

The Programming Contest Programming Contest

USC Programming Contest, Fall 2007

September 15, 2007

Sponsored by VSoE, Google, Northrop Grumman, and Electronic Arts

You wouldn't believe just how much work it is to organize a programming contest. You have to come up with problems, organize helpers, judge solutions, provide food, etc. As a participant, it's also quite a lot of work: coming up with ideas, coding them up, getting frustrated, eating all the food, taking home prizes, etc. All our work could be simplified a lot with the right tools. So in an effort to have smoother contests in the future, you'll be solving some programming contest problems.

You can solve or submit the problems in any order you want. When you submit a problem, you submit your source code file using `PC2`. 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: several problems ask you to print numbers rounded to two decimals. In C, you do this via `printf("%.2f", r);` where `r` is the variable you want to print. And in Java, the syntax is `System.out.print((new java.text.DecimalFormat("#.##")).format(r));`

[Including this page, the problem set should contain 8 pages. If yours doesn't, please contact one of the helpers immediately.]

Problem A: Got problems?

File Name: problems.cpp|problems.java

Input File: problems.in

Description

In preparation for the contest, we came up with a number of algorithms and programming topics we want to check, such as graph algorithms, geometry, searching, sorting, string processing, etc.

Also, we made up a number of problems (one of which you're looking at), and for each problem determined which categories it falls into. A problem will always belong to at least one category, but it can belong to several categories, potentially even all of them.

Now we want to find a set of problems that will cover all categories. And as good problems are hard to make up and therefore precious, we want to use as few problems as possible. You are to write a program that will save us from doing this by hand.

Input

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

The first line of a data set contains two numbers, M and N ($1 \leq M, N \leq 20$). M is the number of problem categories, and N is the number of available problems. Problem categories are denoted by $1, \dots, M$, and problems by A, B, \dots . The following N lines each contain the numbers of the categories covered by a problem — the first line for problem A , the second one for B , and so on.

Output

The output file contains one line for each input, separated by an empty line. Each line contains first the number of the data set, and then the solution. The solution is the list of problems to be used, separated by whitespace, and sorted alphabetically.

You may assume that there is always a solution, but it need not be unique. If there are several best solutions (i.e. of equal length), output the one which comes alphabetically first in the output format just described.

Sample Input/Output

Sample input `problems.in`

```
2
2 3
1
2
1 2
4 5
1 2 3
1 2
2 4
1 3 4
2 3 4
```

Corresponding output

```
Data Set 1: C
Data Set 2: A C
```

Problem B: Contest Seating

File Name: seat.cpp|seat.java

Input File: seat.in

Description

The participants start arriving, are taking seats at the computers. After practicing a bit, many of them leave temporarily in order to grab some snacks. Of course, this sometimes results in a new participant trying to use an already claimed computer. We want to figure out just how often this happens.

Input

The first line will contain the number K of input data sets. This is followed by the K data sets, each of the following format:

The first line contains two numbers P and M , the number of participants ($1 \leq P \leq 500$), and the number of machines ($1 \leq M \leq 500$). This is followed by P lines, each containing an integer between 1 and M , the number of the machine that the participant is trying to sit down at first.

Output

For each data set, first output its number, then in a new line the total number of unsuccessful sitting attempts. An attempt is unsuccessful if by the time participant i is trying to sit at machine j , some other participant has already sat at machine j .

Sample Input/Output

Sample input seat.in

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

Corresponding output

```
Data Set 1:
3
Data Set 2:
0
Data Set 3:
2
```

Problem C: Format Error!

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

Input File: `format.in`

Description

Those of you who participated in past contests probably still remember them with pain, and the new ones will soon find out what we're talking about: Format Errors. You write a wonderful correct solution and submit it, and Bang!, you get the result "Format Error!" from the judges. Just because you had one blank too much, or accidentally typed the wrong kind of parentheses.

Of course, we are aware that this is painful, but we don't want to judge all these solutions with lots of output by hand — you'd never get the results. However, if we had a program to do the job, things might look a lot better. Guess what! You get to write us such a program.

