Problem Set

Trinity University ACM
High School Programming Competition
April 7th, 2007

Problem 0 – Above Average

Professor Hughes has a long standing tradition that if the class average on an exam, rounded to the nearest whole point, is less than an 80, all scores are increased by the difference between the average and an 80 to boost the average up to an 80.

So if a small class had scores of 90, 72, and 68 this would give a class average of 76.67 which rounds to 77. In this case 3 points would be added to all the scores to bring the average up to 80. The poor professor is getting along in years though and is tired of doing this by hand. As such, you have been asked to write a program that takes the list of students and grades and makes the proper adjustment.

Note that if the average is greater than 80 then the scores aren't changed at all.

Input: The first line of the input tells you how many classes Professor Hughes is processing in this run. Each class will begin with a line telling you the number of students in the class. For each student there will be a line giving the student's name and the score he/she got on the exam. All scores will be integer values between 0 and 100 inclusive.

Output: For each class you need to output "Class N had an average of A and a curve of C" where N is replaced by the number of the class starting at 1 for the first one, A is the average for the class rounded to the closest integer, and C is how many points the scores were curved by. After this you will print one line for each student with the student's name followed by their curved score.

Sample Input:

```
2
3
Jason 90
Cat 72
Billy 68
5
Meagan 95
Lily 90
Dave 66
Kim 81
Lauren 100
```

Sample Output:

```
Class 1 had an average of 77 and a curve of 3
Jason 93
Cat 75
Billy 71
Class 2 had an average of 86 and a curve of 0
Meagan 95
Lily 90
Dave 66
Kim 81
Lauren 100
```

Problem 1 – Artistic License

You have been given an assignment in your art class to come up with an abstract piece involving only prints of circles and rectangles. You have decided that to express your inner emotions on the topic for your piece you need to have some of the shapes overlapping and others not. For this problem all you have to do is determine if two shapes overlap or not. The shapes will be either circles or rectangles.

Input: The input for this problem consists of one line telling you how many checks you will make, followed by that many pairs of shapes. Each shape will be on a single line and will have one of the following formats. If it is a rectangle the line with start with 'R' and that will be followed by 4 numbers, x y width height, where x and and y are the position of the top left corner and width and height are the size of the rectangle. If it is a circle the line with start with a 'C' that is followed by 3 numbers, x y radius. Values can be non-integers and shapes will never touch, they will either overlap or not have contact.

Output: For each pair of shapes you should print either "They overlap." or "They don't touch."

Sample Input:

```
3
R 1 1 3 3
R 2 2 4 4
C 1 1 3
C 20 10 5
R 4.5 4.5 2 5
C 2.5 2.5 20
```

Sample Output:

```
They overlap.
They don't touch.
They overlap.
```

Problem 2 – Touchdown!

One of the most heated rivalries in all of NCAA Division II football is California versus Indiana. The two teams from Pennsylvania are in the same division and play each other once a season. This last season, the score was very close, with Indiana winning by four points, 21 to 17.

If you know how scoring works in American football, then you could guess that Indiana scored 3 touchdowns and converted all of the extra points ((6+1) x 3 = 21). However, there are other ways to reach 21. One example would be a safety, a touchdown with a missed extra point, another safety, a touchdown with a 2-point conversion, and a field goal: 2 + 6 + 2 + (6+2) + 3 = 21.

Scoring Rules:
• Safety: 2 points
• Field Goal: 3 points
• Touchdown: 6 points
  – Point After Touchdown (PAT): 1 point
  – 2-point Conversion: 2 points

Note: a PAT and a 2-point conversion can only come after a touchdown is scored.

Input: You will be given an integer n on a single line that will be represent the number of scores to calculate (0 n 50). Each of the following n lines will have a single integer, m, representing a valid football score.

Output: For each m, you will output the total number of distinct combinations that m can be reached using the Scoring Rules stated above. Chronology doesn't matter; that is, 2 touchdowns and a field goal is the same as a field goal and 2 touchdowns.

Sample Input:

```
2
21
8
```

Sample Output:

```
25
4
```

Problem 3 – Rounders

In your undergraduate career at Trinity, many research opportunities are available to you. One such opportunity deals with simulation of the rings of Saturn. As you know, the numbers used in Astrophysics can be rather large and are generally required to be rounded.

With this particular project, the numbers are so large that they cannot be stored in the double (double-precision) data type. The professor doing this research has his simulation code specially written to handle these large values, but he has not gotten around to writing the code that will round the large numbers.

