Seventeenth Annual
University of Central Florida

ACM · UPE

High School Programming Tournament

Problems

Problem Name	Filename
Two X-Men	X-MEN
Bowling For Soup	BOWLING
Blooscreins of Death	SPACESHIP
Blast-a-mon Battle Plan	BLASTAMON
The Array Reloaded	RELOADED
Bits of Candy	CANDY
Anyone For a Game of Darts?	DARTS
Crosstown Traffic	TRAFFIC
The Shawshank Transaction	OVERDRAFT
Springfield Skippy and the Cymbal of Doom	CYMBAL

Call your program file: *filename*.c, *filename*.cpp, or *filename*.java
Call your input file: *filename*.in

For example, if you are solving Bits of Candy:
Call your program file: candy.c, candy.cpp or candy.java
Call your input file: candy.in

Two X-Men

Filename: `X-MEN`

Cyclops and Wolverine are mutants, evolved humans with extraordinary abilities. They fight for peace between humans and mutants as part of the team known as the X-Men. However, Cyclops and Wolverine have always had a rivalry, each trying to best the other in all things, including combat. Professor Charles Xavier, leader of the X-Men, has taken an interest in this rivalry, knowing that friendly competition often leads to the betterment of both involved. He's asked you to write a program that can determine the outcome of a combat involving Wolverine, Cyclops, and a number of enemies. Having this information might help him instruct the two mutants on how to improve their combat techniques.

The Problem:

For our purposes, all combats happen on a grid of squares. Cyclops and Wolverine have different combat styles that have naturally emerged as a result of their mutant abilities. Wolverine has indestructible retractable metal claws that spring from his hands. As a result, he excels at close-range combat, but cannot attack enemies at a distance. In a single attack, Wolverine can eliminate all enemies standing next to him (including those in diagonally adjacent squares). Cyclops on the other hand has a constant stream of pure energy that erupts from his eyes. He has learned to harness this energy as a weapon with the help of a ruby quartz visor that he wears over his eyes. Using the visor, he can take out any number of targets in a straight line in any one direction with a single blast.

Your program should analyze the given combat situation, determine the maximum number of enemies each mutant can take out in a single attack, and print the score for that one attack. Both mutants will move into a position that best suits their abilities before striking. They can move into and through spaces occupied by enemies and each other if necessary, but they cannot attack from any space that is occupied. Wolverine is faster than Cyclops, so he will move first. Wolverine is also quick enough to dodge Cyclops' blast, so Cyclops is free to fire in Wolverine's direction without worrying about hitting him or having his attack blocked by the large mutant. Enemies will never move. Both mutants attack simultaneously, so if an enemy would be eliminated by both mutants, give both X-Men the credit.

The Input:

There will be multiple combats to analyze. Input will begin with a single positive integer, *n*, on the first line, indicating how many combats will be analyzed. The first line of each combat data set will contain two integers *x* and *y*, neither less than 2 nor greater than 50. *x* and *y* represent the width and height of the combat area, respectively. The next *y* lines will contain *x* contiguous characters (no leading or trailing spaces). Each character will be either '`X`', '`C`', '`W`', or '`.`'. An '`X`' indicates the position of an enemy. A '`W`' indicates Wolverine's position. A '`C`' indicates Cyclops' position. A '`.`' simply indicates an empty space. Naturally, there will be exactly one '`W`' and one '`C`' in every data set.

The Output:

For each combat, print the number of enemies that Wolverine and Cyclops each eliminate. List the mutant that scores the most kills first (in the case of a tie, print Wolverine first). Leave one blank line after the output for each combat. Follow the format of the Sample Output.

Sample Input:

```
2
5 5
CX.X.
.X.X.
...X.
...X.
...W.
10 10
X.........
.X....W...
..X.......
...X......
XXX.X.....
X.X..X....
XXX...X...
..C....X..
........X.
..........
```

Sample Output:

```
Combat 1:
  Wolverine: 5
  Cyclops:   4

Combat 2:
  Cyclops:   9
  Wolverine: 8
```

Bowling For Soup

Filename: `BOWLING`

It's "Bowling for Soup" night again at the lanes where you work. Problem is, the scoring computers are on the fritz again. All they'll do is keep track of the number of pins knocked down – they won't tally up a score for each player. So, at the risk of much grumbling and derision, you've asked each lane to keep their own paper scorecard. Each bowler that breaks a score of 250 simply has to present his scorecard to you to receive a free bowl of tomato soup (okay, it's not a real good prize, but it does bring them in). Midway through the first game, it occurs to you that certain unscrupulous bowlers might pad their scores a little bit. You decide that you had better write a program to verify the scorecards.

