Twentieth Annual
University of Central Florida

High School Programming Tournament

Problems

Problem Name	Filename
Phish Phinder	phishy
Skippy and His New Jetski	jetski
Browser Test	browser
Decisions, Decisions...	decisions
Ali's Short Bus	bus
Square in the…	square
Kakuro Konundrum	kakuro
Lice-N-Plates…Are Back!	lice
Capture The Scene!	picture
The Neptune Ordeal	neptune
Su-Do-Kode	sudokode

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

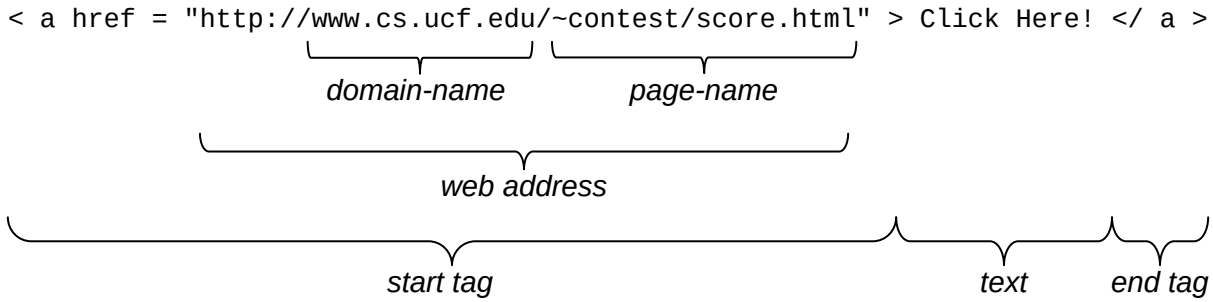
For example, if you are solving The Neptune Ordeal:
Call your program file: neptune.c, neptune.cpp or neptune.java
Call your input file: neptune.in

Phish Phinder

Filename: `phishy`

E-mail is a great technology, but like any technology it can be abused. "Phishing" is when criminals use e-mail to try to obtain financial or personal information. A typical phishing e-mail will try to look like it is from a bank or online payment service. It will ask the recipient to click on an HTML link in the e-mail, and log in to their web account. In many cases, the link really looks like it should go to the bank, but it has been rigged to go to a fraudulent web site instead. When users click the link and then type in their passwords and other personal information, it all goes to the criminals instead of the bank.

You don't need to know all of HTML to understand how links can be rigged for phishing, just the part that is used for a link. Here is a sample link, with its parts explained:



`< a … >` `</ a >`	If these (*start tag* and *end tag*) appear in an e-mail message, they will do so only in pairs as shown above. White space may be present in each tag anywhere it is shown above. The '`a`' and '`href`' will always be lower case, with white space between them. The slash ('`/`') in the end tag will always immediately follow the '`<`' of the tag.
`http://`*domain-name*`/`*page-name*	This is a *web address* (or *URL*), and in a link it is the only thing between the double quotes ('`"`') in the start tag. A web address never contains white space. The '`http://`' part is always present and in lower case. The *domain-name* will always be present. The slash ('`/`') and *page-name* after the *domain-name* each may or may not be present; however if the *page-name* is present then the slash will always be there.
domain-name	A *domain-name* consists of lower-case letters and at least one dot ('`.`'). Each dot will have at least one letter immediately preceding it and at least one letter immediately after it.
page-name	A *page-name* may be composed of any printable characters except double quotes ('`"`'), white space, and angle-brackets ('`<`' and '`>`').
text	Any white space and any printable characters, other than angle-brackets ('`<`' and '`>`'), may occur in the *text* between the start tag and end tag. At least one character will be present in the text part of each link.

Where *white space* is allowed in a link—or anywhere in an e-mail message—it can be spaces, line breaks, or any combination of the two (no tabs). *Printable characters* excludes all control characters, such as ESC, and characters with ASCII values greater than 126.

The Problem:

Your program will scan e-mail messages to find suspicious links and warn the recipient about the possible phishing attempt.

To find a phishing e-mail, your program should examine the text of each link in the message body of the e-mail. Not counting any white space at the beginning and end of this text, if the text contains only a valid domain-name (as described above), then this is one kind of link that might be rigged for phishing, and the text must be compared to the web address in the start tag.

When such a link is rigged for phishing, the domain-name in the text will not match the domain-name in the web address of the start tag.

Domain-names are considered to match as follows: Working from *right* to *left* in each domain-name, the characters must be identical, until reaching, in both, either the second dot or the beginning of the domain-name, whichever comes first in each.

For example, http://acm.org matches www.acm.org and register.ucf.edu matches http://www.contest.cs.ucf.edu/index.htm. However, go.ogle.com does not match www.google.com, because we haven't reached the second dot of www.google.com when we run out of identical characters at the second dot of go.ogle.com. The arrows show the match order in each example above.

