Fourteenth Annual
University of Central Florida

ACM · UPE

High School Programming Tournament

Problems

Problem Name	Filename
Gotta Fetch ‘em All!	POKEYMEN
Soul Tracker	SOULTRAK
Secret Judging League	LEAGUE
Acknowledged	ACKL
Bounding Rects	RECT
Ali’s Area Code	AREA
Ultimate Fantasy Tactics	TACTICS
Cinco de Mayo	MAYO
The Pool Game	POOL
These Are Odd Days We Live In	DAYS

Call your program file: *Filename*.PAS, *Filename*.C or *Filename*.CPP
Call your input file: *Filename*.IN

For example, if you are solving Ali’s Area Code:
Call your program file: AREA.PAS, AREA.C or AREA.CPP
Call your input file: AREA.IN

Gotta Fetch 'em All!

Filename: `POKEYMEN`

Nash Fetchum wants to be the best Pokeyman trainer of all time. For those of you who don't know (where have you been, anyway?), Pokeymen are cute little monsters that kids can capture and train to fight the Pokeymen of other trainers in brutal battles with all sorts of dangerous special abilities for no apparent reason. It's OK, though, the Pokeymen seem to enjoy the matches.

Each Pokeyman can be classified as a certain "element." The elements are fire, water, earth, air, electricity, poison, psychics, and fighting. In addition, if a Pokeyman gains enough experience in its matches, it can evolve into a more powerful form and gain new special abilities in its element. For example, one of the more popular electric Pokeymen, Peekaboo, is a level 1 Pokeyman. However, it can evolve into the more powerful Ryeboo, which is level 2. There are at most three levels of evolution.

Pokeymen from certain elements have advantages over certain other elements. Water Pokeymen, for example, are generally very effective against fire Pokeymen. However, an electric Pokeyman can usually defeat a water Pokeyman without a problem. Psychic Pokeymen are very powerful, and have an advantage over every other element. See the table below for the list of advantages between elements.

Element	**Strong Against (advantage +1)**
Fire	Earth
Water	Fire
Earth	Electricity
Air	Earth
Electricity	Water
Poison	Fire, Earth, Water, Air
Psychics	All (except Psychics)
Fighting	None

A Pokeyman facing another Pokeyman of the same element and evolutionary level is said to have an advantage of zero. Fighting against a different element can increase or decrease the advantage by 1, so a water Pokeyman fighting a fire Pokeyman of the same evolutionary level would have an advantage of 1. If the water Pokeyman were level 2 while the fire Pokeyman was level 1, the water Pokeyman's advantage would be 2. Conversely, if the fire Pokeyman were level 2, while the water Pokeyman was level 1, the evolutionary advantage of the fire Pokeyman would cancel out the elemental advantage of the water Pokeyman, and the total advantage would be zero.

The Problem:

Nash has captured so many Pokeymen that he has a hard time selecting the right Pokeyman for his matches. He wants you to write him a program to help choose the best Pokeyman from his bench, given the element and level of evolution of his opponent's Pokeyman. Nash doesn't know what his next opponent will use until the match begins, so he can't select his Pokeymen ahead of time. However, once a Pokeyman has fought in a match, it can't fight again until the next tournament, so he wants to save his best Pokeymen for the really big matches, if possible. Help Nash out by writing him a program that will select the Pokeyman that has the smallest advantage over his next opponent. If Nash doesn't have such a Pokeyman, pick the one with the smallest disadvantage.

The Input:

There will be multiple data sets. For each set, input will begin with a single integer N ($0 \le N \le 50$) on the first line, indicating how many Pokeymen Nash has in his corner. (If N is 0, this indicates that there is no more data, and the program should stop.) On the next N lines will be the descriptions of Nash's Pokeymen. First on the line will be a string (maximum of 10 characters) with the name of the Pokeyman. Next will be a single space and a string giving the Pokeyman's element in lower-case letters. You may assume that the element will be one of the eight listed in the table above. Finally, there will be another single space followed by a number between 1 and 3, inclusive, indicating the evolutionary level of the Pokeyman.