The Problem:

Of course, in order to write a score-keeping program, you'll have to know the scoring rules for bowling. So, you pry the pro shop guy away from the action and make him explain everything to you. After much scribbling, backtracking, and head-scratching, this is what you get out of him:

1. A game of bowling is broken up into ten frames.
2. Each bowler gets two throws to try to knock down all ten pins in his frame. There are three ways to end a frame, each with a different name:
    a. If all ten pins are knocked down on the first throw of the frame, it is termed a *strike*.
    b. If all ten pins are knocked down, but it takes both throws to do so, it is called a *spare*.
    c. If any pins remain standing after the second throw, it is called an *open frame*.
3. After each frame, the pins are reset – that is, they are replaced so that there are always ten pins at the start of the frame.
4. Bowlers bowl in a specified order; each one bowls exactly one frame on his turn.
5. Scoring is based on the number of pins knocked down, with bonuses for spares and strikes. Specifically:
    a. The score for an open frame is the number of pins that are knocked down.
    b. The score for a spare is ten (the number of pins knocked down) *plus* the number of pins knocked down on the *next throw* (not frame).
    c. The score for a strike is ten (the number of pins knocked down) *plus* the number of pins knocked down on the next *two throws* (note that those two throws could span multiple frames).
6. If the tenth frame ends in a spare or a strike, the bowler immediately bowls an "extra frame". This frame does not count toward the final score, except in calculating the score for the tenth frame. Also, this frame lasts only as long as it needs to (one throw for a spare, two for a strike). No extra "extra frames" are awarded if the bowler scores a spare or strike in his given extra frame because no score is calculated for that spare/strike. Note that this means that the tenth frame and all extras never lasts more than three throws.

The Input:

The input file will contain several lanes' worth of data. Each lane will begin with a line containing a single positive integer *n* ($n < 7$), the number of bowlers. The next *n* lines will each contain one bowler's name, which will be no longer than thirty characters and will not contain spaces. These lines will specify the order that bowlers bowl in. Following the bowler's names will be a line containing the number of pins knocked down on each throw of the game. They are given in the order that they are bowled, so that Player 1's first frame will be given, followed by Player 2's first frame, etc. No more than ten pins will ever be knocked down in a single frame. The line of scores will not exceed 500 characters.

Input ends when *n* (the number of bowlers on a lane) is 0.

The Output:

For each lane, first output a line that says "`Lane #x:`" with the appropriate *x*. Then, print the bowler's names and scores, each on a line by itself, by *descending score*. See the Sample Output for formatting. Leave a blank line between the lanes.

Sample Input:

```
1
Chuck
10 10 10 10 10 10 10 10 10 10 10 10
2
GutterMan
Bill
0 0 6 3 0 0 8 2 0 0 7 3 0 0 8 1 0 0 9 1 0 0 9 0 0 0 8 2 0 1 10 0
0 9 1 0 2 8 2 7
0
```

Note that the scores in the actual input file will be on only one line

Sample Output:

```
Lane #1:
 Chuck 300

Lane #2:
 Bill 156
 GutterMan 3
```

Blooscreins of Death

Filename: SPACESHIP

The year is 2114. The evil Phil Bates, leader of the Blooscrein Empire, is trying to destroy the human race to make room for his own species. The Earth division of the United Network of Intergalactic Transmissions (UNITs) selected you to deliver a distress signal to neighboring star systems. You *must* get this message through to avoid the extermination of mankind and the further domination of the Blooscreins. They have provided you with an old computer system, much like the one in front of you, and a cloaking spaceship. However, the Blooscrein warships have superior technology. If you fly too close to them, their more advanced radar systems will be able to find you. You have managed to escape Earth, but will you save mankind?

