The Anniversary Contest

USC Programming Contest, Spring 2015

April 04, 2015

Sponsored by VSoE, Electronic Arts, Facebook, Google, Qualcomm, SDL, Vivántech

This semester, we are celebrating the ten-year anniversary of the USC Programming Contest. Yes, in Spring of 2005, we got started with 33 participants! To celebrate this anniversary, we will take a trip down memory lane, and you will get to solve one problem each from the first six contests. Let's see whether you folks can compete with the old school.

You can solve or submit the problems in any order you want. When you submit a problem, you submit your source code file using PC^2. Make sure to follow the naming conventions for your source code and the input file you read. Remember that all input must be read from the file of the given name, and all output written to `stdout`. You may not use any electronic devices besides the lab computer and may not open any web browser for whatever reason.

One thing you should know about this contest is that the judging is done essentially by a `diff` of the files. That means that it is really important that your output follow the format we describe — if you have the wrong number of spaces or such, an otherwise correct solution may be judged incorrect. Consider yourselves warned!

Another warning that seems to be in place according to our experience: all of our numbering (input data sets, classes, etc.) always starts at 1, not at 0.

And a piece of advice: you will need to print floating point numbers both rounded to two decimals. Here is how you do that:

C: `printf ("%.2f", r);`.

C++: `cout.precision (2); cout << fixed << r;`.

Java: `System.out.print ((new java.text.DecimalFormat("#.##")).format(r));`

[Including this page, the problem set should contain 9 pages. If yours doesn't, please contact one of the helpers immediately.]

Problem A: Spring 2005: Arriving at USC

File Name: `spring05.cpp|spring05.java`

Input File: `spring05.in`

Description

For the very first contest, in Spring of 2005, we explored a few things around USC, calling it simply "The USC Contest". One of the things we explored was the use of building name abbreviations such as SAL, PHE, OHE, or SSL. The students back then — and you now — were asked to write a program to return all the candidate building names that an abbreviation could refer to.

You will be given a list of building names, and a building abbreviation, such as SAL or LOL. The abbreviation matches a building name if all of its letters appear, in this order, in the building name (no letter can be matched twice). So, for instance, SAL matches "SALvatori" or "Student Aerospace Laboratory", or "univerSity of southern cALifornia". It does not match "angeles", as the letters are in the wrong order. For the comparison, we will ignore case, so 'S' and 's' are the same letter.[1]

Input

The first line is the number K of input data sets, followed by the K data sets, each of the following form:

The first line of the data set contains the number n of buildings, $1 \leq n \leq 100$. This is followed by n lines, each containing the name of a building, consisting of only uppercase and lowercase letters and white space. Finally, the building code will follow on a line by itself, consisting only of letters. Each string will be at most 100,000 characters long.

Output

For each data set, output "`Data Set x:`" on a line by itself, where `x` is its number.

Then output all of the building names that match the building code, each on its own line, and in the order in which they appeared in the initial list.

Each data set should be followed by a blank line.

Sample Input/Output

Sample input `spring05.in`

```
2
3
studENt aeRoSpace laBoratory
salVAtori
angeles
SAL
2
SaLvatori
Science LibrarieS
SSL
```

Corresponding output

```
Data Set 1:
studENt aeRoSpace laBoratory
salVAtori

Data Set 2:

```

[1] We only used capitalization above to show you the match.

Problem B: Fall 2005: Summer on the Beach

File Name: `fall05.cpp|fall05.java`

Input File: `fall05.in`

Description

In Fall of 2005, the contest explored a "Summer on the Beach". One of the most important activities on the beach is lying on the sand and sun bathing. As you are all aware, prolonged exposure to sun can cause painful sunburns, so the use of sunscreen is highly recommended. Another approach pursued by many people is to lie under a parasol, protecting them from rays. But if you know from when to when you will lie at the beach, where should you place the parasol to protect as much of your body as possible?[2]