After the descriptions of Nash's Pokeymen, there will be an integer M ($M \le N$) on the next line by itself, indicating the number of matches. On the next M lines will be descriptions of Nash's opponents in each match. The description will be a string indicating the opposing Pokeyman's element in lower-case letters, followed by a single space and a number between 1 and 3, inclusive, indicating the level of evolution.

The Output:

Before each data set, print a line of twenty asterisks followed by a blank line. For each match, print the number of the match (starting at 1), then the name of the Pokeyman with the smallest advantage greater than zero over the opponent's Pokeyman. If no such Pokeyman exists, print the Pokeyman that's evenly matched against the opponent (advantage of zero). If neither criterion can be met, print the Pokeyman with the smallest disadvantage (negative advantage with smallest absolute value). If there is more than one Pokeyman with the optimal advantage, print the first one listed in the input. No Pokeyman can appear more than once in the output. Follow the format of the Sample Output, placing a single space after the colon and comma, and leaving one blank line after each match.

Sample Input:

```
5
Peekaboo electricity 1
Sandsmash earth 2
Bulbaspore poison 1
Charagard fire 3
Squirtail water 1
4
air 1
water 1
psychics 2
fire 1
3
Venuspore poison 3
Peekaboo electricity 1
Fidgy air 1
1
air 1
0
```

Sample Output:

```
********************

Match 1: Bulbaspore, I choose you!

Match 2: Peekaboo, I choose you!

Match 3: Charagard, I choose you!

Match 4: Squirtail, I choose you!

********************

Match 1: Venuspore, I choose you!
```

Soul Tracker

Filename: `soultrak`

Ever since you discovered MP3s, you've been searching the web for a bootleg copy of your favorite song, "Invisible Touch" by Genesis. One day you said, "Man, I would give my very soul for a copy of 'Invisible Touch'!" You heard a rumbling and the words, "Granted, foolish mortal!" and you felt a wrenching sensation as your soul was ripped from you. Then a CD labeled "Genesis: Invisible Touch" appeared in your hand. Oh no! You traded your soul for a Genesis CD! And you'll have to rip the tracks yourself! Satan sucks.

After ripping the tracks, you decide that you need to find your soul. You phone Satan up and ask how you could buy your soul back. He says, "Hmm. Well, how about an autographed copy of Bjork's 'Homogenic' CD?" You agree to his demands and hang up.

The Problem:

You must find a way to get this CD! You round up your friends and find that each is willing to trade one thing for another. You must write a program that finds whether there is a way to trade things which results in your owning the CD that Satan demands.

The Input:

There will be multiple scenarios in the input file. The first line of the input file will be an integer, N, ($1 \le N \le 100$). This indicates how many scenarios are in the file. Next, for each scenario, there will be a line with an integer, M, ($1 \le M \le 20$), which indicates the number of friends in this scenario. Following will be M sets of three lines each. The first line of each set is the name of the friend. The second line is the item this friend wants to have. The third line is the item this friend will give you if you give them the item from the second line.

Each of the lines with text on them will contain from 1 to 40 characters. There will be no leading or trailing spaces or tabs on any of these lines. For purposes of matching, assume that items must be spelled exactly the same way. Therefore, "A kaleidoscope" is *not* the same as "A kaleidoscope", because the second one has more spaces in it. And "a kaleidoscope" is also not the same, because the capitalization is different.

