ACM Southeast Regional
Intercollegiate Programming Contest

Saturday, 28 October 2006

Georgia Southern University
University of South Alabama

Problem Set

Digital Reversal Addition Method (DRAM) (Purple)

Divisibility (Blue)

Folding Colored Cubes (Orange)

Lawn Mowing (Dark Green)

Logical Mazes (White)

Pebbles (Red)

The Pushy Blockhead (Light Green)

Tracking (Yellow)

Trick or Treat! (Silver)

Whoop-de-doo (Pink)

Digital Reversal Addition Method (DRAM)

Input file: `dram.in`

Pick a number, any number. Reverse its digits. Add the new number to the original. Repeat the process. Do this enough times and you will likely arrive at a palindrome (a number that reads the same forwards and backwards). For example starting with 23 the sequence looks like:

```
   23   the original number
+  32   reverse its digits
 ----
   55   a palindrome
```

Sometimes this may take multiple repetitions. Consider the number 99988877:

```
   99988877  the original number
+  77888999  reverse its digits
 ----------
  177877876  not a palindrome
+ 678778771  reverse the result's digits
 ----------
  856656647  not a palindrome
+ 746656658  etc.
 ----------
 1603313305
+5033133061
 ----------
 6636446366  a palindrome
```

Notice that although all the original numbers you are given will be representable by the standard integer types, the result may be much larger than the largest integer representable by the standard integer types in your programming language.

Input:

The input will consist of one or more integers, one per line. The last line will have the value -1, which should not be processed. Each other integer will be a value greater than or equal to 1 and will represent a separate test case.

Output:

For each test case, echo the input value and give either the first palindrome that results from the DRAM process or a message that no such palindrome exists if there is not a palindrome in the first 1,000 iterations of the process. Use the format in the sample output below.

Sample Input:

```
9
23
99988877
9898
-1
```

Output Corresponding to Sample Input:

```
Initial value: 9 gives palindrome 9
Initial value: 23 gives palindrome 55
Initial value: 99988877 gives palindrome 6636446366
Initial value: 9898 No palindrome found
```

Divisibility

Input: `divisible.in`

Your little brother, Joey, has recently learned that if the sum of digits in a number is divisible by 3 the number is divisible by 3. Joey now thinks that all numbers that are divisible by the sum of their digits are divisible by 3.

You tell Joey that the number 2 is divisible by the sum of its digits but not divisible by 3, so Joey decides all numbers *greater than 10* that are divisible by the sum of their digits are divisible by 3. You think fast and tell Joey that the number 20 is divisible by the sum of its digits but not divisible by 3, so Joey decides all numbers *greater than 20* that are divisible by the sum of their digits are divisible by 3.

You're quickly getting tired of doing math, so decide to write a program that will find the smallest number greater than a given number that is divisible by the sum of its digits, but not divisible by 3.

Input

The input to this program will be one or more lines with one integer n per line, $0 \le n \le 1{,}000{,}000$.

End of input will be marked by $n = 0$. This value should not be processed.

Output

For each n, print the smallest value, *result*, that is greater than or equal to n that is divisible by the sum of its digits but not divisible by 3. Use the format:

`For` *n*`, tell Joey` *result*`.`

Sample Input

```
9
10
11
143
0
```

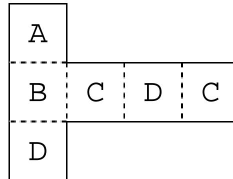
Sample Output (corresponding to sample input)

```
For 9, tell Joey 10.
For 10, tell Joey 10.
For 11, tell Joey 20.
For 143, tell Joey 152.
```

Folding Colored Cubes

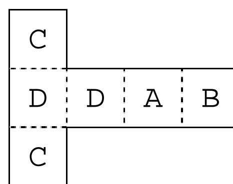
Input file: `cubes.in`

A piece of cardboard of the shape given below can be folded at the dashed lines into a cube.



The faces are colored on one side (top as shown) with distinct colors symbolized by uppercase letters of the alphabet. The faces on the other side (bottom as shown) will be on the inside of the cube, and thus be hidden and immaterial.

