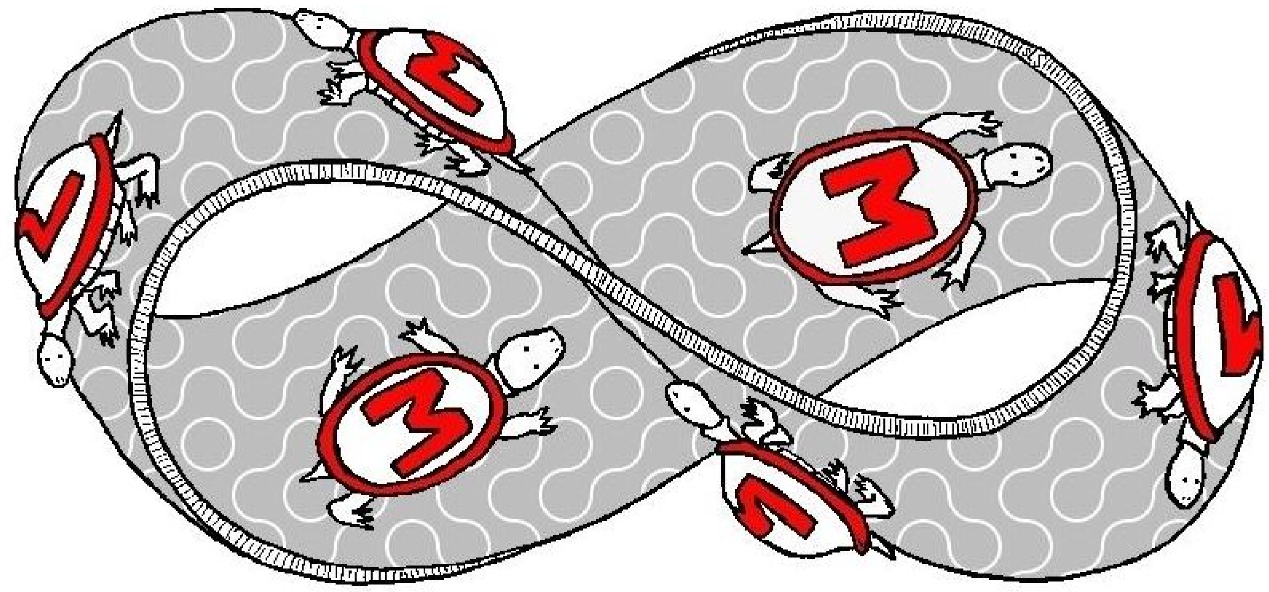
Contents



1 Coin Flip

Hiro and GoGo are trying to solve a puzzle involving n coins laid in a row, showing either heads or tails. Any number of adjacent coins may be flipped over in a single move.

The eventual goal is to find the minimum number of moves that will flip coins so that all coins show heads. For now GoGo plans to start by calculating the result of a sequence of flips, and she would like your help with that.

Each move is specified as a range between 1 and n. For example, given 3 coins showing TTT, the possible flip moves are 1-1, 1-2, 1-3, 2-2, 2-3, and 3-3. Flipping a coin changes H to T and vice versa. Given 4 coins showing HHHH, applying a flip move 1-1 would flip the first coin, yielding THHH.

Coins may be flipped multiple times if a sequence of flips are given. Applying a sequence of flip moves 1-4, 1-4 to 4 coins showing HHHH would first flip all 4 coins, yielding TTTT, then flip all 4 coins again, yielding HHHH.

Input/Output Format:

Input: The first line in the test data file contains the number of test cases. After that each line will contain a string of n coin positions (either H or T), followed by the number of moves. Each move is then given as a pair of numbers representing a range x to y, where $1 \leq x \leq y \leq n$.

Output: For each test case, the program will display on a single line the initial coin position, followed by the coin position after each move, separated by spaces.

Note: We have provided a skeleton program that reads the input and prints the output based on the following `flipCoins()` method. `flipCoins()` is passed the original coin positions and a sequence of flip moves. It should return the different coin positions resulting after applying each flip move in the sequence, as an array of `char[]`.

```
private static char[][] flipCoins(char coins[], int flip[][])
```

Examples:

The output is shown with an extra blank line so that each test case input is aligned with the output; the first blank line should not be present in the actual output. Coins that were flipped by the previous move are underlined for emphasis; the underlines do not appear in the actual output.

Input:	Output:
4	
HHHH 1 1 1	HHHH THHH
HHHH 1 1 4	HHHH TTTT
HHHH 2 1 4 1 4	HHHH TTTT HHHH
TTTTT 3 1 5 2 4 3 3	TTTTT HHHHH HTTTH HTHTH

(This page intentionally left blank)

2 Weakest Microbot

Hiro wants to further analyze the different arrangements of microbots by mapping them to graphs. To recap: Hiro's microbots can link to each other in any arrangement imaginable. The physical properties of the arrangements, however, vary considerably depending on how the microbots are linked together. One way to view the arrangement of the microbots is as a graph over them; here the nodes (vertices) of the graph are the individual microbots and there is an edge between two nodes if the corresponding microbots are linked together. Figure below shows one such graph over 9 microbots. The labels on the nodes indicate IDs of the microbots, that are always of the form $v[number]$ (i.e., $v0, v1, v2, v3, ...$).

