2009/2010 SOUTHERN CALIFORNIA REGIONAL
ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST

Problem 1
Reverse Engineering Alchemy

Frederico da Vinci (one of Leonardo's less talented cousins) operated a successful Alchemy Emporium in Venice for many years. Recently a competing shop, Borgia Potions & Notions, began selling an entire line of particularly effective alchemical potions.

"I tell you," said Frederico's Chief Alchemist, "if we only knew the basic ingredients and their proportions, I'm sure we could figure out a cheaper way to produce the same potions." Frederico smiled and said, "As it happens, I have bribed a clerk at Borgia's. He does not know how the potions are made, but he is responsible for ordering the raw ingredients and for stocking the resulting products in the shop. He sends me regular messages telling me how many units of various ingredients they have used and how many doses of each potion they produced from those ingredients."

Frederico showed the alchemist a small piece of paper on which was written "A 23 B 17 f 15 h 19 q 47." The alchemist asked "What is this?" Frederico explained, "We had to keep the messages short and so we agreed on a code. The upper-case letters stand for potions that were produced and added to the store's stock. 'A', for example, is their Guaranteed Wart Remover, and he is telling us that they have created 23 doses of it. The lower-case letters stand for the ingredients used to make the potions. 'f', is, let me think, ah yes, powdered lodestone. Apparently they ordered 15 vials of that. Now see what you can make of these."

Frederico handed several of the clerk's messages to the alchemist, along with the key explaining the letters used for each potion and ingredient. The alchemist studied them for some time, then said, "I think I know how they make their love potions, but we'll need more reports to figure out any of the others."

Your team is to write a program that processes the clerk's reports and determines the potion formulas that can be derived from those reports.

Input consists of several independent data sets. Each data set is terminated by an empty line. The final data set is followed by the end-of-file.

Each data set consists of one to thirty reports. Each report is written on a single line and consists of two or more pairs. Each pair consists of a single letter denoting a potion or ingredient followed by a quantity as an integer in the range 1 to 200. Letters and integer values are separated from each other by single spaces. Each report will contain at least one potion and one ingredient. Each input set will be free of contradictions and can be resolved without requiring negative numbers in the solution.

Your program is to produce a brief report for each data set. The report is to contain one line for each potion mentioned in the data set, with the potions listed in alphabetical order. If the formula for the potion is known, print the number of doses the formula produces, followed by a single space, the potion letter, and a colon. The potion ingredients are to follow in alphabetical order. For each ingredient, print a single space, the units of the ingredient needed, another space, and the letter of the ingredient. All results are to be expressed as reduced integer values—the entire set of integers appearing on one output line should not have a common divisor greater than one. Do not list ingredients for which zero units are required.

Should it not be possible to determine the formula for any potion listed in the data set, print that potion followed by a colon, a space, and the word "indeterminate".

No leading or trailing space is to appear on an output line.

The printed output for each data set (including the last) is to end with an empty line.

Problem 1
Reverse Engineering Alchemy (continued)

Sample Input

```
A 36 a 6

A 6 B 2 C 1 Q 2 a 19 c 3 d 3 z 1
C 1 B 2 a 6 d 2 z 1
A 3 B 4 C 2 Q 4 a 26 c 3 d 6 z 2
A 9 B 6 C 3 d 9 a 45 c 6 z 3 Q 6
```

Output for the Sample Input

```
6 A: 1 a

3 A: 4 a 1 c
B: indeterminate
C: indeterminate
2 Q: 5 a 1 c 1 d
```

2009/2010 SOUTHERN CALIFORNIA REGIONAL
ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST

Problem 2
Serpentine Belt

A serpentine belt system connects a series of pulleys with a single belt rather than connecting pulley pairs with many belts. There are many advantages to such a system such as lighter weight and simplified maintenance. See figure 1 for an example.

Your team is to write a program that will calculate the length of a serpentine belt given the (x, y) locations of the centers of the pulleys and their radii.

The input is a series of lines terminated by end-of-file. Each line represents one pulley and the line order is the order in which the belt connects them. The belt from the last pulley connects back to the first. The first pulley is the drive pulley and hence determines the overall direction of the belt.