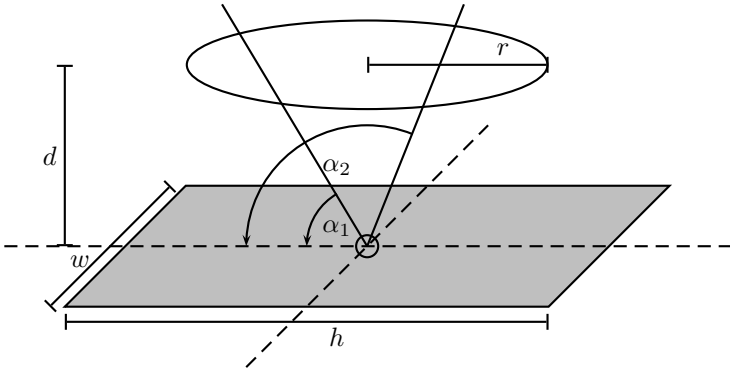
Here, we will answer a slightly easier question. Given the position of your body and the parasol, calculate which percentage of your body is exposed to the sun for at least part of the time. For simplicity, we will treat your body as rectangular, of size $h \times w$ centimeters. We treat the beach as a two-dimensional plane, with the origin at the center of your body. The parasol is assumed to be a circle of radius r, mounted horizontally at distance d above your body, with the center at the origin as well. You are sunbathing for a period from α_1 to α_2, where $1^\circ \leq \alpha_1 < \alpha_2 \leq 179^\circ$ are the angle that the sun makes with the beach at the beginning and end of your sunbath. (Assume that the sun is a point infinitely far away from the Earth, and travels straight from the left to the right.)

Input

The first line contains a number $K \geq 1$, which is the number of input data sets in the file. This is followed by K data sets of the following form:

Each data set consists of a single line, containing (in this order) the real numbers $w, h, r, d, \alpha_1, \alpha_2$. We will always guarantee that $0.01 \leq w, h, r, d \leq 100$.

Here is a figure illustrating these quantities. Notice that the sun travels "from your feet to your head", and *not* "from your left arm to your right arm".



Output

For each data set, first output "`Data Set x:`" on a line by itself, where `x` is its number. Then, output the percentage of your body that will be exposed to sun rays at some time, rounding to two decimals.

Each data set should be followed by a blank line.

Sample Input/Output

[2] In a French student science competition, one winning entry was a "para-tournesol", a "sunflower parasol". Using a light sensor, the parasol moves as the sun does, protecting you as much as possible. We won't explore that avenue here.

Sample input `fall05.in`

```
2
0.5 1.8 5.0 0.2 70.0 120.0
0.5 2.0 0.75 1.0 45.0 90.0
```

Corresponding output

```
Data Set 1:
0.00%

Data Set 2:
76.41%
```

Problem C: Spring 2006: Midterm Elections

File Name: `spring06.cpp|spring06.java`

Input File: `spring06.in`

Description

In Spring of 2006, it was time for midterm elections, so the contest was titled the "Midterm Elections Contest", and we explored some computational aspects thereof. One of the topics we looked at were the famous "butterfly ballots". Butterfly ballots are a tried and true way to ensure that many voters will accidentally vote for the wrong candidate. In a butterfly ballot, all the candidates' names are on one side (say, the left), while all the boxes in which to mark a vote are on the other side. Now, if the boxes are a little bit shifted, it can be hard to tell which box corresponds to which candidate. For instance, look at the following ballot:

Which school do you like best?	
	☐
Berkeley	
	☐
Stanford	
	☐
USC	
	☐
UCLA	
	☐
CalTech	
	☐
UCSD	

You'll have to admit that if one isn't careful with this ballot, one might easily end up making a mark next to UCLA instead of USC. Of course, similar things could happen to political candidates.

You are to write a program to design a ballot that will make your candidate win, if possible. The assumption is that all candidates must be on the ballot, in some order from top to bottom that you can determine. The boxes will be on the other side, so that the first box is above the first candidate, the second box between the first and second candidate, and so forth. The assumption is then that among the voters who intend to vote for candidate i, half will actually vote for i, and half will accidentally vote for candidate $i + 1$. Of course, among the voters who intend to vote for the last candidate on the ballot, all will vote correctly. You are to decide if your candidate can be made to win the election. (If your candidate is tied for first place, we also consider that a win.)