Another piece of cardboard with the same shape, but with faces colored as follows, will form a cube that is identical (in all but orientation).



Given pairs of cardboard pieces, all of the shape above, with specified face colors, we would like to know if the cubes are identical.

Input:

Each piece of unfolded cardboard is specified by three lines of input, having respectively one, four, and one uppercase alphabetic character(s), representing the distinct colors on the six faces. Each data set contains two sets of 3-line specifications. The end of data is the end of the file.

Output:

For each data set, output a line looking like

```
Cube pair <n> is <d>
```

Where `<n>` is the dataset starting at 1, and `<d>` is the word "`identical`" or "`different`" as appropriate.

Sample Input:

```
A
BCDC
D
C
DDAB
C
A
BCDC
D
C
DADB
C
A
BDEF
C
F
EABC
D
```

Output Corresponding to Sample Input:

```
Cube pair 1 is identical
Cube pair 2 is different
Cube pair 3 is different
```

Lawn Mowing

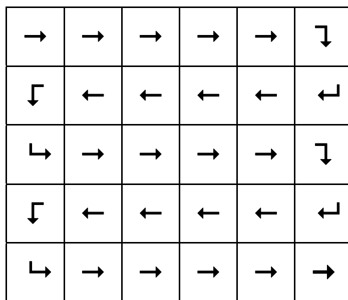
Input file: `lawn.in`

All the lawns I mow are rectangular, having a width and length measured in discrete units corresponding to the cutting diameter of my lawn mower blade. Thus, the lawns can be thought of as made up of unit squares. The first two lawns in the example diagram below have a width of 6, and a length of 5.

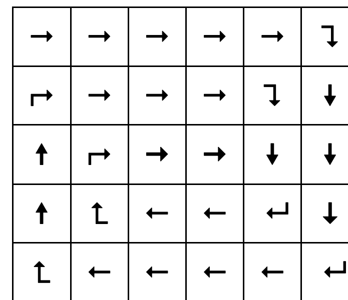
My self-propelled lawn mower takes time t_s to traverse a unit square while moving in a straight direction, t_c when I have to make a 90° turn, and t_r is the total time to make two consecutive 90° turns. (I don't worry about missing corners of the unit areas due to circular cutting.)

I can mow the lawn in either a back-and-forth pattern, or in a spiral pattern (see figures below), but I always want to do it in the least time. To help me to decide which pattern to use, I need a program to calculate the time required to mow the lawn using either one of the patterns.

I always start in the upper-left corner, initially facing in the rightward direction. The last square mowed never involves a turn (unless it has to).

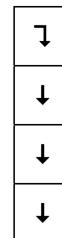


Back-and-forth Pattern. Here, the time needed to mow this area is $22 \times t_s + 4 \times t_r$. The last area mowed is bottom-right.



Spiral Pattern. Here, the time needed to mow this area is $22 \times t_s + 6 \times t_c + t_r$. The last area mowed is near the middle of the lawn.

In special cases when the yard is just one unit wide or long, the back-and-forth and spiral patterns will be identical. If necessary, I will turn 90 degrees and then mow the length or width of the yard.



Input:

The input comprises a sequence of data sets. Each data set gives, on a single line, values for *w* (width of lawn), *l* (length of lawn), t_s, t_c, and t_r. All values are integers such that $0 \le w, l \le 200$, $0 < t_s, t_c, t_r \le 100$. The input terminates if either *l* or *w* is zero, and that data set should not be processed.

Output:

Each line of the output corresponds to an input data set. There are two numeric values and a text label on each line, each separated by a single space. The first value is the time required to mow the area in a back-and-forth pattern. The second is the time required to mow the area in a spiral pattern. In each case, assume you start mowing at the top-left corner, initially facing toward the right. The third item on the line is the string "`Spiral`" if the spiral pattern requires the least time, "`Back-and-forth`" if the back-and-forth pattern requires the least time, or otherwise "`TIE`".

Sample input:

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

Output Corresponding to Sample Input:

```
29 32 Back-and-forth
34 34 TIE
24 31 Back-and-forth
35 33 Spiral
```

Logical Mazes

Input file: `mazes.in`

You have probably all seen maze problems where the maze is described using ASCII characters. For example, a maze might be described as:

```
.XX
...
XX.
```

where '`.`' represents a free space in the maze and '`X`' represents a wall.