Each line consists of three integers separated by whitespace. The first is the x position of the center of the pulley, the second is the y position. The third is the radius of the pulley. If the radius is > 0 then the pulley is on the inside of the belt. If < 0, it is outside. All positions and dimensions are in millimeters: $0 \leq x < 1000$; $0 \leq y < 1000$; $|\text{radius}| > 0$.

There will be no pathologies in the input: the belt will not self intersect; the belt will connect with each pulley only once and with a wrap angle > 0; the pulleys will not intersect. There will be at least 3 pulleys but fewer than 50.

Your program is to print one line containing the length of the belt to the nearest millimeter, starting in the first column and without trailing whitespace.

The following belt equations may prove useful:

For an open belt pair of pulleys (see figure 2):

$$R1 \geq R2$$
$$\theta = \arcsin((R1 - R2)/C)$$
$$L = \sqrt{C^2 - (R1 - R2)^2}$$

For a crossed belt pair of pulleys (see figure 3):

$$\theta = \arcsin((R1 + R2)/C)$$
$$L = \sqrt{C^2 - (R1 + R2)^2}$$

Sample Input

```
100 100 60
150 250 -10
50 350 50
350 350 60
520 300 30
450 200 -50
500 100 20
300 150 -40
```

Problem 2
Serpentine Belt (continued)

Output for the Sample Input

```
1988
```

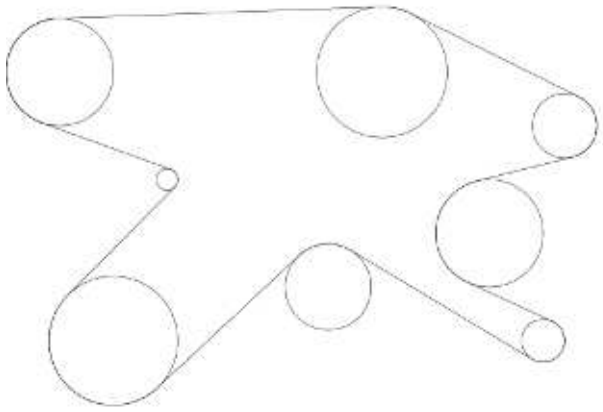

Figure 1. Diagram of the Sample Input.

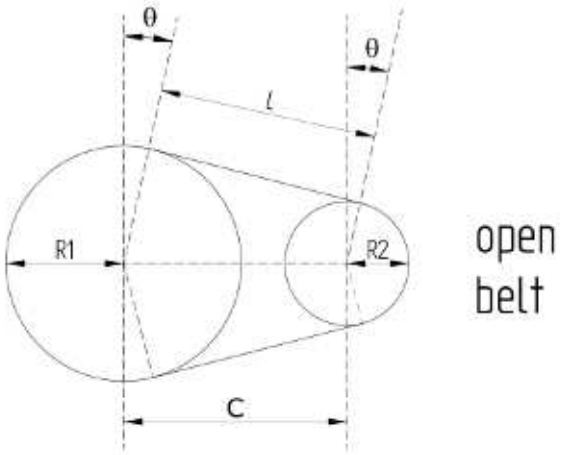

Figure 2. Open belt pair of pulleys.

Problem 2
Serpentine Belt (still continued)

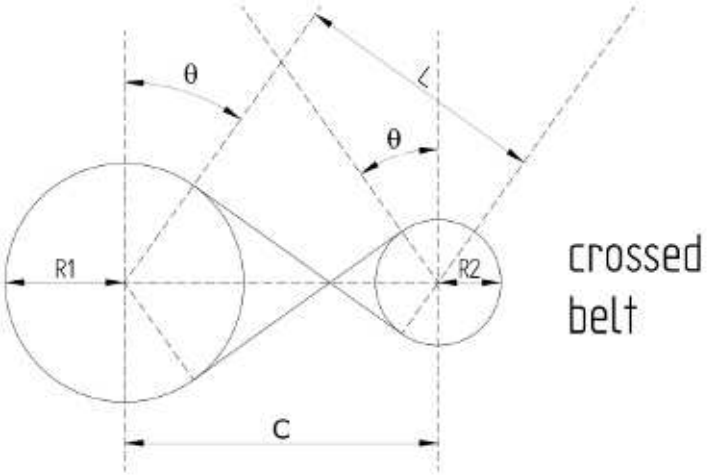

Figure 3. Crossed belt pair of pulleys.

