2006 South Central USA Regional Programming Contest

Rounders

Introduction:

For a given number, if greater than ten, round it to the nearest ten, then (if that result is greater than 100) take the result and round it to the nearest hundred, then (if that result is greater than 1000) take that number and round it to the nearest thousand, and so on ...

Input:

Input to this problem will begin with a line containing a single integer *n* indicating the number of integers to round. The next *n* lines each contain a single integer *x* (0 <= *x* <= 99999999).

Output:

For each integer in the input, display the rounded integer on its own line.

Note: Round up on fives.

Sample Input:

```
9
15
14
4
5
99
12345678
44444445
1445
446
```

Sample Output:

```
20
10
4
5
100
10000000
50000000
2000
500
```

2006 South Central USA Regional Programming Contest

Q

Introduction

You've got a queue. And you just got to mess with it.
Given a queue of items and a series of queue operations, return the resulting queue.
Queue operations are defined as follows:

```
starting-position to requested-position
```

meaning one wants the item at the starting position to be moved to the requested position. So if the queue of items were:

```
Item1 Item2 Item3 Item4 Item5
```

(Item1 being in position 1, Item2 in position 2, etc.)

after applying the queue operation:

5 to 2

the resulting queue would be:

```
Item1 Item5 Item2 Item3 Item4
```

as Item5 (the item in position 5) was moved to position 2. Multiple queue operations are applied at the same time, however; e.g., given the queue of items:

```
Item1 Item2 Item3 Item4 Item5 Item6 Item7 Item8
```

If the following queue operations were applied:

```
2 to 6; 6 to 3; 4 to 5; 5 to 2; 7 to 4; 8 to 1
```

then the resulting queue would be:

```
Item8 Item5 Item6 Item7 Item4 Item2 Item1 Item3
```

As you can see, the queue operations are strictly enforced, with other items (not involved in queue operations) maintaining their order and moving to vacant positions in the queue. Note that no two queue operations will have the same *starting-position* or same *requested-position* defined.

Input

Input to this problem will begin with a line containing a single integer *x* indicating the number of datasets. Each data set consists of three components:

1. Start line – A single line, "*m n*" (1 <= *m, n* <= 20) where *m* indicates the number of items in the queue and *n* indicates the number of queue operations.
2. Queue items – A line of short (between 1 and 8 characters) alphanumeric names for the items in the queue. Names are unique for a given data set and contain no whitespace.
3. Queue operations – *n* lines of queue operations in the format "`starting-position requested-position`".

Output

For each dataset, output the queue after the queue operations have been applied. Print the elements of the queue on a single line, starting from the first and ending with the last, with a single space separating each item.

Sample Input

```
3
5 1
alpha beta gamma delta epsilon
5 2
8 6
a b c d e f g h
2 6
6 3
4 5
5 2
7 4
8 1
3 2
foo bar baz
3 1
1 3
```

Sample Output

```
alpha epsilon beta gamma delta
h e f g d b a c
baz bar foo
```

2006 South Central USA Regional Programming Contest

Enigmatologically Cruciverbalistic

Introduction:

The local ACM Newsletter has decided to start running a crossword puzzle in each issue. Instead of purchasing premade puzzles, however, the editors would like to be able to make puzzles in various shapes and with words of their own choosing. They've turned to you for the development of the puzzles ... and as a programmer, you've decided to turn to your computer and solve the problem once and for all.

A crossword grid looks like this:

```
..#......
#.#......
#########
#.#.#...#
#.#.....#
######..#
#.#......
```

Each continuous row or column of two or more hash marks represents a single word, one letter per mark. Where rows and columns cross, the letter in the shared mark must be the same.