You decide to volunteer and possibly get a publication out of your help. To test your rounding, you create a simple program.

Rounding Rules:
• If there is no decimal (.), then there is no rounding to be done.
• Round to the nearest whole number.
• If the value after the decimal point is greater than or equal to one half, then round up; else, round down.

Input: You will be given an integer n on a single line that will represent the total number of numbers to round (0 n 50). Each of the following n lines will have a sequence of digits with at most one decimal point. Any given sequence will be no longer than 100 characters and will have at least one digit in it.

Output: For each sequence, you will output a single line with the correctly rounded sequence of digits.

Sample Input:
```
2
123.4
56.7
```

Sample Output:
```
123
57
```

Problem 4 – Diffuse

You are a member of an elite counter-terrorist organization. You're team has received intelligence that a terrorist organization is bombing a giant replica of a kitchen. You have deployed to the kitchen, and you're team has neutralized the terrorists.

You have located the bomb next to a large spray painted A under the kitchen sink. On the bomb's casing there are two square girds of buttons that may be either lit or off. In order to defuse the bomb you must make the picture on the top grid match the picture on the bottom grid. If you push a button it will change to its opposite state as will the buttons on the sides, above and below, and diagonal to it. You only get to push one button before the bomb explodes.

Input: You will be given an integer n on a single line that represents the number of bombs you must defuse ($0 < n \leq 50$). The first line of each bomb to defuse consists of an integer, m, which specifies both the width and height ($3 \leq m \leq 6$). The next m lines consist of the working grid followed by the goal grid.

Output: The output for each bomb shall consist of a single line. If the bomb can be defused in the number of seconds listed the output is "Bomb n: Counter-Terrorists Win", or if the bomb cannot be defused in that time the output is "Bomb n: Terrorists Win" where n is the number of the current bomb.

Sample Input:

```
2
4
####
#00#
#0#0
##00
####
#00#
#00#
####
5
#####
00000
00000
00000
#####
#####
00000
0###0
0###0
#####
```

Sample Output:

```
Bomb 1: Counter-Terrorists Win
Bomb 2: Terrorists Win
```

Problem 5 – Shirts Я Us

One of the challenges in doing this programming competition is the ordering of T-shirts and trying to get shirts for all the students that are of the right sizes. We want you to help us out a bit this year by writing a program that will determine if everyone can be satisfied given a certain shirt order. The input for the program will tell you how many shirts we ordered of each of five sizes: XL, L, M, S, XS. For each competitor, you will be given a range of sizes that that person would be happy with getting their shirt in. For example, XL-M would imply that the person was happy with an XL, L, or M sized shirt. The question for you to answer is whether it is possible to make everyone happy given what we have ordered and the contestant preferences.

Input: The input will begin with a number telling you how many scenarios we want you to consider. Each scenario will start with a line that has 5 numbers on it representing how many shirts we have ordered in XL, L, M, S, and XS sizes respectively. There will never be more than 20 of any given shirt size. That line will be followed by a number, N<=20, tell you how many contestants there are. After this will be N lines, one for each contestant, specifying the shirt sizes they find acceptable in the form of largest-smallest. They can be the same if the person is only willing to accept one shirt size. There will be no spaces in the lines giving the shirt sizes.

Output: For each scenario you will output one line. It will either say, "Great Contest! Cool Shirt!" if everyone can be satisfied or it will say, "I'd rather not wear a shirt anyway." if there is no acceptable distribution of shirts.

Sample Input:

```
2
1 2 2 1 0
5
XL-M
S-XS
M-M
L-M
XL-L
1 2 0 2 1
5
XL-L
L-M
XL-M
M-S
L-L
```

Sample Output:

```
Great Contest! Cool Shirt!
I'd rather not wear a shirt anyway.
```

Problem 6 – BASICly

In your computer history class, you learned about the BASIC programming language. After the class ends, you walk to your next class, but all along the way, you think about how easy it would be to write an interpreter for the language. So as a weekend project, you decided to try to write one.

After thinking more about this, you decide that you only want to get simple mathematical operations working (assignment and addition). Addition will test the simple math skills of the language, and the assignment operation will help you store a single variable. Knowing that this task is far below your programming capabilities, you decide to add the GOTO ability to your interpreter.