Input

The first line contains a number $K \geq 1$, which is the number of input data sets in the file. This is followed by K data sets of the following form:

The first line of the data set contains a number n with $1 \leq n \leq 100$, the number of candidates on the ballot. Candidate 1 is the one you are trying to make win. This is followed by n lines, each containing a number $1 \leq v_i \leq 1,000,000$, the number of voters who intend to vote for candidate i. All the v_i will be even numbers, so you don't need to worry about what happens about division by 2.

Output

For each data set, first output "`Data Set x:`" on a line by itself, where `x` is its number. If it is possible for candidate 1 to win, then output "`Possible`" on a line by itself, otherwise output "`Impossible`".

Each data set should be followed by a blank line.

Sample Input/Output

Sample input `spring06.in`

```
2
5
10
60
8
94
54
5
10
60
8
94
56
```

Corresponding output

```
Data Set 1:
Possible

Data Set 2:
Impossible
```

Problem D: Fall 2006: 100 Years of Engineering

File Name: `fall06.cpp|fall06.java`

Input File: `fall06.in`

Description

In Fall of 2006, we were celebrating 100 years of engineering at USC. To join in the fun, the contest was titled "100 Years of Engineering", and had a problem for most of our engineering departments. Here, you get to learn about Aerospace and Mechanical Engineering. Aerospace engineers like building airplanes, rockets, and other stuff that flies. Not all of it is rocket science, though. Some questions are well within the grasp of programming contestants. For instance, once you have built a rocket to launch, you want to know how high it can fly before it lands.

Here is our model: the rocket consists of n stages, which are essentially fuel tanks. A stage lasts for a certain amount of time, and after it is empty, it is discarded. After the last fuel tank is discarded, the rocket cannot accelerate any more. As the rocket sheds its stages, it becomes lighter and lighter. We assume that the rocket itself weights $M > 0$ kilograms (kg), i.e., after shedding all its stages, the leftover weight is M.

In addition to M, you will be given, for each of the rocket's stages i, its weight $m_i \geq 0$, the duration for which it lasts $t_i \geq 0$ (in seconds s), and the amount of thrust (force) it produces during that time $F_i \geq 0$ (in $\frac{\text{kg m}}{\text{s}^2}$), where m stands for meters. We will pretend that the rocket fuel itself does not weigh anything. We assume that the rocket is shot straight up, and ignore all effects of wind, friction, leaving the Earth's gravity field, etc. We also assume that the initial speed of the rocket is 0 m/s, until the first stage starts burning.

The important physical formulas you will need are: (1) $a = F/m$, i.e., the acceleration of an object of mass m under force F is $a = F/m$. (2) In the Earth's gravity field, any object accelerates toward the Earth at a constant acceleration of $g = 9.81\frac{\text{m}}{\text{s}^2}$. (3) If an object starts out with velocity v, and accelerates at acceleration a, then in t seconds, it travels distance $vt + \frac{1}{2}at^2$, and its new speed after those t seconds is $v + at$.

Assume that fuel tanks are used (and discarded) in the order $1, 2, \ldots, n$. You are to compute the height at which the rocket was right when the last stage fell off. Our inputs will always be such that the rocket will lift off, and not crash into the Earth before the last stage is discarded.

Input

The first line contains a number $K \geq 1$, which is the number of input data sets in the file. This is followed by K data sets of the following form:

The first line of each data set contains the number of stages n (between 1 and 30), and the weight M of the rocket itself. $0.01 \leq M \leq 1,000,000$ is a floating point number.

This is followed by n lines, each consisting of three non-negative floating point numbers $0.01 \leq m_i, t_i, F_i \leq 1,000,000$, describing a stage. So the initial weight of the rocket is $M + \sum_i m_i$.

Output

For each data set, first output "`Data Set x:`" on a line by itself, where `x` is its number. Then, output the height at which the rocket was when the last stage fell off, rounded to two decimals.

Each data set should be followed by a blank line.

