The ACSU Event Programming Contest

Cornell ACSU Programming Contest, Spring 2002

02/24/2002

Sponsored by Microsoft

In case you have not noticed yet, the ACSU does a lot more than just organize a programming contest once or twice a year. They have many social events offering computer science majors chances to bond and get to know each other. They bring in lots of companies for Info Sessions, maintain a resume book, and form a link between students and the department. In order to do so, they rely on volunteers who spend a lot of their time for the benefit of others.

In this spring's contest, we will explore some of the events that the ACSU organizes, and solve typically encountered computational problems. If you solve them well, you might become honorary members.

As a change over the fall contest, we will allow you to use *any programming language* you like. In fact, you are welcome to solve all problems by hand (although we don't recommend that approach). You don't submit your program to us, you only submit your output file, and we don't care how you computed it (nor does it matter how you name it). Except of course you may not use any other electronic devices or a web browser.

This problem set does not contain any sample input or output. For each of the six problems, you should have a directory, which contains files `sample.in`, `sample.out` and `realdata.in`. The `sample` files are there solely to show you the exact input and output format, and to give you some easy cases to test your program on. `realdata.in` is the problem set you will have to solve to get your solution judged correct.

One warning that seems to be in place after past contests: all of our numbering (input data sets, persons, 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, this problem set should contain 4 double-sided pages. If yours doesn't, please contact one of the helpers immediately.]

Problem A: Bowling

Description

Bowling is a favorite activity of many Americans. You can make quite a number of jokes about the regular attendants of bowling alleys[1], but this does not alter the fact that an evening bowling with fellow engineers can be quite entertaining, in particular if everyone keeps falling over because of the slippery shoes. But what if you completely forgot who scored how many pins, and just have the sequence of scores available? Clearly, you will need a program to reconstruct the current score.

You play bowling by rolling a heavy ball down the bowling alley, and hitting some or all of initially 10 pins. If the ball hits the side wall of the alley or you commit some other kind of foul, your throw counts as having hit 0 pins. A game consists of a number of "frames" (usually 10). In each frame, every player gets up to two throws. If a player knocks over all pins with one throw, she gets ten points, and a "strike". She does not get to throw again in this frame (there are perks to it to make up for that). If a player knocks over between 0–9 pins on the first throw, he gets a second throw at the remaining pins (the other ones are not restored). The total number of pins knocked over with the two throws is the score for that frame (notice that the total can be at most 10 — if a sequence of numbers claims otherwise, it is inconsistent). If that score was 10, the player gets a so-called "spare". Once the player is done with his two throws (or one throw, in case of a strike), all pins are restored for the next player.

When a player has a spare, it means that the result of the next throw (made in the next frame) will be added to that frame. For example, you throw 8+2 in the first frame, and 6+1 in the second. Then, your score for the first frame is 8+2+6, and for the second, it is 6+1, so your total is 23 so far, and you have no spare for the third frame. Had you had 1+6 in the second frame, your total would only be 8+2+1+1+6 = 18.

When a player has a strike, the results of the next *two* throws will be added to that frame, independently of when they are made. For example, if a player has a strike in the first frame, and 6+1 in the second frame, then the total is 10+6+1 for the first frame, and 6+1 for the second, i.e. 24. If the player has strikes in the first two frames, and 6+1 in the third, the total is 10+10+6 for the first frame, 10+6+1 for the second, and 6+1 for the third, i.e. 50.

Input

The first line will contain the number K of input data sets, followed by K data sets of the following form (each data set is followed by an empty line):

Each data set first contains a line with number p and n. p is the number of players ($1 \leq p \leq 10$), and n the number of throws considered ($1 \leq n \leq 1000$). This is followed by n lines, each containing one number s between 0 and 10, inclusive, denoting the number of pins knocked over in that throw. The sequence will be a *partial* sequence, up to a certain point of a game. That is, the current frame might not be finished (some players might not have obtained the chance to throw yet).

Output

For each data set, first output "`Data Set x:`" on a line by itself, where `x` is its number. If the sequence of numbers is inconsistent, output "Inconsistent" on a line by itself. Otherwise, output the scores for the p players on one line, separated by a single space. Do not output any information regarding whether players have pending strikes or spares. After the output for each data set, output an empty line.