Your goal is to write a program that fills a given grid with a given set of words such that:

- every mark in the grid contains a letter;
- no other cells in the grid contain a letter;
- every word in the word list appears once and only once in the grid with no unused words left over; and
- no words (or other sequences of letters) not in the word list appear in the grid

Words inside of other words (BRIGHT inside of BRIGHTLY, for example) do not count as an appearance of the sub-word. You may assume that every word in the word list is unique in that puzzle. You may also assume that there is at most one possible layout for a given grid and word list.

Input:

The first line of data will be a number representing the number of datasets in the file. For each dataset, the first line consists of two integers *w h* (2 <= *w*, *h* <= 15) where *w* is the width of the puzzle and *h* is the height. The next *h* lines of the dataset are a representation of the crossword puzzle grid, as shown above. All word spaces in the puzzle will be at least two characters in length. The next line consists of an integer *c* (1 <= *c* <= 100) representing the number of words in the crossword. The next *c* lines contain the words in the word list.

Output:

For each dataset in the input, output the heading "`Puzzle #x`", where *x* is 1 for the first dataset, 2 for the second, and so on. Then print either "`I cannot generate this puzzle.`" if the puzzle is impossible to generate given that grid and word list, or print a solved representation of the puzzle as shown below.

Sample Input:

```
2
9 7
..#......
#.#......
#########
#.#.#...#
#.#.....#
######..#
#.#......
6
COMPUTE
LAMPSHADE
EDIT
SO
PLAYER
ESTEEM
5 5
#####
#.#.#
#####
#.#.#
#####
6
ADAGE
ANGRY
YEARN
NEEDS
FLUTE
XYZZY
```

Sample Output:

```
Puzzle #1
..C......
P.O......
LAMPSHADE
A.P.O...D
Y.U.....I
ESTEEM..T
R.E......
Puzzle #2
I cannot generate this puzzle.
```

2006 South Central USA Regional Programming Contest

Blue Jeans

Introduction:

The Genographic Project is a research partnership between IBM and The National Geographic Society that is analyzing DNA from hundreds of thousands of contributors to map how the Earth was populated.

As an IBM researcher, you have been tasked with writing a program that will find commonalities amongst given snippets of DNA that can be correlated with individual survey information to identify new genetic markers.

A DNA base sequence is noted by listing the nitrogen bases in the order in which they are found in the molecule. There are four bases: adenine (`A`), thymine (`T`), guanine (`G`), and cytosine (`C`). A 6-base DNA sequence could be represented as `TAGACC`.

Given a set of DNA base sequences, determine the longest series of bases that occurs in all of the sequences.

Input:

Input to this problem will begin with a line containing a single integer *n* indicating the number of datasets. Each dataset consists of the following components:

1. A single positive integer *m* (2 <= *m* <= 10) indicating the number of base sequences in this dataset.
2. *m* lines each containing a single base sequence consisting of **60** bases.

Output:

For each dataset in the input, output the longest base subsequence common to all of the given base sequences. If the longest common subsequence is less than three bases in length, display the string "`no significant commonalities`" instead. If multiple subsequences of the same longest length exist, output only the subsequence that comes first in alphabetical order.

Sample Input:

```
3
2
GATACCAGATACCAGATACCAGATACCAGATACCAGATACCAGATACCAGATACCAGATA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
3
GATACCAGATACCAGATACCAGATACCAGATACCAGATACCAGATACCAGATACCAGATA
GATACTAGATACTAGATACTAGATACTAAAGGAAAGGGAAAAGGGGAAAAAGGGGGAAAA
GATACCAGATACCAGATACCAGATACCAAAGGAAAGGGAAAAGGGGAAAAAGGGGGAAAA
```

```
3
CATCATCATCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
ACATCATCATAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AACATCATCATTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
```

Sample Output:

```
no significant commonalities
AGATAC
CATCATCAT
```

2006 South Central USA Regional Programming Contest

Core Wars

Introduction:

Core Wars is a game in which two opposing warrior programs attempt to destroy each other in the memory of a virtual machine. They do this by overwriting each other's instructions, and the first program to execute an illegal instruction is declared the loser. Each program is written in an assembly-like language called **Redcode**, and the virtual machine which executes the two programs is known as the **Memory Array Redcode Simulator** (MARS). Your goal is to write a MARS that will read in two Redcode programs, simulate them, and print out which program was the winner.