The Output:

For each scenario, determine if there is a way to trade items so that you end up with an item named "Homogenic CD". (You start with an item called "Invisible Touch CD".) If there is a way to get your soul back, print the message

```
I can get my soul back!
```

Otherwise, print the message

```
I am doomed to hell for all eternity!
```

Sample Input:

```
2
5
Sally
Box of Rice Krispies
Can of Spinach
Garry
Invisible Touch CD
Box of Rice Krispies
Anette
He-Man Action Figure
Homogenic CD
Sue-Lisa
Swiss Army Knife
He-Man Action Figure
Arnold
Can of Spinach
Swiss Army Knife
2
Fred
Pair of Pliers
Homogenic CD
Lisa
Box of Triscuits
Homogenic CD
```

Sample Output:

```
I can get my soul back!
I am doomed to hell for all eternity!
```

Secret Judging League

Filename: LEAGUE

The Secret Judging League is an organization of individuals that judge many programming contests including this one. However, since secrecy and security are of upmost importance, they all have secret code names in order to gain admission into the renowned Hall of Judging. Unfortunately, some of the newer members have trouble remembering their code names so they would like you to write a program to help them determine what it is again.

The Problem:

Given a full name with a first name, zero or more middle names and a last name, determine the person's code name. The Secret Judging League has chosen the technique used by the Evil Professor in "Captain Underpants and the Perilous Plot of Professor Poopypants" by Dav Pilkey. In this technique, each person uses the first letter of their real first name to choose a new first name from the first column of the table below, and the first and last letters of their real last name to choose the two parts that make up the new last name from the other two columns. For example, Ali Orooji would get "Stinky Burgerlips" as his code name (if he was a member of the Secret Judging League and we won't say that he is or isn't).

	First Letter of First Name	First Letter of Last Name	Last Letter of Last Name
A	Stinky	Diaper	head
B	Lumpy	Toilet	mouth
C	Buttercup	Giggle	face
D	Gidget	Bubble	nose
E	Crusty	Girdle	tush
F	Greasy	Barf	breath
G	Fluffy	Lizard	pants
H	Cheeseball	Waffle	shorts
I	Chim-Chim	Cootie	lips
J	Poopsie	Monkey	honker
K	Flunky	Potty	butt
L	Booger	Liver	brain
M	Pinky	Banana	tushie
N	Zippy	Rhino	chunks
O	Goober	Burger	hiney
P	Doofus	Hamster	biscuits
Q	Slimy	Toad	toes
R	Loopy	Gizzard	buns
S	Snotty	Pizza	fanny
T	Falafel	Gerbil	sniffer
U	Dorkey	Chicken	sprinkles
V	Squeezit	Pickle	kisser
W	Oprah	Chuckle	squirt
X	Skipper	Tofu	humperdinck
Y	Dinky	Gorilla	brains
Z	Zsa-Zsa	Stinker	juice

The Input:

Input will begin with a single positive integer, *N*, on the first line, indicating how many names to process. On each of the next *N* lines will be a name consisting of a first name, zero or more middle names, and a last name each separated by a single space. Each name will contain only letters with the first letter of each name capitalized. No line in the input will be longer than 80 characters.

The Output:

For each name, output the code name for that person's name.

Sample Input:

```
3
Ali Orooji
Michael Smith
Harry S Truman
```

Sample Output:

```
Stinky Burgerlips
Pinky Pizzashorts
Cheeseball Gerbilchunks
```

Acknowledged

Filename: ACKL

Adapted from an example used by Dr. James Rogers in the UCF course COT4210

The Problem:

You are debugging a distributed application, and you've decided to write a program to run through logs of network activity to detect errors in communication. There are two entities involved in the communication protocol, a server and a client. The client can send the following messages, subject to the constraints that follow them:

- `m1` – can be sent only if every previous `m1` that was sent to the server was acknowledged, or if none have been sent.
- `m2` – can be sent only if every previous `m2` that was sent to the server was acknowledged, or none have been sent.
- `m12` – can be sent only if every previous `m1` sent was acknowledged and every previous `m2` sent was acknowledged. It can also be sent when neither `m1` nor `m2` have been sent. This message sends `m1` and `m2` at the same time.

The server can send the following messages, subject to the constraints explained with them:

- `a1` – The server can acknowledge the client's `m1` with this only if the client has sent a `m1` that has not been acknowledged yet.
- `a2` – The server can acknowledge `m2` with this only if the client has sent a `m2` that has not been acknowledged yet.
- `a12` – The server can acknowledge an outstanding `m1` and `m2` at the same time by sending this; otherwise it cannot be sent.