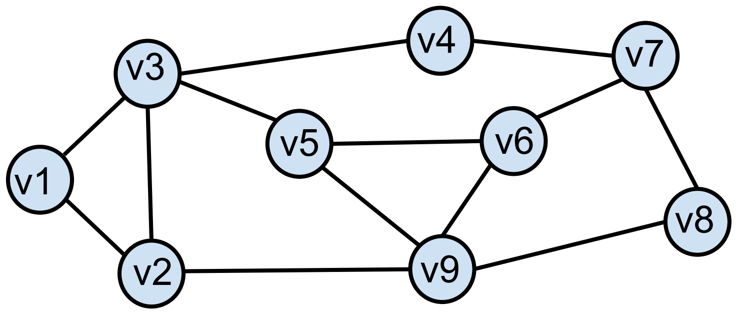
Hiro has determined that the structural integrity of the arrangement depends on how well the neighborhoods of different nodes are connected, and wants your help to identify the microbot whose neighbors are not well connected, i.e., are *weakest*. Specifically, for a node u in the graph, let $N(u)$ denote all the neighbors of u (i.e., all the nodes connected to u in the graph). Let $|N(u)|$ denote the number of such neighbors (i.e., the degree of node u). Let $M(u)$ denote the number of edges among the nodes in $N(u)$. We then define the *density* of a node, *density*(u), to be:

$$\mathit{density}(u) = \frac{2 \times M(u)}{|N(u)| \times (|N(u)| - 1)}$$

As an example, in the graph shown in the figure below:

- $N(v3) = \{v1, v2, v4, v5\}$, $M(v3) = 1$ (there is only one edge among those nodes $(v1, v2)$), and $\mathit{density}(v3) = (2 \times 1)/(4 \times 3) = 0.16666...$
- $N(v6) = \{v5, v7, v9\}, M(v6) = 1$ (edge $(v5, v9)$), and $\mathit{density}(v6) = 1/3 = 0.333333....$
- $N(v1) = \{v2, v3\}, M(v1) = 1$, and $\mathit{density}(v1) = 1/1 = 1.0$.

If a node w has degree 1 (i.e., only 1 neighbor), then $\mathit{density}(w) = 1.0$. Note that, for any node, $\mathit{density}(u) \leq 1.0$, i.e., density can never be more than 1.



Hiro wants your help to find the node that has the weakest neighborhood, i.e., the node with the lowest density, but if two nodes have approximately the same density, he wants you to return the node with the higher degree. More specifically, among two nodes u_1 and u_2:

- If $\mathit{density}(u_2) - \mathit{density}(u_1) > 0.0001$: u_1 is considered weaker than u_2.
- If $abs(\mathit{density}(u_2) - \mathit{density}(u_1)) < 0.0001$ and $|N(u_1)| > |N(u_2)|$: u_1 is considered weaker.

- If $abs(density(u_2) - density(u_1)) < 0.0001$ and $|N(u_1)| = |N(u_2)|$: then both the nodes are considered equally weak and you should return the one that is lexicographically first (using `String.compareTo()`).

Input/Output Format:

Input: The first line in the test data file contains the number of test cases (< 100), followed by the test cases one-by-one. For each test case, the first number represents the number of edges among the microbots, e ($e \leq 10000$). After that, each edge is on a separate line, and is specified by the IDs of the two microbots that it connects (the IDs are of the form: `v0, v1, v2, ...`).

Output: For each test case, you are to output the ID of the microbot that is weakest according to the rules specified above.

The exact format of Input/Output is shown below in the examples.

Note: We have provided a skeleton program that reads the input and prints the output. All you need to do is provide the body of the following procedure:

```
private static String solveWeakestMicrobot(ArrayList<Edge> edges)
```

Here *Edge* is a new class defined for you in the provided skeleton file.

Examples:

The output is shown with extra blank lines so that each test case input is aligned with the output; those blank lines should not be present in the actual output.

Input:	Output:
2	
11	The lexicographically first microbot with the least connected neighborhood is v3
v1 v3	
v2 v3	
v3 v4	
v3 v5	
v4 v7	
v5 v6	
v5 v9	
v6 v7	
v6 v9	
v7 v8	
v8 v9	
8	The lexicographically first microbot with the least connected neighborhood is v1
v1 v2	
v1 v3	
v1 v5	
v2 v4	
v2 v5	
v3 v4	
v3 v5	
v4 v5	

3 Digit Product Sequences

Baymax really likes the concept of Integer Sequences, and as his Math skills are improving, he starts coming up with new sequences of his own to play with. Tadashi has also recently taught him about positional numeral systems and how numbers can be represented in different bases. More specifically, a numeral system uses a *base* to represent/encode numbers. A base-k notation uses k *digits*: 0, 1, 2, ..., $k-1$ to denote a number, and the value of a digit at place l (from the right) is multiplied by k^{l-1}. For example, the number 465 in base k is equal to: $4 \times k^2 + 6 \times k + 5$. Some of the most commonly used bases include 10 (decimal), 2 (binary), 3 (ternary), and 16 (hexadecimal). We will need to use larger bases and to avoid ambiguity, we will use the full expansion as shown above.

Baymax invents a new type of sequence, called Digit Sum Product Sequence (or Digit Product Sequence), that combines the integer sequences and numeral systems. A Digit Product Sequence (DSP) is defined by two numbers: (1) a *start* number, and (2) a base b. Let seq_i denote the i^{th} number in the sequence (so $seq_1 = start$). The next number in the sequence, seq_{i+1} is computed as follows: we first find the base b representation of seq_i, we then multiply all the non-zero digits in that representation, and add the resulting product to seq_i.