The Input:

The first line of input contains only a single integer, *n*, from 1 to 200, which is the number of e-mail messages to scan. The messages will start on the second line of input. The first line of each message starts in column 1, is at most 70 printable characters, and is of the form

`From: sender`

where sender is all the characters after the space. There is no other white space on that line. The second line of each e-mail message will start in column 1 and is of the form

`X-Lines: numLines`

where *numLines* is a number from 1 to 10 inclusive, immediately following the space. The message body of the e-mail will start on the third line and occupy exactly *numLines* lines. No line in the message body will start with "`From`" or "`X-Lines`", and each will have 70 or fewer characters.

Each message body may contain white space, printable characters, and/or HTML links, as described above. No message will contain angle-brackets ('<' and '>') except in links.

The Output:

For each e-mail message in the input, if the message body contains at least one link that could be rigged for phishing as described above, output the warning

`Mail from` *`sender`* `looks phishy!`

Otherwise output

`Mail from` *`sender`* `looks ok.`

Sample Input:

```
3
From: account-info@zitibank.com
X-Lines: 4
Warning!!!!!!! You're bank acount has been frozen due to supecious
activitys. Please login immediatly at <a href =
"http://pesce.penne.pasta/zitibank/login.html">www.zitibank.com </a>
to reactivate you're bank account inclding you're ATM card(s).
From: Velma@mysterymachine.van
X-Lines: 5
heyyyyy,'sup.... yesterday my pal scooby introduced me to a guy named
hrenry hrefner, a =rich= and "famous" man who has greater-than *three*
motorcycles even tho he’s older than...um, older than ghost stories.

hm, i wonder if he likes g.w.g (girls with glasses).
From: security_dept@payfoul.com
X-Lines: 8
Your account has been locked because, well, we felt like it. In order
to send or receive any more money via PayFoul, you must go to <a href=
"http://login.cgi.payfoul.com/update/info/why?waste=time&yr=20">
www.payfoul.com</a> and enter another bank account number, two more
credit cards, your driver's license, and a scan of your fingerprints.
You agreed to this when you signed up, so don't complain about it! Go
review your <a href ="http://tos.payfoul.com/we-own-you.html">terms of
service</a>.
```

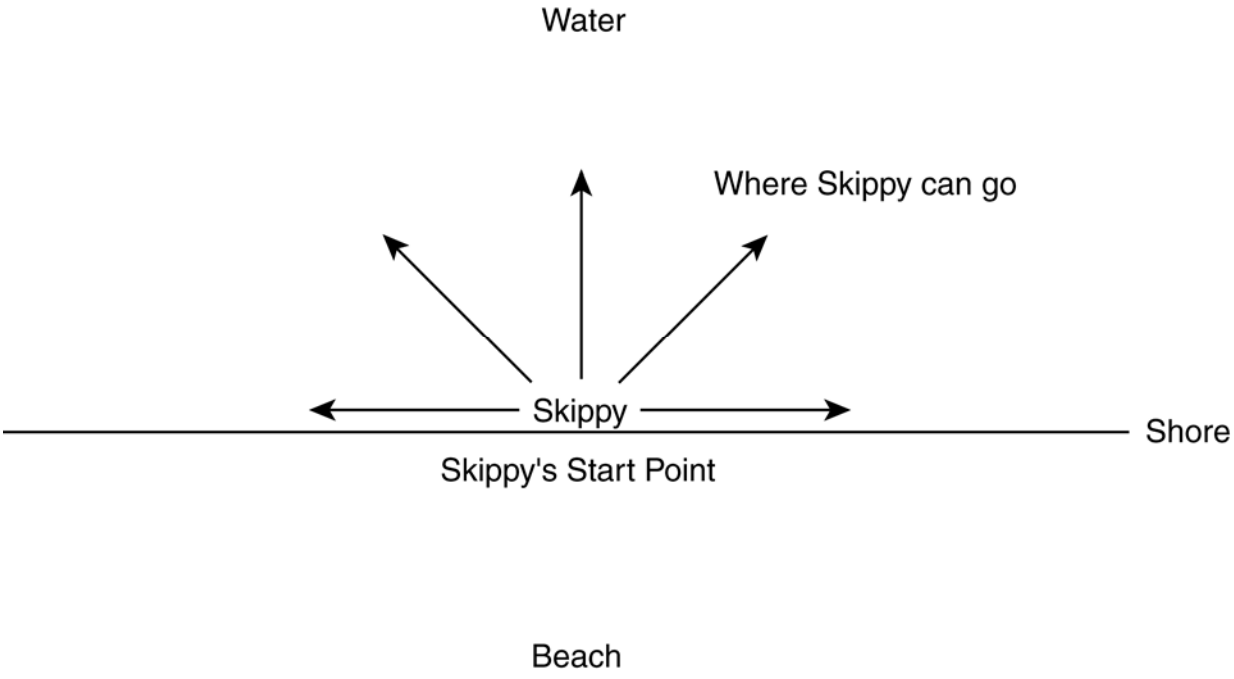
Sample Output:

```
Mail from account-info@zitibank.com looks phishy!
Mail from Velma@mysterymachine.van looks ok.
Mail from security_dept@payfoul.com looks ok.
```

Skippy and His New Jetski

Filename: `jetski`

After a somewhat lengthy disappearance, Springfield Skippy has made his presence known on the beach with his new jetski. The problem is that Skippy is quite reckless on his jetski, so no one else in the water is safe when he's in the area. To deal with this problem, the beach patrol would like to put up warning signs in the areas where Skippy could potentially be riding his jetski.