2009/2010 SOUTHERN CALIFORNIA REGIONAL
ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST

Problem 3
Bumper Boat Autopilot

This is an interactive problem.

Many amusement parks have motorized bumper boats with simple controls: usually a throttle that is either on or off, and a steering mechanism. Despite these common controls, there is considerable variation in their behavior. Some bumper boats have propellers in the back, others might have a front-mounted nozzle that points backwards. Steering mechanisms may be rudders in the back or a steering wheel integrated with the motor. You are to write a program that learns how to steer a bumper boat of unknown configuration, then guide it to a series of coordinates (known as waypoints) within a time limit. Before being expected to steer a bumper boat, your program will be able to train itself in trial runs. After completing your trial runs, you will then be given a list of waypoints and a time limit.

Your bumper boat floats on an infinite lake. Whenever you start a run (trial or judged course), the boat begins at coordinate (0,0) facing north, with the steering mechanism "centered" at zero degrees. The boat starts with zero translational and zero rotational velocity. During the trial runs and the actual test course, your program will issue control commands to the server; each command corresponds to one tenth (0.1) of a second of simulated time.

You begin a run with the server by issuing a C command to query the configuration of the boat. The server responds with a line containing a single integer, d, indicating the range of positions $[-d, d]$ in degrees that the steering mechanism can be turned.

After querying the configuration, you issue a T (trial) or J (judge) command. The server responds with a line reporting the time remaining, in seconds. The J command will respond additionally with a set of waypoints in the form

W
w_{x1}, w_{y1}
w_{x2}, w_{y2}
...
w_{xW}, w_{yW}

where W is the number of waypoints and w_xn,w_yn form an x,y coordinate for each waypoint. W will never exceed 20. You then issue control commands of the form:

E,<steering position>

or

I

where E indicates that the throttle is engaged, and the steering position as degrees (floating point) from the centered steering position (clockwise is positive, counterclockwise is negative). I indicates idle (throttle not engaged). When idle the steering mechanism has no effect.

In trial mode (T command), the server responds to E or I with a line of the form

x y h

whereas in judge mode, the server responds with a line of the form

x y h w_i

where x, y, and h report the position and orientation of the boat 0.1 second *after the issuance of the command.* x and y are the current (x, y) coordinates, in meters, of the center of the boat, and h is the heading, in degrees, of the front of the boat. 0 degrees is north. Negative headings measure degrees counterclockwise from north, and positive headings measure degrees clockwise from north. South is reported as positive 180 degrees.

Problem 3
Bumper Boat Autopilot (continued)

Note that a boat may not be moving in the direction it is facing. w_i is the highest waypoint number encountered thus far. When $w_i = 0$, it indicates no waypoints have been passed yet. It is possible to pass multiple waypoints in a single 0.1 second interval. If the multiple waypoints are consecutive and are the next successive waypoints, then they will be considered in order, even if the exact boat path passes them out of order.

To start a new trial run, issue a T command rather than a control command. The time remaining returned from each successive T command counts down cumulatively from a maximum allotted trial time. To start a judged run, issue a J command rather than a T or control command. The time limit returned by the J command is distinct from the trial time, and is based upon the boat and the geometry of the waypoints. As soon as your program issues a J command, it can no longer issue a T command. Finally, when your program is finished with its judged run, issue a Q command.

To follow the waypoints, your boat must come within 0.5 meters of each waypoint, in the order they are given. It is okay to wander to other waypoints out of order, just so long as any sequential subset of the waypoints followed satisfies the specified list. For example, if you are directed to follow waypoints w_1 through w_{10}, the following sequence of waypoints satisfies the requirement:

$w_3, w_5, w_1, w_2, w_5, w_3, w_6, w_6, w_4, w_5, w_{10}, w_6, w_7, w_8, w_9, w_1, w_{10}$

Your program will not know the control configuration of the bumper boat, but any one conversation with the server will always be with only one boat. Your program will be executed several times, each time against a different boat and/or a different set of waypoints. A typical execution should consist of many trials (limited by the maximum allotted trial time), followed by a judged attempt.

The waypoints are positioned well within the capabilities of the boat configuration, and an adequate client implementation will not require fancy loops or maneuvers to negotiate tricky turns. You might find it useful to structure your training runs to (1) move in a straight line and determine the maximum speed, (2) turn left or right, (3) transition from straight motion to turn, and determine minimum turn radius at maximum speed, and (4) transition from a turn to straight motion.