Following are some examples of how to compute seq_{i+1} given seq_i and the base. Note that: we use the respective base system only to find the digits in that base – all the calculations are still represented in the standard base 10 notation.

- $\underline{b = 10, seq_i = 11\ (base\ 10) = 1 \times 10^1 + 1}$: then $seq_{i+1} = 11\ (base\ 10) + 1 \times 1 = 12\ (base\ 10)$
- $\underline{b = 2, seq_i = 11\ (base\ 10) = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1}$: then $seq_{i+1} = 11\ (base\ 10) + 1 \times 1 \times 1 = 12\ (base\ 10)$
- $\underline{b = 3, seq_i = 11\ (base\ 10) = 1 \times 3^2 + 0 \times 3^1 + 2}$: then $seq_{i+1} = 11\ (base\ 10) + 1 \times 2 = 13\ (base\ 10)$
- $\underline{b = 3, seq_i = 28\ (base\ 10) = 1 \times 3^3 + 0 \times 3^2 + 0 \times 3^1 + 1}$: then $seq_{i+1} = 28\ (base\ 10) + 1 \times 1 \times 1 = 29\ (base\ 10)$
- $\underline{b = 16, seq_i = 28\ (base\ 10) = 1 \times 16^1 + 12}$: then $seq_{i+1} = 28\ (base\ 10) + 1 \times 12 = 40\ (base\ 10)$
- $\underline{b = 100, seq_i = 28\ (base\ 10) = 28 \times 100^0}$: then $seq_{i+1} = 28\ (base\ 10) + 28 = 56\ (base\ 10)$
- $\underline{b = 100, seq_i = 132\ (base\ 10) = 1 \times 100^1 + 32 \times 100^0}$: then $seq_{i+1} = 132\ (base\ 10) + 32 = 164\ (base\ 10)$

The conversion between bases plus all the arithmetic is getting confusing for Baymax, and even more so for Tadashi, and they would like your help to confirm their answers.

Input/Output Format:

Input: The first line in the test data file contains the number of test cases (< 100). After that, each line contains one test case with two numbers: the first number is the base of the Digit Product sequence ($base \leq 100$), the second number (seq) is one of the numbers of sequence, and is provided in the standard base 10 notation ($seq \leq 10000$).

Output: For each test case, you are to find the next number in the Base *base* Digit Product Sequence after *seq*. The exact format is shown below in the examples.

Note: We have provided a skeleton program that reads the input and prints the output. All you need to do is provide the body of the following procedure:

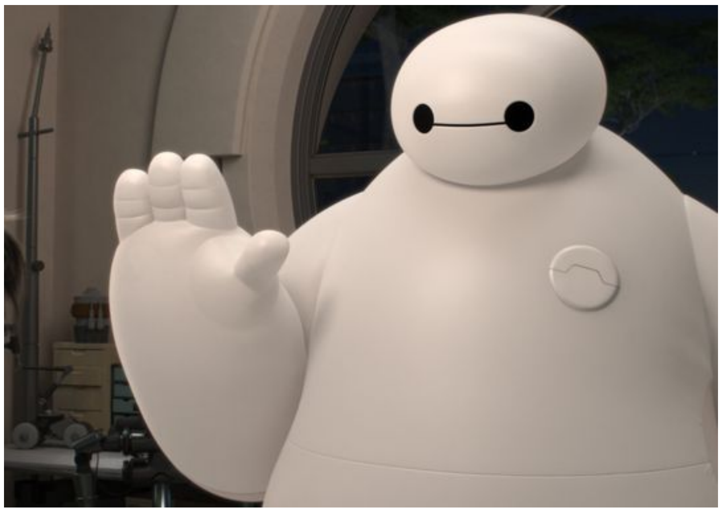
```
private static int solveDSPSequence(int base, int seq)
```

The procedure should return the next number in the sequence as an int. The number should be printed in base 10 representation (the provided output code takes care of this).

Examples:

The output is shown with an extra blank line so that each test case input is aligned with the output; the first blank line should not be present in the actual output.

Input:	Output:
`7`	
`10 11`	`In Base 10 DSP sequence, the number after 11 is 12`
`2 11`	`In Base 2 DSP sequence, the number after 11 is 12`
`3 11`	`In Base 3 DSP sequence, the number after 11 is 13`
`3 28`	`In Base 3 DSP sequence, the number after 28 is 29`
`16 28`	`In Base 16 DSP sequence, the number after 28 is 40`
`100 28`	`In Base 100 DSP sequence, the number after 28 is 56`
`100 132`	`In Base 100 DSP sequence, the number after 132 is 164`



4 SignPost I

Honey Lemon and Fred are fitting Baymax with a new navigation chip to help him get around the university. To test how well the chip works, Honey Lemon creates a difficult navigation course for Baymax. She takes a marker and a bunch of index cards, and writes either "North," "South," "East," or "West" on each card. She then shuffles the cards, and places a single card on each tile of the laboratory floor.

Honey Lemon places Baymax on the tile at the Northwest corner of the room. Baymax must pick up the cards from every tile in the room, starting with the tile in Northwest corner. However, after Baymax picks up a card, he may only walk in the direction written on that card until he decides to stop and pick up the next card.