[1] and if you do, you usually have a good point

Problem B: Ice Skating

Description

Ice Skating is a very popular sport at Cornell, probably because the climate[2] is rather conducive to water freezing. The way ice skating is practiced is that you go to the rink, pay a few dollars to be let in, then another few dollars to get a pair of skates which is way too tight or too large, and then get on the ice, where you will be knocked over by faster skaters. Of course, the latter is entirely unnecessary, since falling over is the easiest part of skating.

When you go skating with ACSU, the entry and rental fees are usually taken care of. We can't do much about the falling over part right now, but we could do something about the skate sizes. Suppose that we knew the exact foot sizes (in centimeters) of everyone who wants to go skating, and the exact sizes of skates available. Then, we could find an assignment of skates to people that would minimize the total collective pain. Of course, you are to write a program doing just that.

If we are given a foot size f and a shoe size s, both in centimeters, the "pain index" π is calculated as $\pi = (f - s)^4$ if $f \geq s$, and $\pi = s - f$ if $f < s$. The collective pain index Π is the sum of the pain indices of all people. Given all shoe and foot sizes, you are to determine the smallest collective pain index attainable by any assignment of people to skates. Of course, each person only wears one pair of skates, and each pair of skates is only worn by one person.[3]

Input

The first line will contain the number K of input data sets, followed by K data sets of the following form (each data set is followed by an empty line):

Each data set first contains a line with integers p and n. p is the number of people skating, and n the number of pairs of skates available ($1 \leq p \leq n \leq 500$). This is followed by p lines, each containing a floating point foot size f with $20.0 \leq f \leq 60.0$, and n lines, each containing a floating point shoe size s with $20.0 \leq s \leq 60.0$.

Output

For each data set, output "`Data Set x:`" on a line by itself, where `x` is the number of the data set. On the next line output the smallest collective pain index Π for that data set, rounded to two decimals. After the output for each data set, output an empty line.

[2] see Fall 2001 Programming Contest

[3] Duh!

Problem C: Senior Cruise

Description

For the graduating seniors, the ACSU organizes boat trips on Cayuga Lake. You get onto a boat, and cruise around the lake for a while, obtaining new perspectives on Cornell University or Ithaca. At least in the literal sense. After you have gained enough new perspective, you turn your attention to different things, such as throwing stones into the water.

Even when we were kids, we would observe that when you throw a stone into water, it makes waves that travel away from the impact point in larger and larger circles. Now that we are engineers, we want to know how long it takes for the wave to reach the shore.

We assume that water waves move with a constant speed of 1 meter per second from the point at which the stone hit the water outward. The lake is represented as a simple polygon.

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 (each data set is followed by an empty line):

The first line contains three numbers, n, x and y. The coordinates x and y are given in meters (with respect to some arbitrary origin), and are floating point numbers. They specify the point at which the stone hits the water. $1 \leq n \leq 1000$ is the number of points of the polygon describing the lake.

This is followed by n lines, each containing two floating point numbers x_i, y_i, giving the ith point of the polygon in meters (with respect to the same origin). Points are given in counter-clockwise order.

The polygon will always be simple, and the point (x, y) will always lie strictly inside the polygon.

Output

For each data set, output "`Data Set x:`" on a line by itself, where `x` is the number of the data set. On a new line, output the earliest time (in seconds) at which the wave reaches the shore at any point of the lake. Output should be rounded to two decimals. The output for each data set should be followed by an empty line.

Problem D: Elections

Description

In order to organize all these nice events for you, the ACSU has a number of volunteer officers, such as a president, treasurer, web master and so forth. These officers are elected every year, and — who knows — you could be one next year.

There are usually not so many individuals around to have real competition for all offices, it's more a question of who should get which job. Maybe, the following voting scheme would be useful, once we implement a program to do the work.

Each individual has a list of all the offices he or she is running for. Then, we cast votes for each office in turn. Now, we look for the clearest winner, i.e. the person i who had the highest percentage for one of the offices j (we break ties by favoring first the lowest candidate number i, and then the lowest office number j). Person i gets office j, and is removed from the candidate lists for all other offices (and all other candidates remove their bid for office j, of course). That means that all votes that were cast for candidate i or any candidate for office j are treated as though they had never been cast. Among the remaining candidates, we look for the one with the highest percentage for some office, and iterate until there are no more choices left.