Rules:
• there is only one variable, A
• each line is represented as a line number followed by one of the valid expressions
• lines are executed sequentially unless there is a GOTO
• a GOTO operation jumps to the given line label, i.e. this jumps to the line labeled 20:
10 GOTO 20
• a GOTO operation will be the only operation on that line
• only one assignment operation (=) per line is allowed:
20 A=5
• there can be many addition operations (+) for each line and the operands can be constants or A:
30 A=A+A+A+A+10
• when you reach the END command, stop processing

Each line will be valid. There will be no more than 100 lines (label possibilities ranging from 1 to 100). The program will end. Negatives numbers are not valid and all numbers will be integers. When the program starts A is zero.

Input: You will be given a number of lines. Each line will follow the rules above. A 0 (zero) on a line denotes the end of the code. The lines will be given to you in order.

Output: At the end of the program, you will output on a single line the contents of A.

Sample Input:
```
10 GOTO 30
20 END
30 A=20
40 A=A+A+A
50 GOTO 20
0
```

Sample Output:
```
60
```

Problem 7 – How I Learned To Jump Good

In a purely fictional future, Commander Lee is a pilot of a scouting spaceship who has just spotted an incoming alien army. Lee's spaceship is equipped with hyperdrive technology, allowing travel at the speed of light across space. Knowing the location of the closest human command bases, you must help Lee figure out the best direction to jump in to arrive closest to a command base. Of course, since you are in 3-D space the locations of the command bases are given as x, y, and z coordinates. Since positions are relative we assume (WLOG) that Commander Lee is at the origin.

Input: An Integer $1 < N < 20$ of scenarios to consider. Each scenario will give the number of bases M (<=10) followed by M lines with the position, given as three numbers, and name of each base. There will never be two bases equidistant from Commander Lee. The numbers do not have to be whole numbers and the base names do not have to be single words.

Output: For each scenario you simply print the name of the base Commander Lee should jump to.

Sample Input:

```
2
3
3 4 0 Alpha
2 2 2 Beta
5 6 8 Gamma
2
6.2 2.1 5.3 Zain
10.7 5.3 18.8 Shamsi
```

Sample Output:

```
Beta
Zain
```

Problem 8 – Road Trip

You and your friends are planning a road trip. Unfortunately, you are a bit strapped for cash. Actually, you are all dirt poor. You figure you have enough money for two tanks of gas. One tank can get you where you want to go and the other has to get you back. You can go 400 miles on a tank.

In addition to being financially challenged, you are lazy and you don't feel like looking through mileage charts to determine where you can go. Thankfully, you found a web site with a handy listing of distances from your home town to various cities. All you have to do now is write a program that will trim the list down to the ones you can actually visit so you and your friends have a nice short list to pick from.

Input: The input will start with a line with a single number, N, on it telling you how many cities you will consider. After that will be N lines with the city name followed by a distance in miles. City names will not have more than one word in them. For those that should the space will be left out.

Output: You should print out only the cities within 400 miles.

Sample Input:

```
5
Austin 95
LasVegas 1400
Houston 180
Boulder 1200
BatonRogue 500
```

Sample Output:

```
Austin
Houston
```

Problem 9 – aMazingly Tired

A group of your "friends" have driven you out into the country and put you in a hay bail maze. This might normally be a fun thing, but they stole you out of bed at 3:30am and you a dead tired. You figure you only have a limited number of steps you can take before you collapse and you aren't too keen on the idea of sleeping the rest of the night in the field. They question is, can you possibly make it out of this maze before you give up and fall asleep?

Input: The input will begin with the number of mazes you have to consider. The mazes are square and each scenario begins with two numbers telling you how big the maze is (N<=10) and how many steps you can take before you fall asleep (this will be a positive integer). The next N lines will have a text encoding of the maze using four characters. A # represents a wall you can't get through. A . represents an area you can walk through. There will be one S in the maze tell you where you start and one E in the maze telling you where you have to get to so you can catch you friends and not spend the night in the field. There will be a path from S to E. Each character represents one step for you to walk. You can't leave the bounds of the maze. For determining the answer assume that you will take the shortest path from S to E.

Output: For each maze you will output a single line. If the path from the start to the end is longer than you can walk you print, "People used to sleep on hay, right?" If you can make it to the end you print, "Get me back to my bed guys."

Sample Input:

```
2
6 10
....#.
.S..#.
.####.
......
#####.
.E....
6 10
....#.
.S..#.
.####.
......
.####.
.E....
```

Sample Output:

```
People used to sleep on hay, right?
Get me back to my bed guys.
```