For example, suppose that when Baymax begins by picking up the first card in the Northwest corner, the card says "East". Baymax then has to walk due East from his current position, and choose another tile to stop and pick up the next card. If it says "South", then Baymax must walk due South and choose another tile. Baymax must continue with this process until he picks up every card from the floor. The challenge here is that: in order to pick up all cards, Baymax must be very careful about choosing which card to pick up next. Figure 1(i) shows one successful traversal where Baymax was able to pick up all the cards – the numbers in the squares indicate the order in which Baymax picked up the cards.

1 E	2 S	16 N	15 W
11 E	12 E	13 E	14 N
9 S	7 E	8 W	6 W
10 N	3 E	4 E	5 N

(i)

1 E	2 S	15 N	14 W
11 E	**16 E**	12 E	13 N
9 S	7 E	8 W	6 W
10 N	3 E	4 E	5 N

(ii)

1 E	2 S	8 N	7 W
14 E	15 E	16 E	6 N
12 S	11 E	10 W	**9 W**
13 N	3 E	4 E	5 N

(iii)

Figure 1: **(i) possible arrangement of cards on the floor, and the order in which the tiles must be visited to successfully pick up all the cards; (ii), (iii) Invalid solutions with the first wrong move highlighted (your program should return 16 and 9 respectively)**

On the other hand, for the same configuration of cards: after picking up the first card and walking due East, if Baymax were to pick up the 3rd card (that says "North"), then he would be stuck; there is no way to go North, and he still has quite a few cards left to pick up. Figure 1 shows two other invalid solutions.

Honey Lemon guarantees Baymax that there is always at least one way to pick up all the cards. But finding that order is a difficult task, and sometimes Baymax cheats when Honey Lemon is not looking. Honey Lemon needs your help in writing a program that examines the path taken by Baymax, and determines whether the path is valid. If the path is valid, your program should return the integer -1. If the path is invalid, your program should output the number of cards held by Baymax after he picks up the first disallowed card. For example, after picking up the first card in the Northwest corner, if

Baymax were to pick up the card immediately below it (which is not allowed since Baymax must go East), then your program should output 2.

Input/Output Format:

Input: The first line in the test data file contains the number of puzzles that Baymax needs to solve. If this number is 3, then there will be 3 different navigation puzzles. The next line contains the number of tiles on each side of the room in the first puzzle (if this number is 4, then the room is 4×4). The floor tiles are then listed in order, with one row of tiles per line. For each tile, an integer is listed, followed by a letter. The integers label the order in which Baymax visited the tiles. The letter determines the cardinal direction printed on that tile's card. Once every square in the first puzzle is listed, the next line of input is the dimension of the second puzzle, followed by the tiles in the second puzzle, followed by the third puzzle, and so on.

The skeleton code for this problem contains the method `readPuzzle()` that will automatically parse each puzzle from the input, and store it in 2-dimensional array of `Square` objects. This array will be handed to your `solveSignPostI` method.

Output: The program should output "The solution is valid" when Baymax found the correct solution. If the solution is invalid, and the Nth tile is the first incorrect move, the program should output "The solution is invalid, and the first incorrect number is N." The skeleton code for this problem will automatically generate these output strings for you.

All you need to do is provide the body of the following procedure:

```
public static int solveSignPostI(Square[][] puzzle)
```

The procedure should return -1 if the ordering of the tiles is valid. It should return the number of the first invalid tile if the path is invalid.

Examples:

The output is shown with extra blank lines so that each test case input is aligned with the output; those blank lines should not be present in the actual output.

Input:	Output:
`3`	
`2`	`The solution is valid`
`01 S 04 N`	
`02 E 03 N`	
`2`	`The solution is invalid, and the first incorrect number is 2`
`01 S 02 N`	
`04 E 03 N`	
`3`	`The solution is valid`
`01 S 06 S 05 W`	
`02 E 07 S 03 S`	
`09 N 08 W 04 N`	

5 Minimum Flips

Hiro and GoGo are now ready to solve the coin flip puzzle. Recall that the puzzle involves n coins laid in a row, showing either heads or tails. Any number of adjacent coins may be flipped over in a single move.

The goal is to find and display all minimum-length flip moves sequences that will flip coins so that all coins show heads. For instance, given the coins TTT, all coins may be changed to Heads using a single flip move of 1-3. The solution is thus the (length 1) sequence TTT HHH.

If there are multiple solution sequences, all minimal length solution sequences should be displayed in alphabetical order. For instance, given the coins THT, there are 6 possible solutions using the minimal number of moves (2). Two solutions are THT HHT HHH and THT HTH HHH, and are displayed in that order since HHT precedes HTH alphabetically.

The number of possible flip move sequences and solutions is quite large even for small numbers of coins. Fortunately, the number of minimal length solutions is usually much smaller. To be able to solve all test cases found within a reasonable time, Hiro and GoGo must design their algorithm to find the minimal length solution quickly, to avoid wasting time considering longer move sequences.

Input/Output Format:

Input: The first line in the test data file contains the number of test cases. After that each line will contain a string of n coin positions (either H or T).

Output: For each test case, the program will first display on a single line the word "Coins" followed by the initial coin position. Minimal flip solutions producing all Heads are listed next, with each solution on a separate line. A solution is a list of coin positions, starting with the initial position and showing the coin position after each flip, separated by spaces.

If multiple solutions are possible, they must be listed in alphabetical order, treating the entire sequence as a single string. No solutions should be displayed if the coins are initially all Heads.

Note: We have provided a skeleton program that reads the input and prints the output based on a findMinFlip() method that you need to implement. The method takes as an argument the string representing starting coin positions, and returns an ArrayList of solution sequences represented as Strings. The solutions may be returned in any order, since the ArrayList will be sorted alphabetically before it is output.