The Problem:

Your goal will be to help the beach patrol with their task by figuring out the total area where Skippy could be after a given time period given information about the speed of his jetski and the amount of time he will be on his jetski.

Assume that coastline at the beach is perfectly straight and can be modeled by the x-axis in the coordinate plane. Thus, all the points with a negative y-value are part of the beach, and all the points with a positive y-value are in the water. Skippy always starts at the point (0, 0) and is liable to go in any direction in the water from that point. Assume that Skippy's speed on the jetski is constant. So his jetski travels exactly the same speed regardless of whether he's headed up shore, down shore, or anywhere in between. Let *S* be the semi-circular zone where Skippy could possibly reach in the time he jetskis. Your goal is to determine the area of *S*. For your calculation, assume π is equal to 3.1415926535898.

The Input:

Since Skippy likes his jetski, he will spend several days at the beach. The first line will contain a single positive integer, *n* (*n* < 100), describing the number of days Skippy ventures into the ocean. The next *n* lines will contain two positive integers each. The first positive integer, *s*, on each of these lines, represents the speed of Skippy's jetski in miles per hour. The second positive integer, *m*, on each of these lines, represents the number of minutes Skippy will ride his jetski.

The Output:

For each day at the beach, you will output a single line header of the following format:

`Day k:`

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

Follow this with a space, and then the area in square miles rounded to the nearest hundredth (.005 rounds up), where Skippy could potentially be for that day at the beach.

Sample Input:

```
2
30 60
5 120
```

Sample Output:

```
Day 1: 1413.72
Day 2: 157.08
```

Browser Test

Filename: `browser`

The makers of the popular open-source Web browser, Icewolf, are nearing their next release. They've fixed some pretty basic problems, and are nearing completion. However, they're running some automated tests on their browser, and they need your help to write a program to verify their tests.

The Problem:

Given a list of URLs and some basic commands (`BACK`, `FORWARD`, and `RELOAD`), output the websites that the browser should be visiting in the order that it visits them.

Icewolf has six commands:

> `START`: Starts the browser session.
>
> `END`: Ends the browser session.
>
> `URL` *`string`*: Indicates that the browser should load the URL given by *string*. The length of *string* will be between 2 and 100, inclusive, and will not contain any whitespace characters.
>
> `BACK`: Indicates that the browser should go back to the previous URL. If no prior URL is present, this command will do nothing. Repeated `BACK` commands will travel farther back along visited URL's.
>
> `FORWARD`: Indicates that the browser should go forward to a URL from which it had previously gone "back". If there is no such URL, this command will do nothing.
>
> `RELOAD`: The browser should reload the current URL. If there is no URL to reload, this command will do nothing.

Uses of the `RELOAD` command do not affect the `BACK` or `FORWARD` commands. Use of the `URL` command clears the `FORWARD` list. Icewolf has a limit of 100 URLs that it can store at any given time. The commands between each `START` and `END` will never cause Icewolf to exceed the limit of URLs it can store. There will be no `START` statements except directly after an `END` command, or at the beginning of the file.

The Input:

The input will consist of a series of these commands, one command per line. Each browser session will begin with a `START` command and end with an `END` command. End of input will be signaled by an `END` command directly after the `END` command from a browser session. All URLs will be valid (no whitespace characters). The first line in the file is guaranteed to be `START`, and all `END` commands (except the last two) will be followed by a `START` command.

The Output:

For each brower session in the input, output a single line containing the message "`Browser Session #n:`", where *n* is the number of the session, starting with 1. Then, for each `URL`, `BACK`, `FORWARD`, and `RELOAD` command in the input, output the URL that is loaded by the browser, if any, due to that command. Output each URL on a separate line, as shown in the Sample Output. Leave a blank line after the output of each browser session.

Sample Input:

```
START
URL http://www.google.com
URL http://www.yahoo.com
URL http://www.icewolf.com
BACK
BACK
FORWARD
RELOAD
RELOAD
END
START
URL http://www.ucf.edu
URL http://www.acm.org
BACK
BACK
BACK
FORWARD
RELOAD
END
END
```

Sample Output:

```
Browser Session #1:
http://www.google.com
http://www.yahoo.com
http://www.icewolf.com
http://www.yahoo.com
http://www.google.com
http://www.yahoo.com
http://www.yahoo.com
http://www.yahoo.com

Browser Session #2:
http://www.ucf.edu
http://www.acm.org
http://www.ucf.edu
http://www.acm.org
http://www.acm.org
```

Decisions, Decisions...