You have access to the same server process that the judges use. Use the `getdata` command to retrieve three sample configuration files, outboard.cfg, speedboat.cfg, and puller.cfg. See *Hints for Testing* to use these files.

Judged Responses

An ill-formed command produces a PRESENTATION ERROR. E command steering positions outside the closed interval $[-d, d]$ result in PRESENTATION ERROR. Failing to pass all waypoints in order before issuing a Q command, running out of trial time, or running out of judged time before passing all waypoints all result in WRONG ANSWER. Spurious dialog after issuing a Q command produces PRESENTATION ERROR. Finally, if the server has to wait longer than one second (real time) for your client to issue a command, the judged response is TIME LIMIT EXCEEDED. Note that this TIME LIMIT EXCEEDED is the exact same message issued if your program accumulates more than the allotted CPU seconds.

It is very important that your program flush standard output after issuing a command to the server. e.g.

C	`fputs("C\n",stdout); fflush(stdout);`
C++	`cout << "C\n" << flush;`
Java	`System.out.println("C"); System.out.flush();`

Problem 3
Bumper Boat Autopilot (still continued)

Sample Transcript

The following is a sample transcript of a client and server conversation. The client commands appear offset from server responses for clarity. The transcript file (see *Hints for Testing*) uses four tab characters to offset client commands from server responses. The *italicized commentary* to the right of the transcript does not occur in a client-server conversation. Additionally, it would be impossible to produce meaningful trial sessions (T commands) that would fit on a single-page sample transcript; the sample session demonstrates the nature of the conversation.

```
                                C               client request for the steering limits
45                                                  ... +/- 45 degrees
                                T               start a trial run
1000.0                                              ... 1000 simulated seconds for training trials
                                E,0             Engage throttle with centered steering
0.000000 0.001250 -0.000000
                                I               coast
0.000000 0.002500 -0.000000
                                E,25.5          throttle with steering 25.5 degrees right of center
0.000000 0.003628 -0.061666
                                T
999.7
                                E,-45
0.000000 0.000884 0.101286
                                J               start a judged run
100.0                                               ... only 100 simulated seconds for judging
2                                               two waypoints follow
0,0.001
0,0.503
                                E,0
0.000000 0.001250 -0.000000 1
                                E,0
0.000000 0.003749 -0.000000 2                   last waypoint encountered
                                Q               signal quit to the server
```

Hints for Testing

You may act as the server to a client program to test bumper boat behavior that configuration files might not produce. Execute your client program with you, the programmer, acting as the server. Let standard input come from your command line environment. Supply correct responses to your program.

The server process used by the judges is available to the contestants. You are encouraged to use the sample configuration files to develop your program. You may use the server in two ways. In the absence of a working client program, you, the programmer, can act as the client, typing input to the server. Use the following command:

`test3` *configurationFile transcriptFile*

To test your program and its conversation with the server, use the command:

`test3` *configurationFile transcriptFile sourceFile*

Problem 3
Bumper Boat Autopilot (still continued)

where

- *configurationFile* is described below.
- *transcriptFile* is a file that captures the server's inputs and outputs for later analysis. The server's inputs are your client's commands. Client commands are offset by four tab characters to distinguish the client vs. server dialog.
- *sourceFile* is the file name of the source code to your client. When specified, the test3 script calls the compile script.

Simulation Details and Specifying Your Own Boat and Waypoints

The simulation model is a round boat with a uniform distribution of mass. The propulsion and steering mechanism are represented as a single force acting on a fixed point within the circle of the boat. The steering mechanism does not change the position or the quantity of force, only its direction with respect to the center of the boat.

There are two components of drag. Dynamic drag acts to slow the boat and is proportional to velocity (translational and rotational). Static drag occurs only when the motor is idle, and acts to stop the boat when the velocity falls below a threshold value.

The boat travels in a straight line between two consecutive 0.1 second time points.