The program will compare two strings at a time, and decide whether they are "equal" (in a relaxed sense) to one another. Strings will consist of uppercase and lowercase letters, digits, whitespace (no tabs), and the following punctuation marks (no others will appear in our test cases):

`() [] { } . , ; :`

The following are the rules according to which strings will be considered equal:

- Upper and lowercase letters can be interchanged.
- The size of a block of whitespace is unimportant (this of course applies only to blocks consisting of at least one whitespace). Whitespace immediately following or preceding punctuation marks is unimportant. Leading or trailing whitespace (i.e. at the beginning or end of strings) is unimportant.
- The various kinds of opening brackets can be interchanged. The various kinds of closing brackets can be interchanged (however, an opening bracket can not be interchanged with a closing bracket). Brackets **need not** match.
- Comma (",") and semicolon (";") can be interchanged.

Input

The first line is the number K of data sets in the file. This is followed by the K data sets, each consisting of two strings s_1 and s_2 (of length at most 1000 characters each) on separate lines. The newline character is not considered part of the string.

Output

For each data set, output its number, and either "equal" or "not equal", depending on whether s_1 and s_2 represent the same string for this data set. Output for different data sets should be separated by an empty line.

Sample Input/Output

Sample input `format.in`

```
3
(1, 4) (2, 3) (2, 4)
{1;4} {2;3} {2;4}
Data Set 1: equal
data set 1 : EQUAL
Data Set 1: equal
DataSet 1: equal
```

Corresponding output

```
Data Set 1: equal
Data Set 2: equal
Data Set 3: not equal
```

Problem D: Time Limit Exceeded

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

Input File: `limit.in`

Description

“Time limit exceeded” is another favorite error message. Most frequently, the cause is either an infinite loop somewhere in your program, or — more often — a very slow brute-force solution to a problem that should be solved more elegantly. The problem is not that we terminate the program after 1 minute; your solutions would probably run 10^{100} years if they use brute-force for a problem where they shouldn't.

The whole thing might be a little easier if you had a little program to figure out the runtime complexity of your code. In general, this is an undecidable problem (i.e., there cannot be any computer program always getting it right, one of the cool results you learn in CSCI 303). So instead, we will restrict our attention to a severely restricted “programming language” for which the problem can be solved again. Programs only consist of four kinds of statements, namely **basic**, **loop**, **endloop** and **endprogram**. Each statement is on a line by itself, there are no semicolons or any other punctuation, and **loop** is the only command with a parameter. That parameter is either a positive integer number, or one of the two variables ‘**x**’ and ‘**y**’ (our simplified programming language has only two variables), denoting that the body between the **loop** and the corresponding **endloop** is executed that many times.

basic denotes a basic statement (like assignment, addition, or some other stuff that does not matter for complexity analysis), and **endprogram** is the end of the program. Notice that the number of times a loop is executed is always a constant (either the variable or a numerical one), since variables cannot be assigned values. Also, we assume that our compiler is smart enough to prune away loops with no **basic** statement anywhere in their bodies.

You are to determine, in big-Oh (O) notation, the complexity of the program, i.e. the total number of times that any **basic** statement is executed. Your program should output an asymptotically tight bound of the form $\Theta(\dots)$.

We say that a function f is $\Theta(g)$ for another function g , if there are positive constants c and d with $c \cdot g(x) \leq f(x) \leq d \cdot g(x)$ for all $x \geq 1$. In practice, that means that you can drop all constant factors (e.g. $4x^3$ is $\Theta(x^3)$), and all terms that have smaller exponents for every variable (e.g. $x^3 + x^2$ is $\Theta(x^3)$, because $x^3 \leq x^3 + x^2 \leq 2x^3$ for all $x \geq 1$). For some more examples, $x^2 + 7$ is $\Theta(x^2)$, and $x^2y + y^2x + xy + x^2$ is $\Theta(x^2y + y^2x)$.