Filename: `decisions`

The UCF programming team at the ICPC finals this year in San Antonio, TX is headed back to the hotel from dinner, and there are many possible routes. Nobody can decide which way to go; finally, Adam gets fed up, pulls out a coin, and flips it, and uses the result to decide. This works so well that the team uses this method for every decision on the way back. There are a number of interesting sights along the way so the team needs your help!

The Problem:

Your task is to write a program to calculate the chances of passing through certain locations.

Each location is represented by a positive integer identifier less than 1000, except for the hotel, which uses -1 as its identifier. From any defined location other than the hotel, there are two routes; each route leads to another location.

The Input:

There will be several scenarios. Each scenario will begin with a line containing two integers indicating which locations the team could go to next from the restaurant. Following this line will be a list of lines defining locations. Each line will contain three integers: the first will specify the location being defined, and the other two will specify where the team might go from there. A destination of -1 indicates the hotel, where the team will go to sleep. No location will be referenced after it is defined in the input (i.e. once a location is defined, no future routes will lead to it), or defined more than once. A line containing only three -1 values defines the hotel and ends the sequence of location definitions. Note that the restaurant at which the team begins is not a defined location.

After the location definitions, a number of queries appear, one per line. Each query consists of an integer defining a location ID being queried; no location will be queried more than once but unknown queries could exist (the team may want to know if it can get to a particular location). The final query within each scenario will always be for the hotel, since the team definitely needs to know whether they arrive there sooner or later! After the final data set, a line containing only two -1's will signify the end of input.

The Output:

For each scenario, display a header line with the data set number (these are sequential and begin at 1), formatted as "`Scenario #`*`n`*`:`". Then, for every query, output a line with the queried location number and the chance of reaching that location, expressed as a percentage to two decimal places of precision, rounded, like this: "`Location` *`l`*`: `*`p`*` %`". However, for the final query, simply use "`Hotel`" as the location identifier: "`Hotel: `*`p`*` %`". Print a blank line after every data set's output.

Sample Input:

```
1 2
1 -1 -1
2 -1 -1
-1 -1 -1
1
2
3
-1
1 2
1 -1 -1
2 -1 -1
-1 -1 -1
1
-1
-1 -1
```

Sample Output:

```
Scenario #1:
Location 1: 50.00 %
Location 2: 50.00 %
Location 3: 0.00 %
Hotel: 100.00 %

Scenario #2:
Location 1: 50.00 %
Hotel: 100.00 %
```

Ali’s Short Bus

Filename: `bus`

Rumor has it that there exists a school bus driver who likes to drive a little bit too fast. Some say that he goes by the name of Ali. What makes Ali so different from your normal run of the mill bus driver you ask? Well, as the legend says, people who get on this bus usually end up riding the short bus. Sound baffling? I said the same thing until I boarded this speed demon, and now everyone calls me crazy! Would you take a minute to hear my story?

I remember when I first set eyes on the yellow devil. It was the first day of school back a few years ago. Like any other day, everyone was anxious to meet new people and figure out which classes they were going to complain about all year. Nothing seemed peculiar on that day. In fact, the bus looked like a plain, dirty, yellow school bus. After I got on the bus, Ali turned and said, "Hello, everybody! How are you doing?" It was rather pleasant in one of those deja vu kind of ways. As I took my seat, I noticed that the bus was right on time, which is odd for the first day of school. Ali must be one of the punctual types.

Anyway, as the bus pulled away, I noticed that we kept going faster and faster. In fact we were going so fast that I could've sworn the bus shrank! Yes, the bus shrank! I didn't notice this myself, but I happened to notice that everything else was getting shorter, so I can only assume that the actual bus I was on was getting shorter as well. I told the girl sitting in the seat next to me, and she just said, "You deserve to ride a short bus with comments like that." I'm still not sure what that comment meant, maybe Ali knows?

So when I got to school, I found the principal and told him my story. He called the school counselor immediately. Between the two of them asking questions like, "Why was your bus so late?" and "Why are you lying to us about the time it arrived?", I thought I was going to go crazy. It was as if time slowed down for us. I told them that Ali's bus was on time and traveled the usual route, and they didn't believe me! After sitting in the office for half an hour explaining my story, I realized that I was going to end up in the looney bin.

Now I ride a smaller bus each day- not to computer classes at school, but to group therapy. But I am not crazy! I have proof! Yes, proof! I just have no way to test it with a computer... But I see that you have one right there in front of you. Please, help me.

I determined that the bus gets shorter the faster that it goes. I also noticed that time seemed to speed up for everyone not on the bus, which is weird! I'm not sure about the time thing yet, so let's ignore that part for now. As for speed, luckily, I noticed the relationship between speed and length is linear.

So, if I give you two measurements, I need to know how fast Ali must go to make his bus as short as the ones I have to ride now. Yes, that is it! I just need you to ride that bus for me and tell me that key speed! I'm not crazy. You'll see!