The Problem:

You are currently in the middle of a galaxy called MicroStars, surrounded by Blooscrein Empire's star fleet. Fortunately, your spaceship knows the range of each enemy warship's radar. So you decide to write a program that will tell you how many Blooscrein warships can "see" you. Given your location, and the location and effective range of each enemy's radar, determine how many warships will pick you up. The Blooscrein radar systems will detect you even if you are at the very edge of the radar's range. Assume the radar signals emit from the location of the ship and that the sizes of all ships are negligible.

The Input:

The first line of the file will consist of the number of data sets, *n*. The first line of each data set contains four integers, *x*, *y*, *z*, and *e*; (*x*, *y*, *z*) is your position in 3D space and *e* is the number of Blooscreins in your vicinity. The following *e* lines each contain four integers, x_i, y_i, z_i, d_i. These represent the position and effective radius of each warship's radar.

The Output:

For each data set, print the line "`You will be picked up by x radars.`" where *x* is the number of enemy ships that will detect your spaceship.

Sample Input:

```
2
1 2 3 2
4 5 6 3
0 1 2 2
5 10 2 1
1 5 10 7
```

Sample Output:

```
You will be picked up by 1 radars.
You will be picked up by 0 radars.
```

Blast-a-mon Battle Plan

Filename: BLASTAMON

Your friend is nuts about the latest trading card game known as Blast-a-mon. Like many other card games, Blast-a-mon involves cute little monsters that engage in battles with one another. They use their extraordinary fighting powers to cause massive amounts of damage to their opponents and gravely endanger the population of whatever city they happen to be fighting in.

When your friend isn't boring you with the list of the latest 127 cards that have come out this week, he's bugging you to play a game or two with him. You don't really care about this whole Blast-a-mon thing, but you figure you can be nice to your friend until the fad passes. You decide to write a program to help you plan your Blast-a-mon strategy, so you don't lose horribly every time you play.

The Problem:

Given a list of Blast-a-mon characters and their hit points (amount of damage they can take before falling unconscious), print out the list in order of lowest to highest hit points.

The Input:

There will be multiple Blast-a-mon card decks to analyze. Each deck will begin with a single positive integer, *n,* on the first line, indicating how many Blast-a-mon cards are in the deck. Following this will be *n* sets of two lines, each pair of lines representing a Blast-a-mon card. The first line contains the name of the Blast-a-mon card. The name will be no longer than 20 characters and will consist of letters only, with no leading or trailing spaces. The second line contains a positive integer between 1 and 1000, inclusive, indicating the number of hit points that the Blast-a-mon character has. In any deck, each Blast-a-mon character will have a different number of hit points. Input will be terminated by a value of zero for the number of cards in the deck (this deck should not be processed).

The Output:

For each deck, print all of the cards' names in ascending order according to hit points. Leave one blank line after the output for each deck.

Sample Input:

```
3
Roar
10
Squeak
1
Bark
5
2
Megamon
1000
Minimon
1
0
```

Sample Output:

```
Squeak
Bark
Roar

Minimon
Megamon
```

The Array Reloaded

Filename: RELOADED

The Problem:

The Unity and his hacker friends have managed to crash the neural-interactive simulation known as the Array. If the Array isn't restored soon, the enslaved humans that power the machines will begin to awaken in the real world. Entire crops could be lost. The Array has had a rather substantial upgrade since we last saw it. Internally, the Array is stored as a two-dimensional rectangular grid of alphanumeric values (a-z, A-Z, 0-9) (since the computers of 2203 are very advanced, they use base-62 rather than binary). Backups of the Array are made, but it is too large for a single unified backup. Rather, backups are stored as smaller rectangular grids. Not all backups are made at the same time, so out of date information from previous backups should be replaced by fresh data. It is entirely possible that some areas of the array have *never* been backed up, and these areas should default to a value of '0'.