The solution string does not need to include the initial coin position, since it will be automatically added to the beginning of the solution output. So calling findMinFlip("TTT") will return ["HHH"] instead of ["TTT HHH"].

Note a String s may be converted to a char array c[] using the String method toCharArray() via "c = s.toCharArray()". A char array c[] may be converted to a String s using the String constructor via "s = new String(c)".

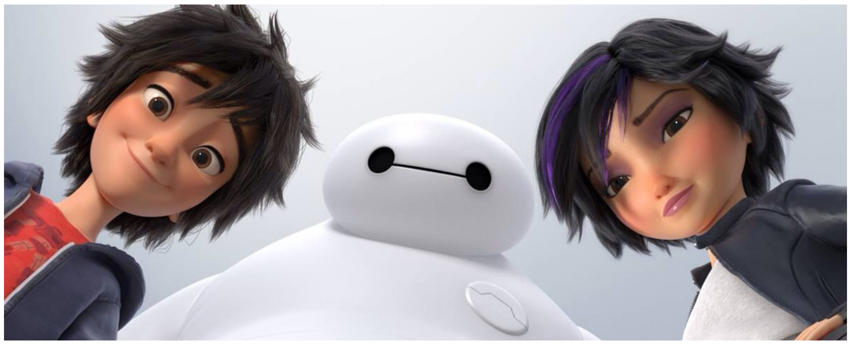
```
    private static ArrayList<String> findMinFlip(String board)
```

Examples:

In the example below, the first test case required only a single flip (of all coins). The second test case has six possible minimal length (2-move) solutions, which are listed in alphabetical order. The third test case did not require any flip solutions because it is already all Heads.

Coins that were flipped by the previous move are underlined for emphasis; the underlines do not appear in the actual output.

`Input:`	`Output:`
`3`	`Coins TTT`
`TTT`	`TTT HHH`
`THT`	`Coins THT`
`HHH`	`THT HHT HHH`
	`THT HTH HHH`
	`THT HTT HHH`
	`THT THH HHH`
	`THT TTH HHH`
	`THT TTT HHH`
	`Coins HHH`



6 The Can't-Move Game

GoGo and Wasabi are looking for a way to pass the time in their lab one Saturday afternoon. They come up with an idea for a game. They find a pile of small stones in the lab (actually, a bowl of stale M&M's left over from the Halloween party). GoGo remembers the ancient game of Nim, where two players take turns removing either one or two stones from the pile until there are no stones remaining. A player loses if he/she can't move, because no stones remain.

Wasabi says that he knows how to win at Nim every time, and so they add a twist. Each player reaches into their pocket and pulls out some number of dollars. Whenever a player takes a stone, they must pay for the stone with an equal number of dollars—one stone costs \$1 and two stones costs \$2. Now, a player loses when no move is possible, either (1) because there are no stones left in the pile or (2) this player does not have any money left.

Baymax overhears them planning this game, and (using his robotic brain) remarks that he can predict the winner (assuming that both players play as well as is possible). Write a function that solves this problem. Your function will be given the number of stones n, the number of dollars t_1 that GoGo has and the number of dollars t_2 that Wasabi has. GoGo always has the first move.

For example, suppose that there are $n = 4$ stones, and $t_1 = \$2$ and $t_2 = \$3$. In standard Nim (without money), the first player can force a win. Let's call the players P_1 and P_2. P_1 takes 1 stone, leaving 3 stones in the pile. It doesn't matter what P_2 does. If P_2 takes 1 stone, then P_1 takes the remaining 2 stones and wins. If P_2 takes 2 stones, P_1 takes the remaining 1 stone and wins. (Recall that a player must choose to pick up either 1 or 2 stones in each move).

In the new version (with $t_1 = \$2$ and $t_2 = \$3$), however, P_2 can force a win. If P_1 starts by taking 1 stone, then P_2 responds by taking 1 stone. They both have at least 1\$ remaining. P_1's only option is to take 1 stone, and P_2 takes the last stone and wins. On the other hand, if P_1 starts by taking 2 stones, then P_2 takes the remaining 2 stones and wins.

Input Format

The first line of the input file contains the number of trials. Each trial involves a single line containing the number of stones n, the number of dollars t_1 for player P_1, and the number of dollars t_2 for player P_2. We will provide a procedure for reading the input. You provide the body of a function that returns either 1 or 2, indicating which player can force the win (assuming both play as well as possible).

```
  private static int getWinner(int nStones, int nTake1, int nTake2)
```

where nStones is n, nTake2 is t_1, and nTake2 is t_2. You may assume that the input is valid and that $1 \leq n \leq 200$, $1 \leq t_1 \leq n$, and $1 \leq t_2 \leq n$.

Output Format

You do not need to generate any output. We will provide a main program that will read the input, print a summary of the input, and print the result of your function.

Examples:

Input:

```
2
4 2 3
4 4 4
```

Output: (Generated by our program)

```
Trial 1:
  4 stones
  up to 2 stones may be taken by P1 in total
  up to 3 stones may be taken by P2 in total

Player P2 wins

Trial 2:
  4 stones
  up to 4 stones may be taken by P1 in total
  up to 4 stones may be taken by P2 in total

Player P1 wins
```

7 SignPost II

In “Signpost I” you designed a program to check whether Baymax picks up Honey Lemon’s index cards correctly. Now that Honey Lemon has this program, Baymax can’t get away with cheating anymore. Help Baymax pick up the index cards correctly by writing a program that accepts a layout of index cards and labels the tiles in the order in which they must be visited. Recall that Baymax always starts in the Northwest corner of the room. For this reason, the top-left square (with indices 0,0) in your output must always store the number “1.” Number every other square in your output to indicate the order in which each tile must be visited.