But there are other ways to represent mazes that are based on the logical structure of the maze instead of its appearance. One such way uses parentheses and underscores to represent the structure of a maze. These are used as follows:

- '_': a location occupied by a wall;
- '`(`' *north-loc-desc* '`,`' *west-loc-desc* '`,`' *south-loc-desc* '`,`' *east-loc-desc* '`)`': an empty location that has not been described previously. Within the set of parentheses will be four other descriptions, defining the neighbors of this empty location;
- '`*`': an empty location that has already been described.

Any space outside the boundaries of the maze will be represented as containing a wall, but these extra walls will not be shown in the final maze. So the description `(_,_,_,_)` describes an empty location with walls to its north, west, south, and east, or the maze:

```
.
```

(That's just a single empty location.)

A slightly more complicated maze is represented by the description `(_,_,(*,_,_(_,*,_,_)),_)`. Here there are walls to the north and the west of the starting empty location. The cell to the south of the starting location is described by `(*,_,_(_,*,_,_))`. The cell to the north is a cell that's previously been described. There are walls to the west and south. The cell to the east is defined by `(_,*,_,_)`, an empty cell with walls to the north, south, and east and a previously described location to the west. All of this description ultimately gives the maze:

```
.X
..
```

You are to write a program that will translate from the parenthesized description of a maze to an ASCII version of the maze.

Input:

The input to your program will be one or more data sets, each with a description of a maze, in the parenthesized version described above. The first line of each data set will be a pair of integers, *row* and *column*, with $0 \leq row \leq 20$, $0 \leq column \leq 20$, giving the overall maze dimensions. The values row and column will only be equal in the maze indicating the end of input. The second line of each data set will be a pair of integers *startrow* and *startcolumn*, with $0 \leq startrow < row$, $0 \leq startcolumn < column$, giving the location initially being described, where (0, 0) is the upper left hand location of the maze. There will then be a syntactically valid description of the starting location and adjacent locations. There may be extra blanks and ends of line in the data.

The last input set will have the *row* and *column* equal to 0. This set should not be processed.

Output:

For each maze, first output a line with the number of the maze, formatted as in the sample output and starting with 1, then the ASCII version of the maze, using 'X' to represent a wall and '.' to represent an empty location. You should assume all descriptions are valid and any space not described is a wall.

Sample Input:

```
5 3
0 1
(_,_,_,_)
5 5
2 2
(_,_,(*,_,_(_,*,_,_)),_)
3 3
0 0
( _,
  _,
  ( *,
    _,
    _,
    ( _,
      *,
      _,
      ( _,
        *,
       (*,_,_,_)
       ,_)
      )
    ),
  _)
0 0
```

Output Corresponding to Sample Input:

```
Maze number 1
X.X
XXX
XXX
XXX
XXX
Maze number 2
XXXXX
XXXXX
XX.XX
XX..X
XXXXX
Maze number 3
.XX
...
XX.
```

Pebbles

Input file: `pebbles.in`

In an old Chinese game, there is some number of piles of pebbles. You and your opponent take turns in removing any positive number of pebbles from a single pile. The objective is to be in a position to remove the last pile.

There is a winning strategy: express the number of pebbles in the piles in binary and see to it that, after your turn, the total number of non-zero digits in any binary place value is even. Of course, if this is already true when it is your turn, it will no longer be true after your turn. In that case, you simply remove one pebble from the first largest pile (and hope that your opponent doesn't know the winning strategy!).

Say that 4 piles have the following numbers of pebbles:

$$
\begin{array}{rcl}
8 & = & 1000_2 \\
9 & = & 1001_2 \\
3 & = & 0011_2 \\
11 & = & 1011_2
\end{array}
$$

The first, and last binary place values (columns) have an odd number of non-zero digits. There are three ways to make all place values even: reduce the first pile to 1, reduce the second pile to zero, or reduce the fourth pile to 2. To provide a unique answer, we will specify that we should always take pebbles from the first largest pile for which it is possible to make all place values even. Thus, in the above case, you should remove 9 pebbles from the fourth pile.