MARS simulates a somewhat unusual environment compared to other virtual machines and processor architectures. The following list describes these differences in detail:

1. MARS simulates a machine with 8000 memory locations and each location stores exactly one Redcode instruction. In fact, a memory location can only store instructions and cannot directly store any data. However, each instruction includes two numeric operands, and these in turn can be manipulated by other instructions for data storage. This also makes self-modifying code possible.
2. The memory locations are arranged as a continuous array with the first location having address 0 and the last having 7999. All address calculations are performed using modulo 8000 arithmetic. Put in another way, memory addresses wrap around so that addresses 8000, 8001, and 8002 refer to same memory locations respectively as addresses 0, 1, and 2. This also works for negative numbers. For example, -7481, -15481, 519, and 8519 all refer to the same memory location.
3. All arithmetic and comparison operations are performed modulo 8000. Additions must normalize their final result to be in the range of 0 to 7999 (inclusive) before writing that result into memory. This also implies that -124 is considered to be *greater* than 511 since after normalization, -124 becomes 7876, and 7876 is greater than 511.
4. The simulator maintains two separate instruction pointers (IPs) that store the address of the next instruction to be executed by the warrior programs. After loading both programs into memory, these IPs are initialized to the first instruction of each program. As the programs run, the IP is incremented by one (modulo 8000) after each instruction is executed. If a jump/skip instruction is executed, then the IP is instead loaded with the destination address of the jump/skip and execution continues from this new address.
5. The simulator "time slices" between warriors by executing one instruction at a time, and alternating between programs after each instruction. For example, if the two programs were loaded at addresses 2492 and 6140, the first six instructions would be executed in this order (assuming no jump/skip instruction were executed): 2492, 6140, 2493, 6141, 2494, 6142.

Every instruction in MARS consists of an opcode, written as a three letter mnemonic, and two operands called the A and B fields. Each operand is a number in the range 0-7999 (inclusive) and each can use one of three addressing modes: immediate, direct, and indirect. These modes are explained in more detail below:

- Immediate operands are written with a "#" sign in front, as in `#1234`. An immediate operand specifies a literal value for the instruction to operate on. For example, the first operand (i.e. the A field) of an `ADD` instruction (which performs integer addition) can be an immediate. In that case, the literal value specified by the first operand provides one of the numbers being added.
- Direct operands identify the memory locations which an instruction is to access. They are written with a "$" sign in front, as in `$1234`. One example would be `ADD #5 $3`. A direct operand is actually an offset relative to the current IP address. For example, if the `ADD #5 $3` instruction were stored in memory location 4357, it would actually be adding together a literal number five with a second value stored in the B field of location 4360 (4357 + 3). However, if that same instruction were stored at location 132 it would be adding five to a value in the B field of location 135 (132 + 3).
- Indirect operands are analogous to how pointers in some programming languages work. Indirect operands are written with a "@" sign in front of the number, as in `ADD #5 @3`. Like before, the indirect operand is an offset relative to the current IP address, and therefore identifies a particular memory location. However, the value stored in the B field of this memory location is then used as an offset relative to *that* location to identify a *second* location. It is the B field of this second location which will actually be operated on by the instruction itself. For example, if location 4357 contained `ADD @1 @3`, location 4358 contained 11 in its B field, and location 4360 contained 7996 in its B field, then this instruction would actually be adding the values stored in locations 4369 (4358 + 11) and 4356 (4360 + 7996 modulo 8000).