There may be multiple valid solutions for a puzzle; any valid solution will be accepted.

Input/Output Format:

Input: The input format matches that of “Signpost I.” The first line in the test data file contains the number of puzzles that Baymax needs to solve. If this number is 3, then there will be 3 different navigation puzzles. Each puzzle is then listed, one at a time. The next line contains the number of tiles on each side of the room (if this number is 4, then the room is 4×4). The floor tiles are then listed in order, with one row of tiles per line. For each tile, an integer is listed, followed by a letter. Because Baymax hasn’t solved these problems yet, the integer in each square is “0.” The skeleton code for this problem contains the method `readPuzzle` that will automatically read each puzzle, and store it in a 2-dimensional array of `Square` objects. This array will be handed to your `solveSignPostII` method.

Output: The output should be formatted the same way as the input. Every line of the output should match every line of the input, except that the integers in each tile should be changed to indicate the order in which the tiles are visited. The skeleton code for this problem will automatically generate these output strings for you.

All you need to do is provide the body of the following procedure:

```
public static Square[][] solveSignPostII(Square[][] puzzle)
```

The procedure accepts an array of `Square` objects. Each objects contains a letter (“N,” “S,” “E,” or “W”) indicating which direction Baymax can move from that square. The number stored in each square of the input is “0.” The method should output an array of `Square` objects with the same dimensions as the input. The directions stored in the output `Square`’s must match the inputs. However, the output `Square`’s should contain numbers labeling the order in which the squares must be visited to complete the puzzle. For an $N \times N$ puzzle, the squares should contain the numbers 1 through N^2, and each number in this range should be used exactly once in your solution.

(Examples on the next page)

Examples:

Input:	Output:
3	3
2	2
0 E 0 S	01 E 02 S
0 N 0 W	04 N 03 W
2	2
0 S 0 N	01 S 04 N
0 E 0 N	02 E 03 N
3	3
0 E 0 S 0 S	01 E 02 S 05 S
0 S 0 W 0 W	08 S 07 W 06 W
0 N 0 E 0 N	09 N 03 E 04 N

8 Guessing on an Exam

All the students in Professor Callaghan's robotics lab are about to do a take-home exam. Professor Callaghan wants to discourage students from guessing, so he has a policy of deducting points when a student guesses incorrectly. Before the exam, the professor tells the students the scoring rules. There are two numbers, C and W, both positive integers.

- If a question is answered correctly, C points are added.
- If a question is answered incorrectly, W points are subtracted.
- If a question is skipped (no answer given), no points are added or subtracted.
- Since he doesn't want to give negative scores, if the total score is negative by the above rules, then the final score is 0.

For example, suppose that $C = 5$ and $W = 10$, and there are 20 questions on the exam. If the student answers 18 problems correctly and 2 incorrectly, then the final score is $18 \cdot C - 2 \cdot W = 90 - 20 = 70$. On the other hand, if 12 are correct and 8 are incorrect, the total score would be $12 \cdot C - 8 \cdot W = 60 - 80 = -20$. Because this is negative, the final score is 0.

Fred is the least studious member of the class, and he is worried. Is it worth guessing? Hiro offers to help. For each of the questions, Fred tells Hiro his probability of providing a correct answer. Each probability is chosen from the set $\{0.25, 0.5, 0.75, 1\}$. Hiro writes a program that will tell Fred which questions he should attempt to give an answer and which he should skip in order to achieve the maximum expected score.

Computing Expected Scores

To get some feeling for what this involves, let's consider an example. Suppose that there are just two questions on the exam, and Fred has a $\frac{1}{4}$ probability of getting the first question correct and a $\frac{1}{2}$ probability of getting the second question correct. Also, suppose that $C = 5$ and $W = 10$. You might be tempted to compute the expected number of points on each problem and then just answer those for which the expected score is positive. If Fred attempts the first question, his expected score is $\frac{1}{4}C - \frac{3}{4}W = -6.25$. If he attempts the second question, his expected score is $\frac{1}{2}C - \frac{1}{2}W = 2.5 - 5 = -2.5$. Since both of these are negative, it might seem that a score of 0 is the best he can hope for.

But the professor's rule about never giving negative final scores comes to Fred's rescue. To see why, suppose that Fred skips Question 1 and attempts Question 2. He gets no points for Question 1, so his total is based on Question 2 alone. With probability $\frac{1}{2}$, he gets 5 points. With probability $1 - \frac{1}{2} = \frac{1}{2}$, he gets -10 total points, but by the professor's rule, the final score in this case would be 0. Therefore, the expected final score is $\frac{1}{2}5 + \frac{1}{2}0 = 2.5$, which is positive. To extend this to attempting two problems, you should consider all the possible correct/incorrect outcomes (there are four of them), the probability of each outcome, and its final score. (Note that a naive exhaustive enumeration like that would not work to actually solve the problem instances below because the number of problems is too high).

Input and Output

The first line of the input file contains the number of trials. Each trial starts with a single line containing the number of questions n and integers C and W. You may assume that $1 \leq n \leq 200$, and both C and

W are strictly positive. This is followed by the n probability values. Each probability $p[i]$ is of type double, and $p[i] \in \{0.25, 0.5, 0.75, 1\}$.