A transaction is said to be valid if and only if each message was sent correctly according to the rules above, and every client message has been properly acknowledged.

The Input:

The input will consist of multiple sets. Each set consists of one or more lines of a message string, ended by the string `EOT` on a line by itself. A message string is one of the following on a line by itself: `m1`, `m2`, `m12`, `a1`, `a2`, and `a12`. These represent the messages sent over the network in the order that they appear in the set. The end of input is marked by a set that starts with the string `EOF`, instead of starting with one of the message strings.

The Output:

For each set in the input, produce the following output. Begin with a label indicating the number of the transaction in the input. On the same line, follow this with an indication of whether or not the string of messages in the order they are presented represent a valid transaction between the server and the client. This should be followed by one blank line. Follow the format presented in the Sample Output.

Sample Input:

```
m1
m2
a1
m1
a2
m2
a12
EOT
m1
a2
m2
a12
m12
a1
a2
EOT
EOF
```

Sample Output:

```
Transaction 1: valid

Transaction 2: invalid
```

Bounding Rects

Filename: RECT

It is common in computer graphics to use rectangles instead of complicated polygons for many calculations. (Picture your physics instructor saying, "First, assume a rectangular object." It's the same sort of thing.) A "bounding rectangle" is defined as the smallest rectangle that encloses a given polygon. A common variation of the bounding rect is known as an "axis-aligned bounding rect." An axis-aligned bounding rect is the smallest rectangle *with sides parallel to the axes* that encloses a given polygon. Because calculations with rectangles are much quicker and easier than calculations with arbitrary polygons, many programs utilize bounding rects. For instance, many windowing operating systems use bounding rects to determine how much of the screen to redraw. Your task is to compute the area of the axis-aligned bounding rect of a given polygon.

The Input:

There will be multiple input sets. The first line of the file will contain an integer, *N*, the number of polygons ($1 \le N \le 500$). Each polygon will begin with an integer on a line by itself, *v*, the number of vertices ($3 \le v \le 20$). The following *v* lines will each contain two integers, *x* and *y*, ($-100 \le x,y \le 100$), the coordinates of the vertex.

The Output:

For each polygon, output the message, "`The bounding rect of polygon i has area A.`", where *i* is the number of the polygon (starting with 1), and *A* is the area of the associated axis-aligned bounding rect.

Sample Input:

```
2
4
1 1
-1 2
0 0
2 –1
7
1 5
7 13
8 12
2 6
0 0
4 4
-1 –1
```

Sample Output:

```
The bounding rect of polygon 1 has area 9.
The bounding rect of polygon 2 has area 126.
```

Ali's Area Code

Filename: AREA

The phone company is running out of telephone numbers and needs to institute 10-digit dialing. When using 10-digit dialing, the area code for the phone number must always be prefixed onto the actual phone number even when dialing a local number. Ali really hates doing this, but luckily he has a really cool phone. He can program it so that if he dials a number which begins with a digit other than the first digit in his area code, the phone will automatically insert his area code first. If Ali dials a number that begins with the same digit as his area code, the phone will not insert Ali's area code (since it cannot distinguish between a 7-digit number or if Ali has actually rememberd to use the area code). Write a program to simulate this phone.

The Problem:

Given a 7-digit phone number, add in the area code if the first digit of the number does not match the first digit of the area code. Ali's area code is 407.

The Input:

Input will begin with a single positive integer, *N*, on the first line, indicating how many phone numbers to process. On each of the next *N* lines will be a single phone number in `xxx-xxxx` format. There will be no spaces on any of the lines.

The Output:

For each phone number, output the new number with area code added as appropriate. Follow the format shown in the Sample Output.