The Input:

The input may contain multiple data sets. Each data set begins with three decimal integers *n*, *m*, and *k*. *n* represents the number of rows in the Array. *m* represents the number of columns. Neither *m* nor *n* will exceed a value of 62. *k* is the number of backups in the set. The input is terminated by a data set in which *n*, *m*, and *k* are equal to zero, which should not be processed. Each backup begins with five integers, *x*, *y*, *i*, *j*, and *t*. *x* and *y* represent the row position and column position respectively of the upper-left value in the backup. Recall that positions in the Array begin at zero, numbered from the upper-left corner. *x* is guaranteed to be strictly less than *n* and *y* will be strictly less than *m*. *i* and *j* represent the number of rows and columns respectively in the backup. When a backup is loaded it will not exceed the bounds of the Array. *t* is the time at which the backup was made, with larger values being more recent. The backups are *not* guaranteed to be in chronological order. The next *i* lines each contain *j* characters, separated by spaces representing the data stored in the backup.

The Output:

The output for each set should begin with a line indicating the number of the current set. The remainder of the output for each set should present the state of the Array once it has been reloaded followed by a blank line. Each row of the Array should be on its own line with each value separated by a space. Follow the format of the Sample Output.

Sample Input:

```
4 4 3
0 0 4 2 16
a b
e f
i j
m n
0 2 2 2 32
c d
g h
2 2 2 2 47
k l
o p
3 3 2
1 1 2 2 16
Z Z
Z Z
0 0 3 3 15
A A A
A A A
A A A
0 0 0
```

Sample Output:

```
Set 1:
a b c d
e f g h
i j k l
m n o p

Set 2:
A A A
A Z Z
A Z Z
```

Bits of Candy*

Filename: CANDY

Gary is obsessed with candy. Just eating it isn't enough, he wants to know all there is to know about candy. One of his favorite activities is to develop relationships of how various candy products relate to each other. He's been doing this by hand (or rather, by taste) so far, but he has realized that using a program might help him complete his task faster, allowing him to move on to another delicious task.

The Problem:

Each candy will be encoded with four bits. The bits represent (in order from most significant to least significant) chocolate, caramel, nougat/peanut butter, peanuts. Gary has compiled the following table of candies with their respective codes.

-	0000
Planters	0001
Reeses_Pieces	0010
-	0011
Sugar_Babies	0100
Pay_Day	0101
-	0110
-	0111
Hersheys	1000
Goobers	1001
Reeses_Cups	1010
Nutrageous	1011
Caramello	1100
Baby_Ruth	1101
Milky_Way	1110
Snickers	1111

Now that the candies are encoded, Gary wants to try bit-wise operations on them to see how they relate to each other. The operations Gary wants you to try are ~ (not), & (and), and | (or). The preceding order indicates operator precedence from highest to lowest. The & and | operators are binary, while the ~ operator is unary. Some sample equations are:

~Reeses_Cups = ~1010 = 0101 = Pay_Day
Goobers | Pay_Day = 1001 | 0101 = 1101 = Baby_Ruth
Caramello & Pay_Day = 1100 & 0101= 0100 = Sugar_Babies

* Inspired by a category of "The Programmers Challenge" event at Game Developer's Conference 2003. Content belongs to the organizers of the event and the International Game Developer's Association.

The Input:

Input will begin with a single positive integer, *n,* on the first line, indicating the number of candy bar operations to process. Each of the next *n* lines will contain one candy bar expression consisting of candy bar names and operators. Operators will consist only of those listed above (no parentheses will be used). Candy bar names will consist of only upper- and lower-case letters and underscores. Only the candies listed in the table above will be used. You may assume all expressions will be valid.

The Output:

Evaluate the candy expressions and print the results one per line. If the given expression evaluates to an unknown configuration (listed as '-' in the table), print "`Unknown candy bar!`" for the result. Follow the format of the Sample Output.