We would like to have a program that tells us what to do on our turn for various game situations.

Input:

For each game situation, the first line of input has a single integer, n, being the number of piles ($0 \le n \le 5$). The next n lines each contain a single positive integer (≤ 1000000000) representing the number of pebbles in the pile. The last case is when $n = 0$, and should not be processed.

Output:

The output line should look like

```
<p> pebbles should be removed from the <i> pile.
```

Where `<p>` is the number of pebbles to be removed, and `<i>` is the appropriate word from the list "`first`", "`second`", "`third`", "`fourth`", or "`fifth`".

Sample Input:

```
4
8
9
3
11
4
3
3
9
9
3
2
10
10
0
```

Output Corresponding to Sample Input:

```
9 pebbles should be removed from the fourth pile.
1 pebbles should be removed from the third pile.
2 pebbles should be removed from the second pile.
```

The Pushy Blockhead

Input file: `push.in`

You have a rectangular grid of cells, each of which can contain:

- an unmovable wall *brick* (■ in the diagram below), or an unmoveable *hole* (☒)
- a moveable *pusher* (◇),
- a moveable *block* (□),
- nothing.

Cells containing a brick cannot contain anything else. Bricks exist in at least all border cells of the grid. In any initial configuration, there is exactly one cell containing a pusher, and exactly one (different) cell containing a block. Since the block is smaller than a hole, it will fall into it (and disappear), if it is moved to a hole (i.e., a cell having a hole). Since the pusher is larger than the hole, it can move into and over a cell having a hole without falling in. The pusher is allowed to move into a horizontally or vertically adjacent cell, unless that cell contains a brick. If the pusher is moved into a cell containing the block, the block moves one cell over in the same direction (i.e., the block is 'pushed' by the pusher). However, a pusher move is not allowed if it would push a block into a cell containing a brick. The goal is, by a sequence of allowed pusher moves, to eventually push the block to a specified position, (without, of course, the block having fallen into a hole).

Consider the following initial grid configuration below, in which the goal is to move the block from initial position[1] (2,3) to a target position at (1,1). This is possible by moving the pusher to (1,3) via an existing path of empty cells. The pusher can then be moved downward to (3,3), at the same time pushing the block downward to (4,3). After moving the pusher from (3,3) to (4,4) via a path of empty cells, the block can then be pushed to (4,2). The pusher can then be moved to (5,2), then to (4,2), pushing the block to (3,2). The pusher is then moved to (3,3) and then to (3,2), pushing the block to (3,1). Finally, the pusher is moved to (4,1) (over the hole), and then to (2,1), pushing the block to its target destination, (1,1).

	0	1	2	3	4	5
0	■	■	■	■	■	■
1	■				■	■
2	■	◇	■	o	■	■
3	■				■	■
4	■	☒				■
5	■					■
6	■	■	■	■	■	■

Now consider again the same grid configuration as above, but where the target position is (1,2). The goal is not achievable (try it!).

For given initial grid configurations, we would like to know if it is possible to push the object block into a given target position. That is, we want to know if there exists at least one way to "solve" the configuration.

[1] Using (row, column) coordinates

Input:

Each grid configuration starts with a line containing two integers, $0 \le r, c \le 100$, being the number of rows, and the number of columns respectively. The last configuration has $r = 0$ and $c = 0$, and should not be processed. The following r lines contain strings of c characters taken from the set {'W', 'H','.'} representing respectively cell contents: (wall) brick, hole, empty cell. The next three lines each contain a pair of integers representing respectively the initial position (i.e., zero-based row and column indices) of the pusher block, the initial position of the object block, and the target position, all of which are within the range of the grid. The initial and final positions of the block will not be coincident with a brick or a hole. The initial pusher position will not be coincident with a brick.

Output:

Each configuration should produce a line like

```
Configuration <n> can be solved.
```

or

```
Configuration <n> can not be solved.
```

where `<n>` is the configuration number starting at 1.

