
A B
A C
B C
B D
C D
2
A 5
D 2
```

Sample Output:

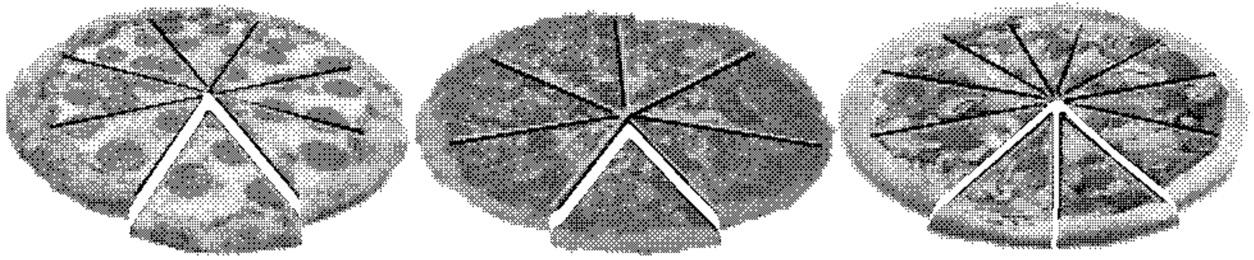
```
Dungeon #1: Ash and Hazel played this game to completion

Dungeon #2: Ash and Hazel did not complete this game

Dungeon #3: Ash and Hazel played this game to completion
```

UCF Local Contest — September 1, 2012

Safely Stacking Leftover Pizzas

filename: `safepizza`

Rafael and his roommates love pizza, but they are on a budget, so they have decided to eat the same exact amount of pizza every day for the semester — up to 100 days in a row. Starting on the first day of the semester, they will order a few different kinds of cheap-but-tasty pizza from Ultracheap Pizza Express — perhaps a Pepperoni, a Sausage & Mushroom, and a Veggie. The store is too cheap to slice the pizzas, so Rafael will slice each one based on their plan: maybe cutting the Pepperoni pie into 8 slices, the Sausage into 7, and the Veggie into 11. Then they will enjoy some fresh hot slices — altogether consuming, for example: 1 Pepperoni, 1 Sausage, and 2 Veggie slices. Finally they will stack the leftover pizzas, in their original boxes, inside their cheap-but-cold refrigerator, which has no shelves. The tiny refrigerator is big enough for only one stack of pizzas and not several separate stacks.



On each day thereafter, they will pull out the entire stack from the refrigerator, open the boxes, and eat the *exact same number* of slices from each pizza that they had decided on the first day — either cold, or nuked in their cheap-but-warm microwave. After the slices are eaten each day, any leftover pizzas are placed back in the refrigerator, and stacked in the same exact order as they were before. Why this meticulous stacking? Ultracheap doesn't label the boxes, so keeping consistent order is the only way to tell the pizzas apart without constantly re-opening them.

When a pizza is nearly finished, they will order a replacement for it that arrives just in time such that the amount of each type of pizza consumed each day is always exactly the same. The new, whole pizzas only show up right before the slices are consumed for the day, so there is never a "leftover" *whole* pizza in the refrigerator and there is never insufficient pizza. Empty boxes are not leftovers either — they go in the trash. But even though sometimes there are no leftovers of a particular kind of pizza, the remaining pizzas will still be stacked in the same *relative* order as on the first day. For example, if the stacking order is Veggie on top, then Pepperoni, then Sausage on the bottom, and the Pepperoni box is empty when it's time to put away the leftovers, then the Veggie box will be stacked directly onto the Sausage box. On the next day when a new box of Pepperoni arrives, the three boxes will again maintain the first-day ordering.

One problem with stacking: well, those cardboard boxes are ultra-cheap. Nothing ruins a pizza like having its cardboard box collapse onto it. Cardboard-flavored mozzarella...Yecch!

Rafael came up with the bright idea to stack the boxes lightest to heaviest, with the heaviest leftover pie on the bottom, to reduce the chance of one pizza crushing another. Since all the pizzas weigh the same initially, he can predict the weight of the leftovers by calculating the fraction of the pie that remains in the box. Given the example above, on the first day there will be the following leftovers: ${}^{7}/_{8}$ Pepperoni, ${}^{6}/_{7}$ Sausage, and ${}^{9}/_{11}$ Veggie, so they should be stacked with Pepperoni on the bottom, then Sausage, with Veggie on top.

Roommate Robin points out that Rafael's plan works fine on the first day, but different pizzas get consumed at different rates. So by always stacking leftover pizzas in the same order, it is possible for a heavier pizza to be stacked on top of a lighter one, risking a ruined pie. Continuing the previous example, on the 6th day, the following leftovers would need to be stacked: ${}^{2}/_{8}$ Pepperoni, ${}^{1}/_{7}$ Sausage, and ${}^{10}/_{11}$ Veggie (having eaten 1 leftover slice and 1 fresh slice to get the required 2 slices). The Pepperoni leftovers are properly under the lighter Sausage, but the Veggie pie is on top, even though it's now the heaviest! This puts the Sausage leftovers at risk.

The Problem:

Given the list of pizzas ordered and the rates at which they are consumed, determine a stacking order for the leftover pizzas that can be used every day but will minimize the chance of a cardboard cheese catastrophe.

The Input:

The first line of input consists of a positive integer, N, indicating the number of semester plans to evaluate. Each semester plan starts with a line that contains an integer P ($2 \leq P \leq 5$), which is the total number of pizzas that will be ordered the first day, followed by a space, followed by an integer D ($1 \leq D \leq 100$), which is the duration of the plan in days, since Rafael might give up on pizza and go on a Paleo diet. The next $2 \times P$ lines will describe each pizza, in exactly 2 lines each.

The first line of each pizza description will consist of 1 to 20 lower-case letters naming the type of pizza. Names will be unique within each semester plan. The second line will consist of exactly two integers E ($1 \leq E < 20$) and S ($E < S \leq 20$), separated by a space, where E is the number of slices eaten from that type of pizza each day, and S is the total number of slices in a whole pizza of that type. There are no blank lines or extra spaces in the input.

The Output:

For each semester plan in the input, output a line "`Leftover Pizza Stacking Order for Semester` *`M`*`:`" where M is the number of the semester plan in the input, counting from 1. Starting on the next output line, print the P names of the pizza types for that semester plan, one per line, in order that they should be stacked, from *top* to *bottom*. The stacking order must be the safest possible one — that is, it must minimize the number of instances over D days in which a heavier pizza is stacked *immediately above* a lighter pizza; note that multiple such instances are possible in the same stack on the same day and each instance counts towards the total; also note that the stacking after the final day must be checked as well and counted towards the total.

If there is more than one safest stacking, output the order that would be first alphabetically, when comparing pizza names from top to bottom of the stacks. Leave one blank line after the output for each semester plan. Follow the format illustrated in Sample Output.
(Sample Input/Output on the next page)

Sample Input:

```
3
3 6
pepperoni
1 8
sausagemushroom
1 7
veggie
2 11
3 96
buffalochicken
1 8
chorizo
1 2
anchovy
1 4
3 3
hawaiian
1 5
feta
1 2
spinnocoli
2 3
```

Sample Output:

```
Leftover Pizza Stacking Order for Semester 1:
veggie
sausagemushroom
pepperoni

Leftover Pizza Stacking Order for Semester 2:
anchovy
buffalochicken
chorizo

Leftover Pizza Stacking Order for Semester 3:
hawaiian
spinoccoli
feta
```