The list below explains what each instruction does based on its opcode. Although not all instructions use both of their operands, these must still be specified since other instructions might use these operands for data storage. **Some instructions update the B field of another instruction; this only affects the numerical value of the field, but does *not* change its addressing mode.**

`DAT`
: This instruction has two purposes. First, it can be used as a generic placeholder for arbitrary data. Second, attempting to execute this instruction terminates the simulation and the program which tried to execute it loses the match. This is the only way that a program can terminate, therefore each warrior attempts to overwrite the other one's program with `DAT` instructions. Both A and B operands must be immediate.

`MOV`
: If the A operand is immediate, the value of this operand is copied into the B field of the instruction specified by `MOV`'s B operand. If neither operand is immediate, the entire instruction (including all field values and addressing modes) at location A is copied to location B. The B operand cannot be immediate.

`ADD`
: If the A operand is immediate, its value is added to the value of the B field of the instruction specified by `ADD`'s B operand, and the final result is stored into the B field of that same instruction. If neither operand is immediate, then they both specify the locations of two instructions in memory. In this case, the A and B fields of one instruction are respectively added to the A and B fields of the second instruction, and both results are respectively written to the A and B fields of the instruction specified by the `ADD`'s B operand. The B operand cannot be immediate.

`JMP`
: Jump to the address specified by the A operand. In other words, the instruction pointer is loaded with a new address (instead of being incremented), and the next instruction executed after the `JMP` will be from the memory location specified by A. The A operand cannot be immediate. The B

operand must be immediate, but is not used by this instruction.

`JMZ` If the B field of the instruction specified by `JMZ`'s B operand is zero, then jump to the address specified by the A operand. Neither the A nor B operand can be immediate.

`SLT` If A is an immediate operand, its value is compared with the value in the B field of the instruction specified by `SLT`'s B operand. If A is not immediate, the B fields of the two instructions specified by the operands are compared instead. If the first value (i.e the one specified by A) is less than the second value, then the next instruction is skipped. The B operand cannot be immediate.

`CMP` The entire contents of memory locations specified by A and B are checked for equality. If the two locations are equal, then the next instruction is skipped. Memory locations are considered equal to another if they both have the same opcodes **and** they have the same values and addressing modes in their respective operand fields. The A or B operands cannot be immediate.

Input:

The input begins with a line containing a single integer *n* indicating the number of independant simulations to run. For each simulation the input will contain a pair of programs, designated as warrior number one and warrior number two. Each warrior program is specified using the following format:

One line with integer *m* (1 <= *m* <= 8000) indicating the number of instructions to load for this warrior. A second line containing an integer *a* (0 <= *a* <= 7999) gives the address at which to start loading the warrior's code. These two lines are then followed by *m* additional lines containing the warrior's instructions, with one instruction per line. If the warrior is loaded at the end of memory, the address will wrap around and the instructions continue loading from the beginning of memory.

The address ranges occupied by the two programs will not overlap. **All other memory locations which were not loaded with warrior code must be initialized to `DAT #0 #0`.** Execution always begins with warrior number one (i.e. the warrior read in first from the input file).

Output:

Each simulation continues running until either warrior executes a `DAT` instruction or until a total of 32000 instructions (counting both warriors) are executed. If one warrior program executes a `DAT`, the other is declared the winner; display "`Program #`*x* `is the winner.`", where *x* is either `1` or `2` and represents the number of the winning warrior. If neither program executes a `DAT` after the maximum instruction count is reached, then the programs are tied; display "`Programs are tied.`"

Sample Input:

```
2
3
185
ADD #4 $2
JMP $-1 #0
DAT #0 #-3
5
100
JMP $2 #0
```

```
DAT #0 #-1
ADD #5 $-1
MOV $-2 @-2
JMP $-2 #0
1
5524
MOV $0 $1
5
539
JMP $2 #0
DAT #0 #-1
ADD #5 $-1
MOV $-2 @-2
JMP $-2 #0
```

Sample Output:

```
Program #2 is the winner.
Programs are tied.
```

2006 South Central USA Regional Programming Contest

'Roid Rage

Introduction:

When writing game programs, it is often useful to determine when two polygons intersect one another. This is especially useful in arcade games like *Asteroids* where one polygon could represent a spaceship while another represents a huge, unyielding chunk of space rock.