Sample Input:

```
7 6
WWWWWW
W...WW
W.W.WW
W...WW
WH...W
W....W
WWWWWW
2 1
2 3
1 1
7 6
WWWWWW
W...WW
W.W.WW
W...WW
WH...W
W....W
WWWWWW
2 1
2 3
1 2
0 0
```

Output Corresponding to Sample Input:

```
Configuration 1 can be solved.
Configuration 2 can not be solved.
```

Tracking

Input file: `track.in`

The local park has two parallel walking tracks, each of which is booked on Saturday mornings by two walking clubs; the Greater Strolling Unicorns (GSU), and the Unparalleled Speedwalking Association (USA). Both of these tracks have a common entrance that also serves as its exit, and requires all members to walk the same direction. Each member of both clubs is totally consistent and always takes the same amount of time to complete a lap.

USA members are super competitive, and are a little more happy each time they pass a GSU member, and a little less happy each time they are passed by a GSU member. Since members of both clubs don't slow down or speed up, USA has figured out that they can maximize their members' happiness by having them change their starting time. Of course, they can't change the start timing too much, but they've discovered members are willing to stretch up to 5 minutes longer, so can change the start time by at most that much.

USA wants to try optimizing each individual member's happiness by changing his/her start time and is asking you to write a program to determine that member's optimal start time, given a group of GSU members using the other track.

Input

The input to this program will be a series of scenarios Each scenario will begin with a single line, giving the scenario description, a string. There will then be a line containing a single integer, *gsu*, $0 \leq gsu \leq 100$, the number of people in GSU using the first track. There will then be *n* lines, each in the form:

start-time lap-time laps

Each line describes a single person on the track. The start-time is given in *minutes*:*seconds* where $0 \leq minutes \leq 600$, $0 \leq seconds \leq 59$. The lap-time is the time to complete one lap, also in *minutes*:*seconds* format and laps is a positive integer less than 100. The list of *gsu* people using the track will be in order of start time.

After the list of GSU people using the track will be a line containing a single integer, *usa*, the number of USA people using the second track for whom we are trying to optimize happiness. After this line there will be *usa* lines, each in the form:

arrival-time lap-time laps

Each line describes a person who arrives to walk, but is willing to wait up to 5 minutes if it means passing more people and/or being passed by fewer. The arrival-time, lap-time, and laps are formatted the same as for GSU people.

End of input is indicated by end of file.

Output

For each scenario, echo the scenario description. Then, for each of the *usa* USA people looking for happiness, compute the maximum possible happiness where happiness is defined to be the number of GSU people passed minus the number of GSU people who pass this person. If a person enters or leaves the track at the same time they would pass or be passed by another person, one person is ***not*** considered to have passed the other. If there is more than one possible start time that results in the same happiness, pick the earliest. Print a line with the person number, the best start time, and the happiness score, using the format in the sample output below.

Have a blank line after the data for each track.

Sample Input

```
Some slow strollers
3
15:00 4:00 6
17:00 5:00 2
27:00 6:00 4
3
10:00 2:00 6
14:00 3:45 5
27:30 2:00 3
Very short track
4
0:00 0:10 60
0:00 0:20 30
11:00 2:00 5
12:00 3:00 10
2
0:00 1:00 1
9:00 0:30 10
```

Output Corresponding to Sample Input

```
Some slow strollers
Member 1: start at 14:00 to get happiness 7.
Member 2: start at 15:15 to get happiness 3.
Member 3: start at 29:00 to get happiness 5.

Very short track
Member 1: start at 0:00 to get happiness -5.
Member 2: start at 12:30 to get happiness 20.
```

Trick or Treat!

Input file: `treat.in`

Halloween is coming soon and you've been asked to create a program to help parents and children get just what they want. Parents want to give children an upper bound on the amount of candy they can receive and children want as much candy as they get without upsetting their parents.

Fortunately, in our neighborhoods, we know in advance exactly how many pieces of candy each home hands out to the children. A child has to take all the candy given by the home so they don't seem rude, and they can't throw away or eat any candy on the way. Children also have to stop at every home in the sequence of homes.

Given these conditions, write a program to find the best sequence of homes for children to visit.