You create a boat configuration with a configuration file of the form:

t_T	cumulative time allotment, in seconds, for trial runs
t_J	time allotment, in seconds, for the judged runs
m	mass, in kilograms, of the boat
r	radius, in meters, of the boat
x	x coordinate of the propulsion force, with respect to the center of the boat at 0,0
y	y coordinate of the propulsion force, with respect to the center of the boat at 0,0
F	the propulsion force, in Newtons
d	the range of steering deflection, $[-d, d]$ in degrees
a_0	the skew coefficient of steering inputs
a_1	the scale coefficient of steering inputs
w_{x1}, w_{y1}	waypoint definitions
w_{x2}, w_{y2}	
...	
w_{xW}, w_{yW}	

Problem 3
Bumper Boat Autopilot (still continued)

For example, a two-meter-diameter boat with a rear-mounted outboard motor steered by a rudder might have a configuration of

```
1000    1000 simulation seconds total for training
100     100 simulation seconds for the judged run
200     200 kg
1.0     2 m diameter = 1 m radius
0.0     propeller at
-1.0                back of boat
25      25 N propeller force
45      the rudder can be swung from -45 to 45 degrees
0.0     the force is in the direction
1.0                                  the rudder handle points
0.0,0.001
0.0,0.503
```

This might be considered an "inverting control" because moving the rudder to the left steers the boat to the right.

You should choose masses and forces of about equal values to avoid under-powered or unstable configurations.

2009/2010 SOUTHERN CALIFORNIA REGIONAL
ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST

Problem 4
Character Checker

The judges for the Programming Contest always carefully check their sample and judged input. However this can be tedious for large files so they would like your team to write a program that will report on the characters in an input file.

The characters are to be split into the following classes based on their eight-bit values:

non-space control:
  0x00 through 0x1F that are not also whitespace
  0x7F delete

whitespace:
  0x20 space
  0x09 tab
  0x0A newline
  0x0B vertical tab
  0x0C formfeed
  0x0D carriage return

non-space printable:
  0x21 through 0x7E

non-ascii:
  0x80 through 0xFF

Input to your program will be a file of eight-bit bytes.

Your program is to print the number of characters in the file by printing a line containing the string "total:", a single space, followed by the count with no leading zeroes. If the count is non-zero print a detail report on the characters seen in the file.

In the detail report represent an eight-bit character by a two-place hexadecimal value, using capital letters for the values A_{16} through F_{16} with a leading zero if necessary. For example, the character "n" would be 6E.

Report the last character seen before the end of the file by printing "last:", a single space, and the hex representation of the character.

Next, report the count of characters in each class, in the order: non-space control, whitespace, non-space printable, and non-ascii. Print the class name, a colon, a single space, and the count without leading zeroes.

If the whitespace count is non-zero report the whitespace characters seen immediately after the whitespace report line. For each whitespace character that appears at least once in the file print two spaces, the hex representation of the character, a single space, and the count without leading zeroes. Print one line per character, in character numerical order.

No trailing whitespace is to appear on an output line.

Problem 4
Character Checker (continued)

Sample Input

```
this is a sample
```

Output for the Sample Input

```
total: 17
last: 0A
non-space control: 0
whitespace: 4
  0A 1
  20 3
non-space printable: 13
non-ascii: 0
```

2009/2010 SOUTHERN CALIFORNIA REGIONAL
ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST

Problem 5
Parlay Wagering

Parlay wagering offers sports bettors the opportunity to win a large sum of money from a small initial wager. A parlay wager is a combination of individual independent wagers that only pays if no individual wager loses. The total returned from each wager is applied or "parlayed" to the next wager in turn. If any individual wager loses, the bettor receives nothing. If any individual wager is a tie or "push", the amount wagered is returned—meaning that individual wager is effectively ignored. The ultimate payout is therefore reduced.

The sports book quotes the payout rate for an individual wager as a "money line"—a non-zero integer in the range −2000 to 2000. To compute the payout for a successful wager, the money line is converted to a decimal multiplier as follows: if the money line is positive, it is divided by 100 to obtain the multiplier. If the money line is negative, the absolute value is divided into 100 to obtain the multiplier. The multiplier is always truncated to three digits after the decimal point. The wager is multiplied by this multiplier to determine the amount won. The amount won is truncated to the cent (the sports book keeps the fractional cents). The winnings are added to the wager amount and that total is parlayed as the next wager.