You need only provide the body of a function that (1) determines which problems Fred should attempt to maximize his expected score and (2) returns this maximum expected score.

```
static double whichToAttempt(int n, int C, int W, double[] p, boolean[] attempt)
```

Here n is the number of questions, C and W are as described above, $p[n]$ is an n-element array containing the probabilities. The questions to be attempted are returned through the boolean array attempt$[n]$ (which is allocated for you). To indicate that Fred should attempt problem i, set attempt$[i]$ = true. The return value is the expected final score, assuming that these problems are attempted.

Notes 1: The number of questions on the exam can be fairly large (up to 200, say), and hence brute-force search will not be feasible.

Notes 2: There may be many solutions with the same maximum expected score (e.g., in the second example below, skipping Problem 0 and attempting Problem 4 would leave Fred with the same expected score). It is fine to return any solution that achieves the best expected score.

Sample Input/Output

Input:

```
2
2  5 10 0.25 0.50
7  10 16 0.50 0.25 0.50 0.50 0.50 0.75 0.50
```

Output: (Generated by our program)

```
Trial 1:
  2 questions
  5 points for correct answers
  10 point deduction for incorrect answers

  Maximum expected score = 2.5
  Prob[0] = 0.25  ---  skip
  Prob[1] = 0.5  ---  attempt

Trial 2:
  7 questions
  10 points for correct answers
  16 point deduction for incorrect answers

  Maximum expected score = 8.125
  Prob[0] = 0.5  ---  attempt
  Prob[1] = 0.25  ---  skip
  Prob[2] = 0.5  ---  attempt
  Prob[3] = 0.5  ---  attempt
  Prob[4] = 0.5  ---  skip
  Prob[5] = 0.75  ---  attempt
  Prob[6] = 0.5  ---  skip
```

Practice 1 – Fibonacci Sequence

Tadashi is trying to teach Baymax about integers and their interesting properties. As a way of increasing Baymax's Math skills, Tadashi introduces him to the concept of *integer sequences*, and starts with the perhaps the most famous of them, the Fibonacci sequence. The Fibonacci sequence is a series of numbers where the next number is obtained by summing the previous two numbers in the series, with the first two numbers being 0 and 1.

$\underline{0}$, $\underline{1}$, $0 + 1 = \underline{1}$, $1 + 1 = \underline{2}$, $1 + 2 = \underline{3}$, $2 + 3 = \underline{5}$, $3 + 5 = \underline{8}$, $5 + 8 = \underline{13}$, $8 + 13 = \underline{21}$, ...

Using different two numbers (denoted $start1$ and $start2$) to start with gets us different sequences, sometimes called Generalized Fibonacci sequences. We will denote such a sequence by Fibonacci($start1$, $start2$), so the original sequence above would be denoted Fibonacci(0, 1). The following are some examples:

- Fibonacci(2, 4): $start1 = 2$, $start2 = 4$, 6, 10, 16, 26, ...
- Fibonacci(2, 5): $start1 = 2$, $start2 = 5$, 7, 12, 19, 31, ...

Baymax would like your help with confirming that he is able to compute the numbers in a Generalized Fibonacci sequence correctly.

Input/Output Format:

Input: The first line in the test data file contains the number of test cases (≤ 100). After that, each line contains one test case: the first two numbers are the start of the Fibonacci sequence ($start1$ and $start2$), and the third number is a "position", pos (all provided as `int`s).

Output: For each test case, you are to find the pos^{th} number in the Fibonacci sequence starting with $start1$, $start2$. The numbers at the first few positions are: position 1 = $start1$, position 2 = $start2$, position 3 = $start1 + start2$. The exact format is shown below in the examples.

Note: We have provided a skeleton program that reads the input and prints the output. All you need to do is provide the body of the following procedure:

```
private static int solveFibonacci(int start1, int start2, int pos)
```

The procedure should return the number at position pos.

(Examples on the next page)

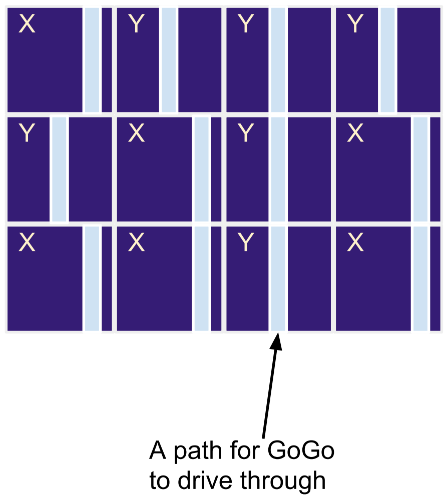
Examples:

The output is shown with an extra blank line so that each test case input is aligned with the output; the first blank line should not be present in the actual output.