Input:

The input will consist of information about one or more neighborhoods. The first line contains a single integer, *homes*, $0 < homes \leq 100$, that represents the maximum number of homes on the block. The next line contains a single integer, *max*, $0 \leq max \leq 100$, representing the maximum number of pieces of candy that the child may collect. There will then be *homes* lines, each containing an integer, *pieces*, $0 \leq pieces \leq 100$, representing the number of pieces given at each home. Homes are numbered consecutively starting at 1.

The last neighborhood will be followed by a line containing just the value 0. This line should not be processed.

Output:

For each neighborhood, first give the number of the neighborhood, starting with 1, using the format shown in the sample output below. Then, if there is no way to select one or more consecutive homes where the sum of pieces of candy is $\leq max$ on a single line, print "`Don't go here`". Otherwise, in the format shown below, for the sequence of homes that yield the largest number of pieces of candy $\leq max$, include the number of the first home to visit, the number of the last home to visit, and the sum of pieces of candy.

If there is more than one such sequence of homes, select the one with the lowest numbered first home. Have a blank line after the data for each neighborhood.

Sample Input:

```
5
10
2
4
3
2
```

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

Output Corresponding to Sample Input:

```
Neighborhood 1
Start at home 2 and go to home 5 getting 10 pieces of candy

Neighborhood 2
Don't go here

Neighborhood 3
Start at home 1 and go to home 3 getting 7 pieces of candy
```

Whoop-de-doo

Input file: `whoop.in`

To encourage people to live up to its name—Convivialville—the town has inaugurated the Whoop-de-doo club. The by-laws of the club are as follows:

I. Membership is restricted to two-handed people over the age of 16.

II. The club will meet at or about 8 PM on Saturdays in the local fire station.

III. The first member to arrive will not participate in the fun (see below), but will be secretary for the current meeting [hint: don't arrive too early!]. The duty of secretary is preparing a box of 26 name tags labeled only with the upper-case characters A though Z (to keep anonymity), and making sure that at least 26 blindfolds are available

IV. As members arrive they will randomly choose a name tag from the box. In the rare event the box is empty, those members will be turned away [hint: don't arrive too late!]

V. At some point in the evening, the secretary blows one blast on a whistle, and everyone puts on a blindfold. Then, when everyone is blindfolded, the secretary blows two blasts, everyone has to move around, and organize themselves such that each hand is holding the hand of a different person.

VI. When all hands are held, the secretary quickly writes down (for the minutes of the meeting) all pairs of hands held. The notation "A B" represents that the person with name tag A holds the hand of the person with name tag B. This fact could equivalently have been noted as "B A" (duh…)

VII. The secretary then determines from his/her minutes whether the handholding group forms a single cyclic chain. If it is, the secretary gives a triple blast, and everyone yells "Whoop-de-doo!" Otherwise, the secretary blows one long blast, and everyone says a mournful "Ohh".

VIII. Then everyone removes their blindfolds, has some more fruit punch, and, in any case, goes home with an added sense of conviviality.

Your job is to make the secretary's job easier by determining, from the notations in the minutes, whether everyone forms a single cyclic chain. Unfortunately, the secretary in his/her eagerness not to overlook any handholding pairs may accidentally include redundant information. In spite of this eagerness, occasionally, he/she may forget to note a handholding pair (but never more than one).

Input:

For each meeting, the first line contains a single integer, $3 \le n \le 100$, represents the number of notations for pairs of hands held. This is followed by n lines each containing two different upper-case letters separated by a single space (using the notation described in by-law VI) , representing a pair of held hands. A value of $n = 0$ represents the end of data.

Output:

For each meeting there should be an output line looking like

```
Meeting <x>: Whoop-de-doo! <list>
```

or

```
Meeting <x>: Ohh
```

where <x> is the meeting number starting with 1, and <list> is a string representing the chain of people. This list should be the lexicographically smallest string.

Sample Input:

```
6
F B
P Y
B P
F B
B F
F Y
6
A B
B A
F B
C D
D E
E C
2
A B
B C
7
A B
B C
D F
D E
E F
E F
F E
0
```

Output Corresponding to Sample Input:

```
Meeting 1: Whoop-de-doo! BFYP
Meeting 2: Ohh
Meeting 3: Whoop-de-doo! ABC
Meeting 4: Ohh
```