A slightly less obvious consequence is that certain mixed terms can be omitted as well. For instance, $x^2 + 17xy + y^2$ is $\Theta(x^2 + y^2)$, because either $x \leq y$, which implies that $x^2 + 17xy + y^2 \leq y^2 + 17y^2 + y^2 \leq 19(x^2 + y^2)$, or $x \geq y$, which implies that $x^2 + 17xy + y^2 \leq x^2 + 17x^2 + y^2 \leq 19(x^2 + y^2)$. If you have questions about the notation, please ask for assistance.

Input

The first line is the number K of input data sets, followed by K data sets, each being one program (and each followed by an empty line).

A program consists of only the four commands above, and will always be syntactically correct (e.g. the **loop** and **endloop** commands match, there is exactly one **endprogram** outside the outermost **loop** and so forth). Each command will be on a line by itself. There will be exactly one whitespace between the string **loop** and its parameter, and no tabulator or whitespace characters anywhere else (besides the line breaks). The nesting depth of **loop** statements will be at most 50 in all examples.

Output

For each data set, output “**Data Set x:**” on a line by itself, where x is its number. On the next line, output the asymptotic complexity of the program. The order of the terms should be by non-increasing exponents of **x**, breaking ties by non-increasing exponents of **y**. Make your output look “mathematically neat”, e.g. output xy instead of x^1y^1 and x instead of x^1y^0 . The output for each data set should be followed by an empty line.

Sample input limit.in

```
3
loop x
endloop
endprogram

basic
basic
endprogram

loop y
basic
loop y
basic
basic
endloop
endloop
loop x
loop y
basic
endloop
loop x
basic
endloop
endloop
endprogram
```

Corresponding output

```
Data Set 1:
0

Data Set 2:
1

Data Set 3:
x^2 + y^2
```

Problem E: Who gets the prizes?

File Name: prizes.cpp|prizes.java

Input File: prizes.in

Description

The contest is over, everybody is disappointed at having screwed something up, and now, the judges have to look at hundreds of solutions, correct and incorrect, and determine who has won. How they wish they had a program to do that!

Judging will go exactly by the rules of the contest — if you are not sure about the exact rules, you should look them up on the “rules” handout that you received. You can assume that in all our test cases, no two participants will simultaneously have the exact same number of problems solved and exact same total time.

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 contains three number, M , N , and P . M is the number of problems ($1 \leq M \leq 10$), N is the total number of submissions ($1 \leq N \leq 5000$), and P is the number of participants ($1 \leq P \leq 500$). The problems are numbered A, B, \dots , participants are numbered $1, \dots, P$.

This line is followed by N lines, each describing one submission. Each line is of the form “ $p m t j$ ”. Here, p is the number of the submitting participant, m the identification letter of the problem (i.e. A, B , and so on), t the time (in minutes into the contest) at which the submission took place, and j the judgment, a 1 for a correct submission, and a 0 for an incorrect one. The lines will be sorted by non-decreasing submission times.

Output

The output for a data set is the number of the data set, and the list of all participants sorted according to their ranking. For each participant, it should list the number of problems solved and the total time (including penalties). The output for different data sets should be separated by an empty line.

Sample Input/Output

Sample input prizes.in

```
2
4 6 3
2 C 11 0
1 B 24 1
2 C 25 1
1 C 40 0
3 D 55 1
2 B 55 1
2 8 2
1 A 8 0
2 B 11 1
1 A 11 0
1 A 14 0
1 A 20 0
2 B 22 1
2 A 23 0
1 A 24 1
```

Corresponding output

```
Data set 1:
2 2 100
1 1 24
3 1 55

Data set 2:
2 1 11
1 1 104
```

Problem F: Pizza!

File Name: pizza.cpp|pizza.java

Input File: pizza.in

Description

Finally, we get to the most important part of the contest: Eating the pizza. But even here, problems are lurking! 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 its number, followed by 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. Output for different data sets should be separated by an empty line.

Sample Input/Output

Don't worry about the grammar in the output.

Sample input pizza.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
```