Sample Input:

```
2
555-6473
411-8376
```

Sample Output:

```
(407) 555-6473
411-8376
```

Ultimate Fantasy Tactics

Filename: `TACTICS`

You've been playing Ultimate Fantasy games since the 8-bit days. After all of those battles, you feel that you've come up with a good strategy for distributing your party member's attacks over the enemies. To test your strategy, you are going to write a program to execute it for a party and enemy configuration.

The data you have to work with is the number of party members, the average amount of damage each party member inflicts, the number of enemies, and how many hit points (HP) each enemy starts with (referred to as the maximum HP for that enemy). Given this information, your program should figure out which enemy each party member should attack during the first round of combat so that you kill the maximum number of enemies in that round. An enemy is considered dead if his HP falls to zero or less. It may be that there are several ways to kill this maximum number of enemies; the method for breaking ties is as follows.

An attack pattern lists each party member and the enemy it is attacking. If there is more than one attack pattern that yields the maximum number of destroyed enemies, the one that has the lowest total *wasted damage factor* (WDF) should be chosen. For a given attack pattern, here is how to calculate the total WDF:

- Carry out the attack pattern by subtracting each party member's average damage from the HP of the enemy it is attacking. The HP value for an enemy after the attack pattern is carried out is called the enemy's current HP.
- Calculate the WDF for each enemy:
  - If the enemy's current HP is positive, the WDF is 0.
  - Otherwise, the WDF for the enemy is:

$$\text{WDF} = -\frac{\text{current HP}}{\text{maximum HP}}$$

- Sum the individual enemy WDF values.

If there are ties among the attack patterns after considering all of the total WDF values, the one with the lowest total *leftover damage factor* (LDF) should be chosen. The total LDF is calculated as the total WDF is calculated, except the LDF for a given enemy is as follows:

- If the enemy's current HP is negative, the LDF is 0.
- Otherwise, the LDF for the enemy is:

$$\text{LDF} = \frac{\text{current HP}}{\text{maximum HP}}$$

If there are still ties among the attack patterns after considering all of the total LDF values, any of the attack patterns in the tie is considered optimal. Two WDF or LDF values are considered equal if they are the same out to 6 decimal places.

The Input:

The input consists of multiple sets. Each set consists of two lines. These lines contain only digits, spaces, and new-line characters. There will be one space between each pair of numbers, with the first number starting in the first column of the input file.

The first number on the first line will be the number of members in the party; call it P ($1 \le P \le 5$). Following are P positive integers, which are the average damage values for the party members. The first value is for the first party member (party member number 1), the second value is for the second party member (number 2), and so on.

The first number on the second line is the number of enemies; call it E ($1 \le E \le 8$). Following are E positive integers, giving the enemy HP values in order. The first value is for the first enemy (number 1), the second value for the second enemy (number 2), and so on. End of input is indicated by $P = 0$ (this case should not be processed).

The Output:

For each case that is processed, produce a table using the format shown in the Sample Output. The battles are numbered starting with 1. The numbers under the `Party Member` heading are the numbers of the party members (1 to P inclusive), and should be listed in order. For each party member number, print the number of the enemy it is attacking (enemy numbers are numbered 1 to E inclusive – in the order that they appear in the input) under the `Enemy` heading. The total WDF for the attack pattern should be printed underneath the `Party Member` heading, and the total LDF under the `Enemy` heading, as shown in the Sample Output. These values should be rounded to three decimal places. Place one blank line after each data case.

Sample Input:

```
4 13 19 35 70
3 30 35 20
0
```

Sample Output:

```
+=========================================+
| Battle #1                               |
+=========================================+
| Party Member        Enemy               |
| ------------        -----               |
| 1                   3                   |
| 2                   3                   |
| 3                   1                   |
| 4                   2                   |
+=========================================+
| WDF: 1.767          LDF: 0.000          |
+=========================================+
```

Cinco de Mayo

Filename: `MAYO`

Mason really likes mayonnaise. Unfortunately, his doctor doesn't. You see, Mason's cholesterol is somewhat high, so his doctor wants him to eat fewer foods which have mayonnaise. His doctor wants your help to watch over Mason.