Write a program that can determine which polygons of a given set intersect one another.

Input:

Input to this problem will begin with a line containing a single integer *n* indicating the number of datasets. Each data set consists of the following components:

1. A line containing a single positive integer *m* (1 <= *m* <= 10) indicating the number of polygons to analyze.
2. *m* lines, each representing a single polygon, with the first line describing polygon 1, the second line describing polygon 2, and so on. Each line begins with a single positive integer *v* (3 <= *v* <= 20) indicating the number of vertices describing this polygon. This is followed by *v* (**x**,**y**) coordinate pairs (0 <= *x, y* <= 100), each of which is a vertex of this polygon. The vertices are connected by edges in the order listed with the last vertex connected back to the first by a final edge. All polygons are "simple"; they do not self-intersect.

Output:

For each dataset in the input, output the heading "`Data Set #z`", where *z* is 1 for the first dataset, 2 for the second, etc. If this data set contained no intersecting polygons, output the message "`no collisions`" on its own line. Otherwise, output the list of all pairs of intersecting polygons, one pair per line, each pair formatted with the lowest-numbered polygon first. Output the polygon pairs in ascending order, sorting first by the lowest-numbered polygon in the set and then the second.

Note: The definition of "intersecting" for the purpose of this problem means that two polygons either share an interior region (i.e., they overlap), or they share boundary points (i.e., they touch at a point or along an edge).

Sample Input:

```
2
2
```

```
4 0,0 1,0 1,1 0,1
4 2,2 3,2 3,3 2,3
4
3 2,1 1,2 2,3
3 2,1 3,2 2,3
5 2,0 4,2 2,4 5,4 5,0
4 3,3 1,3 1,5 3,5
```

Sample Output:

```
Data Set #1
no collisions
Data Set #2
1 2
1 4
2 4
3 4
```

2006 South Central USA Regional Programming Contest

Children of the Candy Corn

Introduction:

The cornfield maze is a popular Halloween treat. Visitors are shown the entrance and must wander through the maze facing zombies, chainsaw-wielding psychopaths, hippies, and other terrors on their quest to find the exit.

One popular maze-walking strategy guarantees that the visitor will eventually find the exit. Simply choose either the right or left wall, and follow it. Of course, there's no guarantee which strategy (left or right) will be better, and the path taken is seldom the most efficient. (It also doesn't work on mazes with exits that are not on the edge; those types of mazes are not represented in this problem.)

As the proprieter of a cornfield that is about to be converted into a maze, you'd like to have a computer program that can determine the left and right-hand paths along with the shortest path so that you can figure out which layout has the best chance of confounding visitors.

Input:

Input to this problem will begin with a line containing a single integer *n* indicating the number of mazes. Each maze will consist of one line with a width, *w*, and height, *h* (3 <= *w*, *h* <= 40), followed by *h* lines of *w* characters each that represent the maze layout. Walls are represented by hash marks ('`#`'), empty space by periods ('`.`'), the start by an '`S`' and the exit by an '`E`'.

Exactly one '`S`' and one '`E`' will be present in the maze, and they will always be located along one of the maze edges and never in a corner. The maze will be fully enclosed by walls ('`#`'), with the only openings being the '`S`' and '`E`'. The '`S`' and '`E`' will also be separated by at least one wall ('`#`').

You may assume that the maze exit is always reachable from the start point.

Output:

For each maze in the input, output on a single line the number of (not necessarily unique) squares that a person would visit (including the '`S`' and '`E`') for (in order) the left, right, and shortest paths, separated by a single space each. Movement from one square to another is only allowed in the horizontal or vertical direction; movement along the diagonals is *not* allowed.