Sample Input:

```
5
~Reeses_Cups
Goobers | Pay_Day
Caramello & Pay_Day
~Sugar_Babies & Nutrageous
Milky_Way & ~Hersheys
```

Sample Output:

```
Pay_Day
Baby_Ruth
Sugar_Babies
Nutrageous
Unknown candy bar!
```

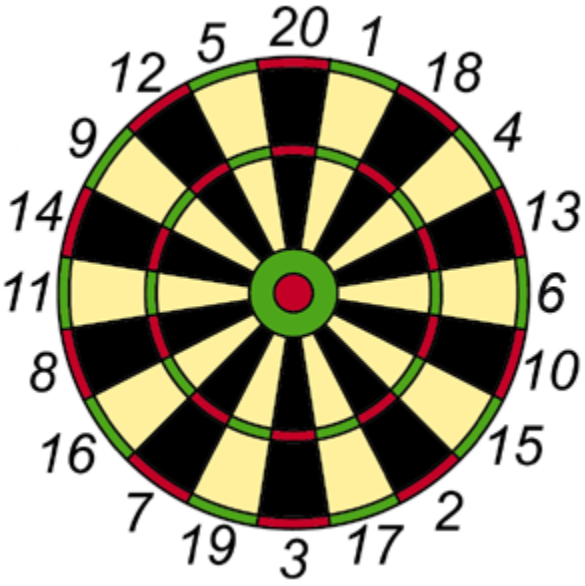
Anyone for a Game of Darts?

Filename: DARTS

David and Timothy went to the Student Union to play darts one day (they always wanted to try it). Unfortunately, they did not know how to score even an individual dart and quickly got into an argument accusing each other of cheating. It was quite a public embarrassment for them so they would like you to write a program to help them with scoring.

The Problem:

Given a standard dartboard centered on the origin of standard coordinate axes and a number of (x, y) positions where darts hit, report the score of each dart. A dartboard looks as follows:



The very center area, known as the Inner Center Ring or Bulls-eye, is worth 50 points and has a radius of 7 mm. There is also an Outside Center Ring, and it is worth 25 points and has an outer radius from the dartboard center of 16 mm. Darts hitting any other area besides these two rings are awarded the point value of the pie wedge it hits with two exceptions. The Inner Narrow Band, called the Triple Ring, is worth three times the value of the pie wedge and has an outer radius of 107 mm. The Outer Narrow Band, called the Double Ring, is worth double the value of the pie wedge and has an outer radius of 170 mm. Both the Triple Ring and the Double Ring are 8 mm wide. All wedges are equal size and the vertical axis (y-axis) goes through the center of the top-middle wedge (the 20). A dart hitting outside the board scores zero points.

The Input:

The first line will contain a positive integer *n* indicating the number of darts to score. For each dart, there will be a floating-point (x, y) location in millimeters on a line by itself. Assume that no dart will ever fall exactly on any segment separating regions.

The Output:

For each dart, report the score that that dart receives. Each report should be on a line by itself.

Sample Input:

```
2
0.0 0.0
0.0 76.5
```

Sample Output:

```
50
20
```

Crosstown Traffic

Filename: TRAFFIC

It is 2064 and as usual, there is a crash on the Interstate. All lanes are blocked except for the rightmost. Mike needs to get to work so he isn't late and wants to know when he might be able to get past the crash. He needs your help to determine it!

The Problem:

Given the number of lanes and number of rows of cars on the highway, determine when a specified car will exit past the accident. When a driver gets around the accident, there is a free space that either the car behind can fill or the car to the immediate left can fill. A number of rules specify how the flow works:

1. Only the rightmost lane is not blocked by the accident.
2. Cars in 2064 have four-wheel steering and can move to the right without also moving forward.
3. Any car that gets an empty space in front of it will allow the car from the left enter instead of moving forward, but will do this only once (each driver only has a small amount of patience).
4. Cars that have already let in another car will always move into an empty space in front of them.
5. Cars will always move to the right if possible (whether being let in or moving into an unclaimed spot). If they cannot move to the right, they will move forward if possible.
6. Cars do not move off the road.