The Problem:

Given the list of ingredients in each of Mason's meals, warn him when he has had mayonnaise five or more times within a single week.

The Input:

Input will begin with a single positive integer, *N*, on the first line, indicating how many weeks to watch over Mason. For each week, there will be a line with a single positive integer, *M*, indiciating how many meals Mason had this week. On each of the next *M* lines there will be a single-space-separated list of single word ingredients in each of Mason's meals, respectively. Each line will be no longer than 80 characters. A word will be a sequence of one or more lower-case letters. Note that Mason may list mayonnaise either as "`mayo`" or "`mayonnaise`" and that he never has it more than once in each meal.

The Output:

For each week, output a header for that week as shown in the Sample Output. Follow the header with a single space and then either "`Cinco de Mayo!`" if Mason had mayonnaise five or more times that week or "`Way to go, Mason!`" if he did not. List one response per line, and leave one blank line after the output for each week.

Sample Input:

```
2
6
hamburger bun ketchup lettuce mayo pepper
turkey lettuce bun tomato peppers mayonnaise oil vinegar
hotdog bun fries ketchup mayo drink
chicken sauce bun
fish chips ketchup mayonnaise
hamburger bacon bun mayo salt
1
salad dressing
```

Sample Output:

```
Week #1: Cinco de Mayo!

Week #2: Way to go, Mason!
```

The Pool Game

Filename: `POOL`

Spike and his friend Don decided to play pool one day. Unfortunately, Spike had not played it much so he had some difficulties. In particular, he often hit it straight ahead with such force that the ball for which he was aiming would exit the playing surface. To make matters worse, there were people sitting at tables around the table in the pool hall where they played. You are to help Spike determine whether or not he will hit somebody if he hits the ball out of the playing area.

The Problem:

Given two points that represent the cue ball and the ball for which Spike is aiming, and given (x, y) points for each of the locations of people sitting around the table, determine if Spike will hit any of them. Assume that all the balls and all the people sitting around the table are points (i.e. they have no width), and that Spike will always hit the ball in a perfectly straight line (he hits too hard but he is incredibly accurate).

The Input:

Input will begin with a single positive integer, *N*, on the first line, indicating how many shots with which Spike needs help. For each shot, there will be a line containing five integers representing the (x, y) position of the cue ball, the (x, y) position of the ball for which Spike is aiming, and the number of people sitting around the table, *M*. On the next *M* lines will be integer pairs, (x, y), describing the position of each person. Assume all positions will be in the positive x and positive y quadrant.

The Output:

Output will be either "`Watch out, Spike! You're going to hit somebody!`" if Spike will hit somebody or "`Go ahead, Spike. It looks clear.`" if Spike will not. List only one response per line.

Sample Input:

```
2
1 1 5 1 3
9 35
20 1
1 33
3 4 4 4 1
12 8
```

Sample Output:

```
Watch out, Spike!  You're going to hit somebody!
Go ahead, Spike.  It looks clear.
```

These Are Odd Days We Live In

Filename: DAYS

November 19, 1999 was an odd day. This is because all of the digits in its numeric format (11/19/1999) are odd. As it turns out, the next odd day will not occur until January 1, 3111 (1/1/3111). This is a rather large number of days away so you decide to write a program to figure out how many days away the next odd day is from a given date.

The Problem:

Given a date, figure out how many days away the next odd day is. Ignore leap years for the purposes of this problem.

The Input:

Input will begin with a single positive integer, *N*, on the first line, indicating how many dates to process. On each of the next *N* lines will be a single date in month/day/year format. For each date, compute how far away the next odd day is. Note that no numbers will be zero padded (for example, the month January is listed as "1" and not "01"), and that no years will be more than four digits.

The Output:

For each date, output how many days away the next odd day is.

Sample Input:

```
2
11/19/1999
1/1/3111
```

Sample Output:

```
405558
2
```