The Problem:

As Ali's school bus travels at a given speed, I will provide you the length of the bus. I've only got enough time to give you two measurements, but rest assured, the bus shrinks in a linear manner. I will also provide you with the length of the short bus that comes to pick me up these days. Furthermore, if the bus moves faster, it must be getting shorter. You will always be able to determine the ratio of speed to length. All you have to do is tell me how fast Ali must drive his mystic bus in order to make it a short bus. Please, do this for me, and I'll give you a balloon!

The Input:

The problem will have multiple bus trips. For each trip, you will be given 5 integers on a single line each separated by a single space, *a*, *v*, *b*, *w*, and *s*. Ali's bus is length *a* meters when it travels at *v* kilometers per hour. Likewise, it is length *b* meters when it travels at *w* kilometers per hour. *s* is the length of my short bus in meters. *a* and *b* will be positive, and *v*, *w* and *s* will be non-negative. Input terminates with a line of five zeroes. This trip should not be processed.

The Output:

For each bus trip, print "`Ali's Bus Trip #`*c*`: `", where *c* represents the current bus trip in sequential order, starting from 1. Following this, on the same line, print, "`The bus must travel at least `*x*` kph.`" where *x* represents the smallest possible non-negative integer speed such that the bus traveling at *x* kph is not longer than my short bus. All final answers will be at most 1,000,000,000.

Sample Input:

```
30 1 20 2 10
45 0 30 100 25
15 15 25 5 10
0 0 0 0 0
```

Sample Output:

```
Ali's Bus Trip #1: The bus must travel at least 3 kph.
Ali's Bus Trip #2: The bus must travel at least 134 kph.
Ali's Bus Trip #3: The bus must travel at least 20 kph.
```

Square in the…

Filename: `square`

Celes and Kefka have a difference of opinion on an important issue. Celes and her friends feel that the world should not be cast into ruin for all time, whereas Kefka takes the opposite stance. The two of them have long since exhausted the possibility of reasoned debate on the matter. They have both agreed to settle their differences with a good old-fashioned game of "Ro-Slam-Beau".

The Problem:

Your goal is to simulate the game of Ro-Slam-Beau. Ordinarily, Ro-Slam-Beau is a game played between two people, but Celes has a large number of friends who also want to participate. On each player's turn he or she strikes one other player. Each of Celes' friends will choose to strike Kefka, and Kefka will choose to strike the friend of Celes who is earliest in the *turn sequence* (as defined below in The Input) and who has not yet been removed from the game. Each player begins the game with a certain number of Hurt Points (HP), which will not exceed 100000. When one player strikes another, the other player is dealt hurt and loses some HP (maximum damage per hit dealt in the game is 9999 HP). When a player's HP are exhausted, he or she falls to the ground and is removed from the game. Play proceeds in a round robin fashion with the first player taking a turn first, then the second player, and so on. The first player doesn't get another turn until all the other players have their turns. The game ends when either Kefka or all of Celes' friends are removed from the game.

The Input:

Input begins with a positive integer, *n*, indicating the number of games of Ro-Slam-Beau to be played. Each of the *n* games begins with an integer, *m*, between 2 and 20, indicating the number of participants in the game. The following *m* lines describe participants in the order in which they take their turns (i.e. the turn sequence). Each line describing a player contains a unique string of 1 to 100 alphanumeric characters indicating the player's name, a positive integer indicating his or her starting number of HP, and a positive integer indicating the amount of hurt he or she will deal when striking another player, each separated by a space. There is guaranteed to be exactly one player named Kefka for each game. All other participants are considered to be friends of Celes.

The Output:

The output for each game should begin with a header line in the following format:

`Game #`*`x`*`:`

where *x* is the game number. Then for each turn taken by a player, print a line indicating the action taken by that player during the turn in the following format:

`PlayerA` `struck` *`PlayerB`* `for` *`y`* `points of hurt.`

where *PlayerA* is the person whose turn it is, *PlayerB* is the person *PlayerA* chose to strike, and *y* is the amount of hurt dealt to *PlayerB*. If *PlayerB*'s HP falls to zero or below, also print a message on a separate line indicating that *PlayerB* has been removed from the game like so:

```
PlayerB has been removed from the game.
```

When the game is over print one of the following messages indicating the outcome:

```
The world has been saved!
```

if Kefka was removed from the game or

```
The world will be cast into perpetual ruin.
```

if Kefka remained standing. The output for different games should be separated by a blank line.