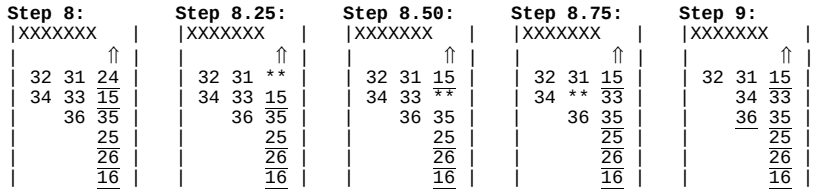
Let's consider an example. First, cars are numbered by combining the lane number as a ten's digit and the position within the lane as the one's digit. Underlined numbers will denote cars that have already let somebody in front of them and will not do it again. Here is the example showing how all the cars would make it past the accident:

```
Step 1:    Step 2:    Step 3:    Step 4:    Step 5:    Step 6:    Step 7:
|XXXXXXX  |  |XXXXXXX  |  |XXXXXXX  |  |XXXXXXX  |  |XXXXXXX  |  |XXXXXXX  |  |XXXXXXX  |
|       ⇑ |  |       ⇑ |  |       ⇑ |  |       ⇑ |  |       ⇑ |  |       ⇑ |  |       ⇑ |
| 31 21 11 |  | 32 31 21 |  | 32 31 12 |  | 32 31 22 |  | 32 31 13 |  | 32 31 23 |  | 32 31 14 |
| 32 22 12 |  | 33 22 12 |  | 34 33 22 |  | 34 33 13 |  | 34 33 23 |  | 34 33 14 |  | 34 33 24 |
| 33 23 13 |  | 34 23 13 |  | 35 23 13 |  | 36 35 23 |  | 36 35 14 |  | 36 35 24 |  | 36 35 15 |
| 34 24 14 |  | 35 24 14 |  | 36 24 14 |  |    24 14 |  |    25 24 |  |    25 15 |  |       25 |
| 35 25 15 |  | 36 25 15 |  |    25 15 |  |    25 15 |  |    26 15 |  |       26 |  |       26 |
| 36 26 16 |  |    26 16 |  |    26 16 |  |    26 16 |  |       16 |  |       16 |  |       16 |

Step 8:    Step 9:    Step 10:   Step 11:   Step 12:   Step 13:   Step 14:
|XXXXXXX  |  |XXXXXXX  |  |XXXXXXX  |  |XXXXXXX  |  |XXXXXXX  |  |XXXXXXX  |  |XXXXXXX  |
|       ⇑ |  |       ⇑ |  |       ⇑ |  |       ⇑ |  |       ⇑ |  |       ⇑ |  |       ⇑ |
| 32 31 24 |  | 32 31 15 |  |    32 31 |  |    32 33 |  |    32 35 |  |    32 25 |  |    32 26 |
| 34 33 15 |  |    34 33 |  |    34 33 |  |    34 35 |  |    34 25 |  |    34 26 |  |    34 16 |
|    36 35 |  |    36 35 |  |    36 35 |  |    36 25 |  |    36 26 |  |    36 16 |  |       36 |
|       25 |  |       25 |  |       25 |  |       26 |  |       16 |  |          |  |          |
|       26 |  |       26 |  |       26 |  |       16 |  |          |  |          |  |          |
|       16 |  |       16 |  |       16 |  |          |  |          |  |          |  |          |

Step 15:   Step 16:   Step 17:   Step 18:
|XXXXXXX  |  |XXXXXXX  |  |XXXXXXX  |  |XXXXXXX  |
|       ⇑ |  |       ⇑ |  |       ⇑ |  |       ⇑ |
|    32 16 |  |    32 36 |  |    32 34 |  |       32 |
|    34 36 |  |       34 |  |          |  |          |
|          |  |          |  |          |  |          |
|          |  |          |  |          |  |          |
|          |  |          |  |          |  |          |
|          |  |          |  |          |  |          |
```

To clarify, here's how we transitioned from step 8 to step 9 (asterisks represent empty spaces):



The Input:

The first line of the input will consist of a single integer representing the number of data sets. Each data set will be on a single line consisting of four integers, *a*, *b*, *c* and *d* where *a* is the number of lanes (no more than 5), *b* is the number of rows of cars (rows are always full of cars), *c* is the lane of the car Mike is in (with 1 being the rightmost lane), and *d* is the order of the car in that lane from front to back (beginning with 1 at the front). There will be a maximum of 9 rows of cars.

The Output:

For each data set, output the header "`Traffic Jam i:`" with *i* beginning at one. Then, output each car movement that occurs in clearing the traffic jam through to the point where Mike finally can make it past the accident. For each step, output as appropriate:

1. "`Car c clears the accident.`" if car *c* clears the accident.
2. "`Car c moves forward.`" if car *c* moves forward (but not past accident).
3. "`Car c lets car d in.`" if indeed car *c* does let car *d* in.
4. "`Car c moves to the right.`" if car *c* moves to the right without being let in.