Input:	Output:
`8`	
`1 2 1`	`In the Fibonacci(1, 2) sequence, the number at position 1 is:  1`
`1 2 2`	`In the Fibonacci(1, 2) sequence, the number at position 2 is:  2`
`1 2 3`	`In the Fibonacci(1, 2) sequence, the number at position 3 is:  3`
`1 2 4`	`In the Fibonacci(1, 2) sequence, the number at position 4 is:  5`
`1 2 5`	`In the Fibonacci(1, 2) sequence, the number at position 5 is:  8`
`1 2 6`	`In the Fibonacci(1, 2) sequence, the number at position 6 is:  13`
`1 2 30`	`In the Fibonacci(1, 2) sequence, the number at position 30 is:  1346269`
`20 30 10`	`In the Fibonacci(20, 30) sequence, the number at position 10 is:  1440`

Practice 2 – Motorcycle Paths

GoGo is chasing the man in the Kabuki mask on her motorcycle through the streets of San Fransokyo. They come to an open field with many big square-shaped slabs packed in a grid (see figure below). The slabs are too tall for GoGo to jump on top of them. However, all the slabs have grooves in them that she can drive through. The slabs are of two types, X and Y, with slightly different placement of grooves. If two slabs are both of type X or type Y, then their grooves are aligned and there is path through both of them. In order to cross the field and continue chasing after the man in the Kabuki mask (who has jumped over the slabs using his microbots), GoGo needs to find an entire column of slabs that are of the same type so she can drive through them. You are to help GoGo with finding such a path, if it exists.



Input/Output Format:

Input: The first line in the test data file contains the number of test cases ($\leq$ 100), followed by the test cases listed one by one. For each test case, the first line contains the number of rows, *num* (*num* < 50), of slabs. The next *num* lines contain *num* `String`s, each containing only letters “X” and “Y”, indicating the types of slabs in each row. The strings are all of the same length.

Output: For each test case, you are to output the first column that is all X’s or all Y’s, or report that there is no such column. The exact format is shown below in the examples.

Note: We have provided a skeleton program that reads the input and prints the output. All you need to do is provide the body of the following procedure:

```
private static int solveUniformColumn(String[] puzzle)
```

The function should return the first column that contains all X’s or all Y’s, or 0 if there is no such column.

(Examples on the next page)

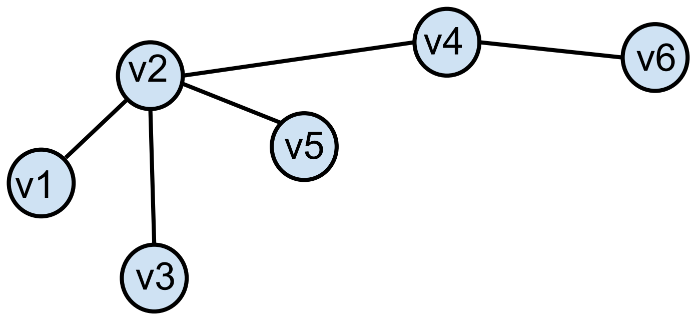
Examples:

The output is shown with extra blank lines so that each test case input is aligned with the output; those blank lines should not be present in the actual output.

Input:	Output:
3	
3	Column 1 is the first column with all X's or all Y's.
XXX	
XYX	
XYY	
3	There is no column with all X's or all Y's.
XXX	
YYX	
XYY	
3	Column 2 is the first column with all X's or all Y's.
XYX	
YYX	
XYY	

Practice 3 – Microbot Graphs

Hiro's microbots can link to each other in any arrangement imaginable. The physical properties of the arrangements, however, vary considerably depending on how the microbots are linked together. One way to view the arrangement of the microbots is as a graph over them; here the nodes (vertices) of the graph are the individual microbots and there is an edge between two nodes if the corresponding microbots are linked together. Figure below shows one such graph over 6 microbots. The labels on the nodes indicate IDs of the microbots, that are always of the form $v[number]$ (i.e., $v1, v2, v3, ...$).



Hiro would like your help to analyze the graph structure, starting with identifying the microbot that is connected with most other microbots. In the example above, $v2$ is connected to most other microbots (4), and thus should be returned as the answer.

Input/Output Format:

Input: The first line in the test data file contains the number of test cases (≤ 100), followed by the test cases one-by-one. For each test case, the first number represents the number of edges among the microbots, e ($e < 1000$). After that, each edge is on a separate line, and is specified by the IDs of the two microbots that it connects (the IDs are of the form: `v1, v2, ...`).

Output: For each test case, you are to output the ID of the microbot that is connected to most other microbots. If there is more than one microbot that satisfies the condition, you should return the ID that appears earlier in the lexicographical order. Use `String.compareTo()` to check the lexicographical order (i.e., $s1$ appears earlier than $s2$ in lexicographical order if `s1.compareTo(s2)` < 0).

The exact format of Input/Output is shown below in the examples.

Note: We have provided a skeleton program that reads the input and prints the output. All you need to do is provide the body of the following procedure:

```
private static String solveMostNeighbors(ArrayList<Edge> edges)
```

Here *Edge* is a new class defined for you in the provided skeleton file.

(Examples on the next page)

Examples:

The output is shown with extra blank lines so that each test case input is aligned with the output; those blank lines should not be present in the actual output.

Input:	Output:
`3`	
`4`	`The lexicographically first microbot with largest number of neighbors is v2`
`v1 v2`	
`v2 v3`	
`v3 v4`	
`v4 v5`	
`11`	`The lexicographically first microbot with largest number of neighbors is v3`
`v1 v3`	
`v2 v3`	
`v3 v4`	
`v3 v5`	
`v4 v7`	
`v5 v6`	
`v5 v9`	
`v6 v7`	
`v6 v9`	
`v7 v8`	
`v8 v9`	
`8`	`The lexicographically first microbot with largest number of neighbors is v2`
`v1 v2`	
`v1 v3`	
`v1 v4`	
`v1 v5`	
`v2 v3`	
`v2 v4`	
`v2 v5`	
`v2 v6`	