Consider the following example for a five-way parlay wager:

Money Line	**Wager**	**Result**	**Multiplier**	**Amount Won**
−170	$10.00	Win	100/170	$5.88
−160	$15.88	Win	100/160	$9.92
125	$25.80	Win	125/100	$32.25
−135	$58.05	Win	100/135	$42.95
−140	$101.00	Win	100/140	$72.11
			Subtotal	**$163.11**
			Original Wager	**$10.00**
			Total Returned	**$173.11**

The maximum total returned for any parlay wager is $1 million. If the calculated total exceeds that amount, the actual total returned will be $1 million.

Your team is to write a program that will calculate the total amount returned for a series of parlay wagers. The first line of input to your program will contain the total number of parlay wagers as a single integer starting in the first column. Each wager that follows will be represented by a series of lines. The first line of each parlay wager contains the initial bet and the count of individual wagers as integers separated from each other by a single space. The following lines represent the individual wagers, one per line. Each individual wager is given as its money line followed by a single space and the result of the wager ("Win", "Tie", or "Loss").

For each parlay wager, your program is to print the total amount returned in dollars and cents on a single line starting in the first column without embedded or trailing whitespace. Print the leading dollar sign and insert commas at the millions and thousands positions as needed.

Problem 5
Parlay Wagering (continued)

Sample Input

```
3
10 8
-170 Win
-160 Win
125 Win
-135 Win
-140 Win
110 Win
120 Win
130 Win
15 5
100 Win
-100 Tie
-250 Win
135 Tie
265 Tie
10 2
100 Loss
300 Tie
```

Output for the Sample Input

```
$1,839.44
$42.00
$0.00
```

2009/2010 SOUTHERN CALIFORNIA REGIONAL
ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST

Problem 6
Endless Plains Capital

The capital city of the (nearly) Endless Plains is laid out as a dense rectangular grid with streets that run along each block, north/south and east/west. Each street runs the entire length or width of the city—there are no dead ends. Outside of the city, however, the Endless Plains are flat and unobstructed; one can travel in any direction he or she pleases.

Navigating and determining distances within the capital city is quite easy. The distance from one block to the next is considered one unit. One can only travel on the streets, so the distance between two intersections is simply the east/west difference added to the north/south difference.

However, sometimes residents need to travel outside of the city to other points on the Endless Plains. While the grid is logically extended outside of the capital city, there are no streets limiting the direction of travel. Residents of the Endless Plains are not sure what the shortest distance is between any two points. Your team is to help by writing a program that, given any two integer-coordinate points on the Endless Plains, will calculate the shortest distance between them.

Input to your program will be a series of capital city scenarios. Each scenario will be represented as a series of lines. The first line contains two (x, y) pairs, representing the southwest and northeast corners of the capital city rectangle. These integer coordinates will begin in the first column and be separated from each other by a single space. The remaining lines in the scenario will be pairs of points, again expressed as two (x, y) pairs. The integer point coordinates will again begin in the first column and be separated from each other by single spaces. The list of point pairs will continue until an empty line appears to end the scenario. The final scenario will be followed by the end-of-file.

For each point pair, your program is to print a line containing the shortest Endless Plains distance between them. The value should be printed to three decimal places starting in the first column. No trailing whitespace is to appear on a line. Print an empty line at the end of each scenario (including the last).

Coordinates will always be integers in the range -1000 to 1000. The capital city corners will always be distinct, and the x and y coordinates of the northeast corner will always be greater than the corresponding coordinates of the southwest corner. All point pairs will also be distinct.

Sample Input

```
4 2 10 11
12 9 8 9
6 14 9 17
8 9 14 1
5 4 8 9

4 3 13 13
10 1 6 6
6 6 2 11
```

Problem 6
Endless Plains Capital (continued)