Again, cars are numbered as shown in the example above (the tens digit represents the lane and the ones digit represents the order of that car within that lane). After the steps for each data set, output the phrase "`Mike's car will exit after j cars go in front of it.`" on a separate line, where *j* is the number of cars leaving (clearing the accident) in front of Mike. Leave a blank line after the output of each data set.

Sample Input:

```
2
3 6 1 2
3 6 3 3
```

Sample Output:

```
Traffic Jam 1:
Car 11 clears the accident.
Car 12 lets car 21 in.
Car 22 lets car 31 in.
Car 32 moves forward.
Car 33 moves forward.
Car 34 moves forward.
Car 35 moves forward.
Car 36 moves forward.
Car 21 clears the accident.
Car 12 moves forward.
Car 13 lets car 22 in.
Car 23 lets car 33 in.
Car 34 moves forward.
Car 35 moves forward.
Car 36 moves forward.
Car 12 clears the accident.
Mike’s car will exit after 2 cars go in front of it.

Traffic Jam 2:
Car 11 clears the accident.
Car 12 lets car 21 in.
Car 22 lets car 31 in.
Car 32 moves forward.
Car 33 moves forward.
Car 34 moves forward.
Car 35 moves forward.
Car 36 moves forward.
Car 21 clears the accident.
Car 12 moves forward.
Car 13 lets car 22 in.
Car 23 lets car 33 in.
Car 34 moves forward.
Car 35 moves forward.
Car 36 moves forward.
Car 12 clears the accident.
Car 22 moves forward.
Car 13 moves forward.
Car 14 lets car 23 in.
Car 24 lets car 35 in.
Car 36 moves forward.
Car 22 clears the accident.
Car 13 moves forward.
Car 23 moves forward.
Car 14 moves forward.
Car 15 lets car 24 in.
Car 25 moves forward.
Car 26 moves forward.
Car 13 clears the accident.
Car 23 moves forward.
```

Car 14 moves forward.
Car 24 moves forward.
Car 15 moves forward.
Car 16 lets car 26 in.
Car 23 clears the accident.
Car 14 moves forward.
Car 24 moves forward.
Car 15 moves forward.
Car 26 lets car 25 in.
Car 14 clears the accident.
Car 24 moves forward.
Car 15 moves forward.
Car 25 lets car 35 in.
Car 36 moves right.
Car 24 clears the accident.
Car 15 moves forward.
Car 35 lets car 33 in.
Car 36 lets car 34 in.
Car 15 clears the accident.
Car 33 lets car 31 in.
Car 34 lets car 32 in.
Car 31 clears the accident.
Car 33 moves forward.
Car 35 moves forward.
Car 25 moves forward.
Car 26 moves forward.
Car 16 moves forward.
Car 33 clears the accident.
Mike’s car will exit after 10 cars go in front of it.

The Shawshank Transaction

Filename: OVERDRAFT

After years of carefully balancing his checkbook, laundering money in offshore bank accounts, and avoiding paying any income tax whatsoever, Ali has been nailed by the United States Internal Revenue Service. He's now serving a short term in a minimum security penitentiary in Maine. He's a quiet, unassuming, and most importantly very intelligent inmate. He's realized that he can earn extra privileges and benefits for himself and his friends… *if* he employs his unique talents for the guards. His first task is to create a checking account overdraft monitor for Warden Norton and Captain Hadley, the most powerful men in the prison. If he's successful, Ali will be quite a bit more comfortable during his institutional years. If he fails, on the other hand… well, he'd better not fail.

The Problem:

You will be given two balances: a checking account balance and a savings account balance, followed by a series of checking withdrawals. You are to apply these transactions to the checking balance. If the checking balance reaches zero, you are to deduct any overage, and any further transactions from the savings balance. This situation is known as an overdraft.