Sample Input:

```
2
8 8
```

```
########
#......#
#.####.#
#.####.#
#.####.#
#.####.#
#...#..#
#S#E####
9 5
#########
#.#.#.#.#
S.......E
#.#.#.#.#
#########
```

Sample Output:

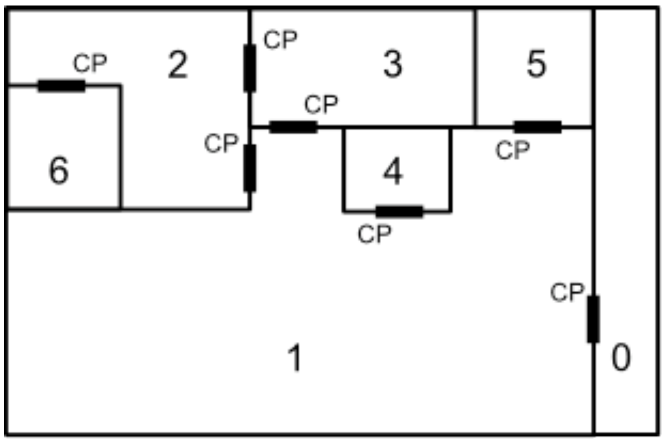
```
37 5 5
17 17 9
```

2006 South Central USA Regional Programming Contest

Panic Room

Introduction

You are the lead programmer for the Securitron 9042, the latest and greatest in home security software from Jellern Inc. (Motto: We secure your stuff so YOU can't even get to it). The software is designed to "secure" a room; it does this by determining the minimum number of locks it has to perform to prevent access to a given room from one or more other rooms. Each door connects two rooms and has a single control panel that will unlock it. This control panel is accessible from only one side of the door. So, for example, if the layout of a house looked like this:



with rooms numbered 0-6 and control panels marked with the letters "CP" (each next to the door it can unlock and in the room that it is accessible from), then one could say that the minimum number of locks to perform to secure room **2** from room **1** is two; one has to lock the door between room **2** and room **1** *and* the door between room **3** and room **1**. Note that it is impossible to secure room **2** from room **3**, since one would always be able to use the control panel in room **3** that unlocks the door between room **3** and room **2**.

Input

Input to this problem will begin with a line containing a single integer *x* indicating the number of datasets. Each data set consists of two components:

1. Start line – a single line "*m n*" (1 <=*m*<= 20; 0 <=*n*<= 19) where *m* indicates the number of rooms in the house and *n* indicates the room to secure (the panic room).
2. Room list – a series of *m* lines. Each line lists, for a single room, whether there is an intruder in that

room ("`I`" for intruder, "`NI`" for no intruder), a count of doors *c* (0 <= *c* <= 20) that lead to other rooms *and* have a control panel in this room, and a list of rooms that those doors lead to. For example, if room 3 had no intruder, and doors to rooms 1 and 2, and each of those doors' control panels were accessible from room 3 (as is the case in the above layout), the line for room 3 would read "`NI 2 1 2`". The first line in the list represents room 0. The second line represents room 1, and so on until the last line, which represents room *m - 1*. On each line, the rooms are always listed in ascending order. It is possible for rooms to be connected by multiple doors and for there to be more than one intruder!

Output

For each dataset, output the fewest number of locks to perform to secure the panic room from all the intruders. If it is impossible to secure the panic room from all the intruders, output "`PANIC ROOM BREACH`". Assume that all doors start out unlocked and there will not be an intruder in the panic room.

Sample Input

```
3
7 2
NI 0
I 3 0 4 5
NI 2 1 6
NI 2 1 2
NI 0
NI 0
NI 0
7 2
I 0
NI 3 0 4 5
NI 2 1 6
I 2 1 2
NI 0
NI 0
NI 0
4 3
I 0
NI 1 2
NI 1 0
NI 4 1 1 2 2
```

Sample Output

```
2
PANIC ROOM BREACH
1
```