This will ensure that no one gets more than one office, and it's always the remaining person with the highest vote who wins an office — people who were even more qualified for another office excluded. Of course, this runs the risk of leaving offices unassigned even when there were candidates (because they ended up with different jobs). Hey, there are hazards to every voting system, as we found out in Florida.

Input

The first line is the number K of input data sets, followed by the K data sets, each of the following form (each data set is followed by an empty line):

The first line contains two numbers, p and n. $1 \leq p \leq 100$ is the number of candidates for offices, and $1 \leq n \leq 100$ is the number of offices. This is followed by p sets describing one candidate each.

A candidate set for candidate i ($1 \leq i \leq p$) has as its first line a number m ($1 \leq m \leq n$) which is the number of offices the candidate is running for. This is followed by m lines, each containing two numbers j and v. $1 \leq j \leq n$ is the number of the office, and $v > 0$ the number of votes that candidate i received for office j. You may assume that there will never be two entries with identical j-values for the same candidate i.

Output

For each data set, output "`Data Set x:`" on a line by itself, where `x` is its number. Then output the list of all office assignments, as follows. Print n lines, each containing first the number j of the office (in ascending order), and then the number i of the candidate who got it. If an office remains vacant, print "`vacant`" instead of a candidate number. Each data set should be followed by an empty line.

Problem E: Info Sessions

Description

Info Sessions are a great way to get free pizza, and win PalmPilots. As a side effect, you might even end up with a job. Every semester, there are lots of them, and the companies have wide ranges, including computing, consulting, engineering, finance. A lot of the info sessions are organized by the ACSU.

After you have gone to a lot of info sessions, you often end up getting them mixed up. Was IBM the one with Pepperoni Pizza or Mushrooms? And did Boston Consulting give out PalmPilots or DreamCasts again? Maybe, by looking at the complete schedule, and comparing with your fragmentary memory, you will be able to reconstruct which info sessions you attended.

Each info session is characterized by five parameters: the pizza topping, the goodies raffled away, the company name, the company's field, and the number of buzzwords in the job description.

For each info session you attended, you remember a (possibly empty) subset of these parameters. Your program should determine, for each fragmentary memory set, all candidate info sessions that your memory could be referring to.

Input

The first line is the number K of input data sets, followed by the K data sets, each of the following form (and each followed by an empty line):

The first line contains two numbers n and m. $1 \leq n \leq 50$ is the number of info sessions that were held during the semester, and $1 \leq m \leq n$ is the number of info sessions you attended.

This is followed by n info session descriptions, each consisting of five lines. The five lines contain (in this order) the pizza topping, goodies, company name, company field, and buzzword number for that info session. Company names need not be distinct, since one company could have had several info sessions in one semester.

After this, there are m memory descriptions, each consisting of five lines. The order of the lines is the same as above — except that there is the additional possibility that a line says "unknown" which means that you cannot remember that parameter. (The string "unknown" will never appear as part of the n info session descriptions.)

You can assume that the spelling of company names is always consistent, i.e. we will not be writing "IBM" once and "ibm" another time, or "Hewlett Packard" once and "Hewlett-Packard" another time when referring to the same company.

Output

For each data set, first output "`Data Set x:`" on a line by itself, where `x` is its number. Then output m lines, one for each memory description. That line should contain a list of the numbers of all info session descriptions (in increasing order) that match the memory fragment, separated by a single space. If there are no matches at all for a given memory fragment, output "`no match`" instead. The output for each data set should be followed by an empty line.

Problem F: Programming Contest

Description

Of course, the best event of the year is always the Programming Contest — everyone is looking forward to it all year long[4]. But why do the problems have to be such that when you write down the natural solution, it runs for 10^{100} years? If at least they provided you with a program to test the complexity of your code!

In general, this is of course undecidable, so we will restrict our attention to some simpler form of programs. 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 $\Theta(\ldots)$.

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.

[4]The author is aware of the irony here.