The Input:

There will be multiple data sets. The first line of each data set will be the initial checking balance ($0 \leq c \leq 1000000$). The second line will be the initial savings balance ($1 \leq s \leq 1000000$). The third line will be the number of transactions, n ($1 \leq n \leq 10$), followed by n lines, each representing a withdrawal ($0 < w \leq 100000$). All amounts will be integers. There will never be more withdrawn than the sum of the balances (withdrawals $\leq c+s$). The final data set will be represented by three zeroes. This case should not be processed.

The Output:

You are to output three lines per case, with each case followed by a blank line. The first line should be the final checking balance. The second line should be the final savings balance. The third line should be "OVERDRAFT" if an overdraft occurred, and "CLEAR" otherwise.

Sample Input:

```
2000
3000
2
4000
550
612354
100034
4
100000
12324
1001
4000
0
0
0
```

Sample Output:

```
0
450
OVERDRAFT

495029
100000
CLEAR
```

Springfield Skippy and the Cymbal of Doom

Filename: CYMBAL

Back in 1991, we followed Springfield Skippy as he thwarted the attempt of the evil Nat Malloy to disrupt Skippy's teaching of geometry by stealing his prized geometry teaching tool, the Golden Circle. Since that time, Skippy has been forced to also be the school's bandleader due to budget cutbacks. Not doing anything less than 100%, Skippy has planned a grand band recital for the parents of his students highlighting the cymbal (hey, it's circular!). Nat Malloy heard wind of this and swapped one of the school's prized cymbals with his own Cymbal of Doom! This Cymbal of Doom creates such a hideous sound that it will totally wreck Skippy's planned recital. Fortunately, Skippy has learned that the evil Malloy has taken the school's cymbal back to his hide-out in an abandoned mine. He has infiltrated the mine and recovered his cymbal, but he is lost in a mine car and is being chased by Malloy!

The Problem:

Given a description of the mine car tracks including the switches, determine how Springfield Skippy should set each switch track so that his car will exit the mine the quickest and output the path he follows with the necessary switch changes. Switches can either have one entering track and multiple exit tracks (the switch is splitting the track into multiple possible destinations), or multiple entering tracks and one exit track (the switch is combining multiple sources). The track system within the mine is running downhill so no part of the track will ever backtrack deeper into the mine.

The Input:

The first line will contain a positive integer indicating the number of mine track systems to check. Each track system will start with a line with a single integer representing the total number of switches within the system. A switch will always have either a single entering track or a single exiting track. A single line will represent each track switch within a system. The line begins with two integers that represent the current setting of the switch and the number of tracks exiting the switch, respectively. The current setting of the switch will be the track from the "previous" switch for switches with multiple possible entering tracks going to a single exit track (in other words, which source is being selected by the switch) or will be the track for the "next" switch for switches with a single entering track going to multiple possible exiting tracks (in other words, which destination is being selected by the switch). Following the first two integers on the line will be a list of switches (each an integer) to which it connects. Switches are numbered in the order they appear in the input beginning with one. The switch at which Skippy begins will be the one with zero entering tracks and the exit where he needs to go will be the one with zero exiting tracks.

The Output:

For each track system, determine which switches must be changed and to what they need to be set so that Skippy gets out the quickest. The quickest way out is the one requiring the fewest switch changes (if there are multiple paths with the same fewest switch changes, output any one of them). Output a header for the track system (starting with 1) and the path Skippy should

follow. If a switch needs to be changed, output to what setting it should be changed surrounded by parenthesis as Skippy passes through it (the parenthetical expression should be output before the switch number itself for entering switches and after the switch number for exiting switches). Follow the format illustrated in to the Sample Output and leave a single blank line between the output from each mine track system.

Sample Input:

```
3
6
2 2 2 3
4 2 4 5
5 2 4 5
2 1 6
3 1 6
4 0
8
2 3 2 3 4
6 2 5 6
6 2 5 6
6 2 6 7
2 1 8
3 1 7
4 1 8
5 0
8
2 3 2 3 4
5 2 5 6
6 2 5 6
6 2 6 7
3 1 8
3 1 7
4 1 8
5 0
```

Sample Output:

```
Track System 1:
1 2 4 6

Track System 2:
1 2(5) 5 8

Track System 3:
1 2 (2)5 8
```