Sample Input:

```
2
5
Celes 9999 3500
Edgar 9999 5000
Terra 9999 4000
Sabin 9999 7000
Kefka 62000 5000
4
Kefka 62000 1250
Relm 1250 417
Strago 1251 305
Gau 1000 512
```

(Sample Output is on next page)

Sample Output:

```
Game #1:
Celes struck Kefka for 3500 points of hurt.
Edgar struck Kefka for 5000 points of hurt.
Terra struck Kefka for 4000 points of hurt.
Sabin struck Kefka for 7000 points of hurt.
Kefka struck Celes for 5000 points of hurt.
Celes struck Kefka for 3500 points of hurt.
Edgar struck Kefka for 5000 points of hurt.
Terra struck Kefka for 4000 points of hurt.
Sabin struck Kefka for 7000 points of hurt.
Kefka struck Celes for 5000 points of hurt.
Celes has been removed from the game.
Edgar struck Kefka for 5000 points of hurt.
Terra struck Kefka for 4000 points of hurt.
Sabin struck Kefka for 7000 points of hurt.
Kefka struck Edgar for 5000 points of hurt.
Edgar struck Kefka for 5000 points of hurt.
Terra struck Kefka for 4000 points of hurt.
Kefka has been removed from the game.
The world has been saved!

Game #2:
Kefka struck Relm for 1250 points of hurt.
Relm has been removed from the game.
Strago struck Kefka for 305 points of hurt.
Gau struck Kefka for 512 points of hurt.
Kefka struck Strago for 1250 points of hurt.
Strago struck Kefka for 305 points of hurt.
Gau struck Kefka for 512 points of hurt.
Kefka struck Strago for 1250 points of hurt.
Strago has been removed from the game.
Gau struck Kefka for 512 points of hurt.
Kefka struck Gau for 1250 points of hurt.
Gau has been removed from the game.
The world will be cast into perpetual ruin.
```

Kakuro Konundrum

Filename: `kakuro`

Much like any pair of brothers, Timmy's older brother picks on him a lot. Recently, Timmy was pushed into a closet with a flashlight, a pen and a puzzle. Timmy's brother will not let him out of the closet until the puzzle is solved. Unfortunately, Timmy has never seen this kind of puzzle before. It requires a set of three positive single digits to sum to a target and each digit must be unique. Timmy needs your help so his brother will let him out!

The Problem:

Given a goal sum and three positive digits, determine if the three positive digits are unique and sum up to the goal.

The Input:

Input will begin with a single, positive integer, *n,* on a line by itself. On the next *n* lines will be a single positive integer representing the goal sum followed by three single positive digits, each separated by a single space.

The Output:

For each line of input, determine if the three digits sum to the goal and are unique. Output "`Proper triplet`" on a line by itself if so, or "`Not a good triplet`" on a line by itself if not.

Sample Input:

```
3
19 4 7 8
10 1 9 6
14 3 8 3
```

Sample Output:

```
Proper triplet
Not a good triplet
Not a good triplet
```

Lice-N-Plates…Are Back!

Filename: `lice`

Two years ago several clever students helped vanquish the lice using clever license plate tracking techniques. However, the lice have regrouped and are back, stronger than ever. They still use license plates, but created a new system of rules by which the license plates are issued. In particular, each louse strongly desires to be unique from the rest and does not want to be mistaken for another louse. Thus, in order for a license plate to be issued, it can not be "too close" to another already existing license plate. Specifically, the "closeness" of two license plates is defined as the *edit distance* between those two license plates. The edit distance between two license plates is defined as the minimum number of changes that must be applied to the first license plate to obtain the second. The types of valid changes are the following:

1. Substitution – changing a character in a single location to another character, such as changing "home" to "hope"

2. Insertion – inserting a character in a single location in a license plate, such as changing "pint" to "print"

3. Deletion – deleting a character in a single location in a license plate, such as changing "whole" to "hole"

The po-lice will not register a license plate that has an edit distance of less than two from a license plate that is already registered. Any application for a valid plate is granted immediately so future applications must not be “too close” to these newly accepted plates either (the po-lice are very efficient that way).

The Problem:

Given a list of existing license plates, which are guaranteed to all be separated by an edit distance of greater than one, and a list of new applications (in the order that they are received), determine which license plates get registered and which ones do not.

The Input:

The first line of input will contain a single positive integer, n, representing the number of registrations. The rest of the file will contain each of the test cases. The first line of each registration case will contain a single positive integer, k ($0 < k < 100$), representing the number of existing license plates. The following k lines will contain one license plate each.

All valid license plates will contain either uppercase letters or digits and be in between 3 and 10 characters long. The next line of the input case will contain a single positive integer, m ($0 < m < 100$), representing the number of applications for new license plates for that test case. The following m lines will contain one application for a license plate each. Each of these applications is guaranteed to have a string of three to ten uppercase letters or digits, inclusive.

The Output:

For each input case, the first line should contain the header:

```
Registration i:
```

where *i* is replaced by a positive integer in between 1 and *n*, inclusive, representing the corresponding input case. The following *m* lines should contain the result of each of the *m* license plate applications for that input case. Each line should have one of the following two formats:

```
PLATE is registered.
PLATE is NOT registered.
```

where *PLATE* is replaced by the corresponding license plate to be. Separate the output for each test case with a blank line.