Output for the Sample Input

```
4.000
4.243
10.944
8.000

7.472
7.385
```

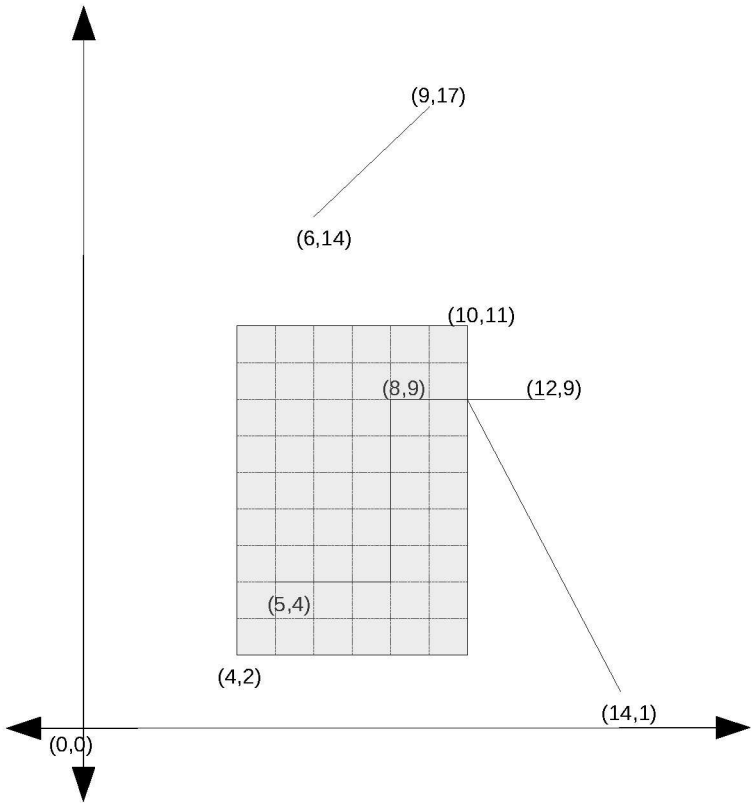

Figure 1. Diagram of the first Sample Input scenario.

2009/2010 SOUTHERN CALIFORNIA REGIONAL
ACM INTERNATIONAL COLLEGIATE PROGRAMMING CONTEST

Problem 7
Disaster Recovery Plans

ForwardProgess.com is a huge future studies think tank. An exhaustive internal study indicates that the company is weak in disaster preparedness. A deep backlog of externally funded studies prevents the firm from allocating internal resources to prepare a Disaster Recovery Plan (DRP). As a result, ForwardProgess.com has contracted your team to publish a DRP and give a single-sided hard-copy to each employee in the corporation. Single-sided print publishing is expensive, but necessary because the computers might be down or the pages might get wet and bleed through during a disaster. Your plan must minimize the number of printed pages.

Within the company, each supervisor has been tasked to prepare a list of possible disasters, such as fire, earthquake, flood, or Jolt Cola shortage, and the appropriate responses that will ensure business continuity. For each event, the supervisor determines unique actions that individual employees should perform, and the order in which these actions must be performed. The actions must be ordered to ensure a proper reaction to the given event. Some actions require tests for completion and repeated actions. The person executing the steps in the plan is expected to determine when a cycle in the plan is complete.

There is one quirky rule that your team must implement in the DRP: true to its company name, the culture of ForwardProgess.com mandates that one may only turn pages forward in the plan, not backward.

Your program must read the set of actions for a given event and determine the minimum number of pages that produces a properly ordered set of actions, subject to the condition that one may never leaf backwards through the plan.

Input to your program will be a series of DRPs. Each DRP will be represented as a series of input lines ending with an empty line. The first line of the DRP is the name of the disastrous event: up to 60 characters starting in the first column. The second line contains T, the maximum number of tasks per page, and N, the total number of unique tasks. These values will appear starting in the first column and be separated from each other by a single space. The remaining input lines contain DRP task dependency pairs. The DRP tasks are represented by integer values 1 through N. Each dependency pair represents a predecessor-successor relationship: the successor task cannot begin until the predecessor task has completed. The dependency pairs are given as integer pairs separated by a comma. For example, the pair "2,5" means that task 5 cannot begin until task 2 has completed.

For each DRP in the input, your program is to print a single line containing the name of the disaster, followed by a colon, a single space, and the integer count of the minimum number of pages needed to print the plan. If the plan cannot be printed using the specified page size, print the string "not printable" in place of the minimum page count. No leading or trailing whitespace is to appear on an output line.

There will be no more than ten tasks per page and no more than forty tasks in any plan.

Problem 7
Disaster Recovery Plans (continued)

Sample Input

```
Jolt Cola Shortage
4 8
1,2
2,5
5,1
2,6
5,6
2,3
6,7
7,6
3,4
4,3
5,7
4,8
8,4
8,7

Pizza Strike
2 4
1,2
2,3
3,1
```

Output for the Sample Input

```
Jolt Cola Shortage: 3
Pizza Strike: not printable
```