Sample Input/Output

Sample input `fall06.in`

```
1
3 1000.0
325.0 120.0 18000.0
75.0  60.0 5100.0
200.0 120.0 15000.0
```

Corresponding output

```
Data Set 1:
8550.0

```

Problem E: Spring 2007: Final Exams

File Name: `spring07.cpp|spring07.java`

Input File: `spring07.in`

Description

In Spring of 2007, we were a little late in scheduling the contest, so that it ended up being on a weekend during study days. To "help" students make up for the time they were wasting at the contest, we called it the "Final Exams" Contest, and had all problems have study and exam themes. One of the problems asked students to figure out optimal ways to allocate study time across different courses.

Given that study time is rather limited, it is crucial to divide it up well for studying for various exams. In the end, you just want to maximize your GPA, so maybe it's worth to put up with a C in one course (where a better grade would require a lot of studying) to secure A's (which can be achieved with relatively little studying each) in all other courses. Clearly, this is a crucial optimization problem, and solving it well will probably benefit you tremendously.

Input

The first line contains a number $K \geq 1$, which is the number of input data sets in the file. This is followed by K data sets of the following form:

The first line of a data set contains two numbers n, H, the number of courses you are taking (an integer n between 1 and 10) and the number of hours available for studying (an integer $H \leq 100$).

This is followed by n lines, each containing 10 integers. These integers in line i will describe, in the order given, the number of hours you need to study in order to get an A (4.0), A- (3.7), B+ (3.3), B (3.0), B- (2.7), C+ (2.3), C (2.0), C- (1.7), D+ (1.3), D (1.0) in the i^{th} of your courses. The numbers will be non-increasing. The interpretation is that if you study at least that many hours in the particular subject, you will get the given grade. If you don't even study enough for a D, your grade will be F (0.0).

Output

For each data set, first output "`Data Set x:`" on a line by itself, where `x` is its number. Then, output the maximum possible GPA (rounded to two decimals) you can obtain by dividing your H hours of studying between the subjects.

Each data set should be followed by a blank line.

Sample Input/Output

Sample input `spring07.in`

```
1
3 60
40 37 35 33 30 26 20 10 5 1
10 10 10 10 10 10 10 10 10 1
24 23 22 21 20 20 20 20 20 20
```

Corresponding output

```
Data Set 1:
3.43
```

Problem F: Fall 2007: The Programming Contest Programming Contest

File Name: `fall07.cpp|fall07.java`

Input File: `fall07.in`

Description

Finally, in Fall of 2007, we went meta, and had a "Programming Contest Programming Contest", in which the participants had to solve problems about runnning a programming contest. Clearly, the most important part of the programming contest is the pizza. The pizza has delicious soy nuggets on it, but it isn't cut yet. So we have to cut it into slices. There are two important constraints we have to meet: every slice must have the same size, and every slice must have the same number of soy nuggets on it. To make things worse, the pizza is circular, and we can only cut along a straight line from the center of the circle to its circumference. And we can never cut through a soy nugget, it must always be on one side of the cut or the other.

You are to write a program that, given the positions of the soy nuggets, determines which is the largest number of slices into which the pizza can be cut such that every piece has the same size and the same number of soy nuggets (notice that a pizza can always be cut into 1 slice — so there always is a solution.)

Input

The first line is the number K of input data sets, followed by K data sets, each of the following form:

The first line contains the number N of soy nuggets on the pizza ($1 \leq N \leq 200$). This is followed by N lines, describing the positions of the soy nuggets. Each position is given in **polar coordinates**, measured from the center of the pizza. That is, a position is a pair "α r", where $0 \leq \alpha < 2\pi$ is the counterclockwise angle between the positive x-axis and the line from the pizza center to the soy nugget. $0 < r \leq 1$ is the distance of the nugget from the center (each pizza will have radius 1). We assume that nuggets are points, and no two nuggets will be lying on the same spot of the pizza.

Output

For each data set, output "`Data Set x:`" on a line by itself, where `x` is its number.

Then set, output the maximum number of slices into which it can be cut according to the above rules. Remember that no cut can go through a soy nugget.

Each data set should be followed by a blank line.

Sample Input/Output

Don't worry about the grammar in the output.

Sample input `fall07.in`

```
2
2
1.57 0.5
1.57 0.7
4
0.7 0.9
1.5 0.1
1.8 0.5
3.05 1.0
```

Corresponding output

```
Data Set 1:
1 slices

Data Set 2:
2 slices

```