Sample Input:

```
2
5
ABC123
TYRUQE
937QQQ
UTQABC
999999
4
999998
AC123
HGUTY
AGUTY
2
QWERTY
ASDFGH
3
ZXCVBN
QWER
ASDFGHK
```

Sample Output:

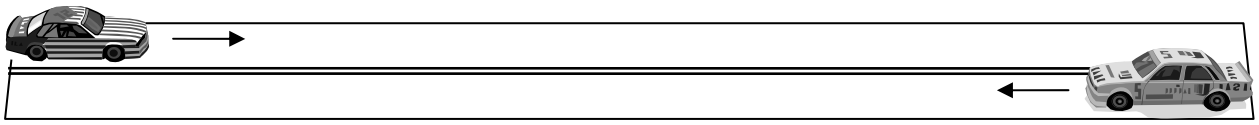
```
Registration 1:
999998 is NOT registered.
AC123 is NOT registered.
HGUTY is registered.
AGUTY is NOT registered.

Registration 2:
ZXCVBN is registered.
QWER is registered.
ASDFGHK is NOT registered.
```

Capture The Scene!

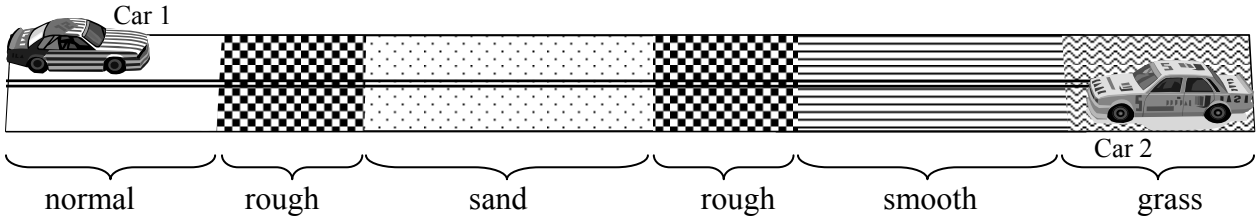
Filename: `picture`

Two cars start at opposite ends of a racetrack, each in its own lane, heading towards each other:



Jason, a professional photographer, would like to take a picture as the two cars pass each other. In order to capture the scene, Jason needs know the exact time this will occur. The track is straight; however, it is composed of different types of terrain, which affects the velocity of the cars.

For example, a racetrack may be composed of the following terrain types:



In this example, Car 1 will start on normal terrain and move through rough, sand, rough again, smooth, and grass terrain. Car 2 will start on grass and move through exactly the opposite sequence. Each terrain type affects the velocity of cars by a constant *terrain factor*. The resulting velocity of a car moving on a terrain is *normal velocity* × *terrain factor*.

Each car has a constant *normal velocity*. If Car 1 has a normal velocity of 80 meters/second, then it will move at this constant velocity on any "normal" terrain. When Car 1 moves onto "rough" terrain, where the terrain factor is 0.75, the resulting velocity of the car will be 80 × 0.75 = 60 meters/second. These cars accelerate instantly; so this same car moving from normal to rough terrain will change from 80 meters/second to 60 meters/second immediately.

The Problem:

Write a computer program to help Jason to determine the time when two cars pass each other, accounting for the terrain differences. For simplicity, your program should not consider the lengths of the cars (it is safe to treat the car lengths as zeros).

The Input:

Jason will be taking pictures of scenes at several different racetracks. Each scene's input consists of 3 parts: *terrain specification*, followed by *racetrack specification* and *car velocities*. The terrain specification begins with an integer n ($1 \le n \le 100$) on one line, the number of terrain types on the racetrack. The following n lines each contain a terrain name and the terrain factor of that terrain, separated by space(s). The terrain names for a given scene will be unique, but different terrains may have the same terrain factor. Terrain names are strings of lowercase letters, with length between 1 and 30, inclusive. Terrain factors are positive real numbers. *If* a terrain with name "`normal`" appears in the terrain specification, its terrain factor is guaranteed to be 1.0.

The racetrack specification for the scene starts on the first line after the terrain specification. The first line of the racetrack specification will be an integer s ($1 \le s \le 100$), representing the number of terrain segments on the racetrack. s lines follow, listing the terrain segments in the order Car 1 encounters them. Each line contains the terrain name and the length of that segment in meters, separated by space(s). The terrain names specified on these s lines are guaranteed to come from the terrain specification of the same scene. Lengths will be positive real numbers.

After racetrack specification, there will be one line containing two positive real numbers separated by space(s). These are, in order, the normal velocities of Car 1 and Car 2, in meters/second.

A single line containing 0 follows the last scenario.

The Output:

For each scene in the input, output the number of the picture, starting with 1, and the time (in seconds) when the two cars pass each other. Round to 3 decimal places (0.0005 rounds up).

(Sample Input and Sample Output on next page)

Sample Input:

```
1
rough 0.75
1
rough 100.0
12.5 37.5
2
normal 1.0
smooth 1.2
3
smooth 100.0
normal 50.0
smooth 100.0
70.0 80.0
6
normal 1.0
rough 0.5
sand 0.3
smooth 2.0
grass 0.8
ice 3.0
6
normal 100
rough 120
sand 320
rough 110
smooth 280
grass 100
80 40
0
```

Sample Output:

```
Picture 1: 2.667
Picture 2: 1.444
Picture 3: 15.764
```

The Neptune Ordeal

Filename: `neptune`

On a New Year's Eve voyage to the Mediterranean, the cruise ship Neptune is hit by a 90-foot tidal wave and turned completely upside down! In a desperate search for survivors, the Coast Guard has managed to receive the Neptune's final distress call at two different listening stations. They know the position where the transmission was received, and the direction from which it was received. From this information, an approximate last known position of the ship can be computed. This will give the Coast Guard a good idea where to start the search. The Coast Guard has contracted you to write a program to triangulate the Neptune's last known position.

The Problem:

Given two unique listening post positions and the corresponding directions to the Neptune's final distress call, triangulate the ship's last known position. The Coast Guard will be sure to give you reports that can be used for triangulation. The directions from the two listening post positions will be different by at least one degree and never by exactly 180 degrees, and they will be convergent (i.e. not parallel).

The Input:

The last rescue of a capsized cruise ship at sea didn't go so well, so to avoid re-making the same mistake, the Coast Guard wants to test your program before putting it into service. There will be multiple simulated rescue missions. Input begins with a single integer, *n*, on a line by itself. On each of the next *n* lines will be six integers *x1*, *y1*, *d1*, *x2*, *y2*, *d2*, where *x1* ($-180 \leq x1 \leq 180$) represents the longitude and *y1* ($-90 \leq y1 \leq 90$) represents the latitude of the position where the first distress call was received, and *d1* ($0 \leq d1 \leq 359$) represents the direction of the call in degrees. The remaining three integers (*x2*, *y2*, *d2*) likewise describe the second distress call. Direction is specified such that 0 is north and 90 is east. To make things simpler, the signals officers have translated the rather confusing spherical coordinates of the Earth to ordinary rectangular coordinates for you, so you can assume the Earth is flat when doing your triangulations.

The Output:

For each rescue, print "`Rescue #`*d*`:`" (*d* beginning with 1 and incrementing) followed by two spaces and "`Neptune's last known position is `*x*`, `*y*" where *x* and *y* represent the longitude and latitude where the ship may be found. Round all coordinates to two decimal places (0.005 rounds up, less rounds down). Leave one blank line after the output for each rescue mission. The last known position will always be within -180 to 180 degrees longitude and -90 to 90 degrees latitude, without wrapping around.

(Sample Input and Sample Output on next page)

Sample Input:

```
2
-50 50 90 -50 60 100
10 0 45 30 0 -45
```

Sample Output:

```
Rescue #1:  Neptune’s last known position is 6.71, 50.00

Rescue #2:  Neptune’s last known position is 20.00, 10.00
```

Su-Do-Kode

Filename: `sudokode`

Dave has become addicted to Sudoku, the latest puzzle craze in all the newspapers and bookstands. In case you don't know, a Sudoku is a simple number puzzle played on a 3x3 grid of 3x3 subgrids. Below is an example:

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

Initial Puzzle

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

Solution

The object of Sudoku is to place numbers 1 through 9 in the empty spaces such that no row, column, or 3x3 subgrid has any number more than once. An interesting property of Sudoku puzzles is that there is always only one possible solution, and it can always be determined using logic, without the need for guessing. Although Dave is wild about Sudoku, he still comes up with incorrect solutions. Dave is tired of being made fun of by his more Sudoku-savvy friends, so he's asked you to write a program to check his solutions for him.

The Problem:

Given a set of Dave's Sudoku puzzle solutions, determine which ones are correct, and which are invalid. For a Sudoku solution to be correct, every row, column, and 3x3 subgrid of the puzzle must have each digit (1 through 9) exactly once.

The Input:

Dave will give you multiple solutions to check. Input will begin with a single integer, *n*, on a line by itself. Following this will be *n* sets of 9 lines, each containing 9 digits. Each digit will be in the range 1 through 9, inclusive. Each set of 9 lines of 9 digits represents one of Dave's potential Sudoku puzzle solutions.

The Output:

For each Sudoku solution, print "`Sudoku #d:`", where *d* represents the number of the puzzle (beginning at 1). Follow this with two spaces and either "`Dave's the man!`", if the solution is correct, or "`Try again, Dave!`", if it is invalid. Leave a blank line after the output for each solution.

Sample Input:

```
2
357648912
216539748
948712536
521486397
463197285
789325164
632974851
174853629
895261473
263847159
514936278
987125364
645382917
139574826
872619543
658791632
791263485
326458791
```

Sample Output:

```
Sudoku #1:  Dave’s the man!

Sudoku #2:  Try again, Dave!
```