Contents


EARTH'S MIGHTIEST HEROES
CSCODERS

1 Error Detection

In the midst of a fierce battle, Tony Stark's suit constantly communicates with JARVIS for technical data. This data as transmitted takes the form of 16-bit integer values. However, due to various atmospheric issues (such as those created by all of that lightning Thor spreads around) there is some risk of data corruption. To help detect such corruption, for each 16-bit value that is transmitted, an additional single bit is sent. This additional single bit, called the *check bit*, is a 1 if the corresponding 16-bit integer has an ODD number of 1s when represented in binary. The check bit is a 0 if the corresponding 16-bit integer has an EVEN number of 1s when represented in binary. The effect is that: the number of bits set to 1 in the combined 17 bits is always EVEN.

For example, the integer 45 would appear in binary as 0000000000101101 which has an even number of 1s so the check bit would be 0. The integer 34173 would appear in binary as 1000010101111101 which has an odd number of 1s so the check bit would be 1.

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 number is the 16-bit integer (provided as an `int`), and the following number is the check bit (also provided as an `int`).

Output: For each test case, you are to output "Corrupt" if the check bit doesn't match up with the even or oddness of the integer, or "Valid" if it does.

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 boolean solveErrorDetection(int value, int checkbit)
```

The procedure should return "false" if the check bit doesn't match up, and "true" otherwise.

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:
4	
34173 1	Valid
45 1	Corrupt
15 0	Valid
31 0	Corrupt

2 Pythogorean Triples

In pursuit of one of the Infinity Gems, rumored to be in possession of Loki, Thor and the Avengers have reached what they believe to be one of Loki's dens. However, their attempt to infiltrate is thwarted by a locked door. The Avengers need to quickly figure out which number (*key*) to enter on the keypad to gain access; however, they don't the time to try the trillions of possible combinations. Thor notices a sequence of numbers on a nearby wall. Thor knows Loki well (they are half-brothers after all), and believes that the key is a *Pythogorean triple*, with the numbers drawn from the sequence.

A triple $\{x, y, z\}, x < y < z$ is called a Pythogorean triple if: $x^2 + y^2 = z^2$. So, for example, {3,4,5} is a Pythogorean triple, while {1,2,3} is not.

Your goal is to write a computer program that helps the Avengers. More specifically, given a sequence of positive integers $x_1, \cdots, x_n$, all different from each other, your goal is find all the Pythogorean triples in that sequence.

For example, given a sequence, $13, 12, 3, 4, 5$, there are two Pythogorean triples in it: $\{3, 4, 5\}$ and $\{5, 12, 13\}$.

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 number is n ($3 \leq n \leq 50$), the size of the sequence, and the following n numbers are the sequence (not necessarily in any order). None of the individual numbers are greater than 2^{24}.

Output: For each test case, you are to output the different Pythogorean triples. The exact form of the output is shown below.

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 void solveTriples(int[] sequence)
```

Further, there is an additional function provided:

```
private static void addTripleToAnswer(int i, int j, int k)
```

This function should be called for every triple that you find. The triples do not need to be added in any specific order; however, when calling the above function, you need to make sure that: $i < j < k$.

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:
4	
6 6 5 4 3 2 1	Found Pythogorean triples: {3 4 5}
3 3000000 5000000 4000000	Found Pythogorean triples: {3000000 4000000 5000000}
5 13 12 3 4 5	Found Pythogorean triples: {3 4 5} {5 12 13}
6 1 2 4 8 16 32	No Pythogorean triples found in the sequence.

3 Vigenère Cipher

Black Widow and Hawkeye need to discuss the plans for a surprise Birthday party for Nick Fury, and decide to use an encryption technique so that Nick can't read their messages. In particular, they decide to use the *Vigenère Cipher*. Vigenère cipher is a simple form of polyalphabetic substitution, and was described by Giovan Battista Bellaso in 1553. It was misattributed to Blaise de Vigenère in the 19th century, because of a similar but stronger cipher presented by Vigenère in 1586.

Vigenère cipher is similar to the Caesar cipher. In Caesar cipher, each letter of the alphabet is shifted along some number of places; for example, in a Caesar cipher of shift 3, "A" would become "D", "B" would become "E", "Y" would become "B" and so on. The Vigenère cipher consists of several Caesar ciphers in sequence with different shift values.

For example, suppose that the plaintext to be encrypted is:

```
ATTACKATDAWN
```

The person sending the message chooses a *keyword* and repeats it until it matches the length of the plaintext, for example, the keyword "LEMON":

```
LEMONLEMONLE
```

Each letter in the plaintext is shifted according to the corresponding letter in the keyword. So the first letter "A" is shifted according to "L", second letter "T" is shifted according to "E", the third letter "T" is shifted according to "M", and so on.

The shifting itself works as follows. If we are shifting according to, say, "L", then: "A" becomes "L", "B" becomes "M", "C" becomes "N", and so on. If we reach the end of the alphabet, then we wrap around to the beginning. So "O" becomes "Z", and "P" becomes "A".

If the keyword letter is "E", then each letter is shifted by 4 (so "T" becomes "X" and so on).

The plaintext above is encrypted to ciphertext:

```
LXFOPVEFRNHR
```

You are to write a program that helps Black Widow and Hawkeye encrypt their messages.

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 strings: the first string is the *keyword*, and the second string is the *plaintext*. Both the keyword and plaintext only contain capital letters (from A to Z) – all numbers or punctuation marks (including white spaces) are stripped out.

Output: For each test case, you are to output the encrypted ciphertext. The exact form of the output is shown below.

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 solveVigenereI(String key, String plaintext)
```

The procedure should return the ciphertext as a string.

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:
4	
LEMON ATTACKATDAWN	Ciphertext: LXFOPVEFRNHR
ABCD CRYPTOISSHORTFORCRYPTOGRAPHY	Ciphertext: CSASTPKVSIQUTGQUCSASTPIUAQJB
ABCDE CRYPTOISSHORTFORCRYPTOGRAPHY	Ciphertext: CSASXOJUVLOSVISRDTBTTPIUEPIA
LUCKY COMPUTINGGIVESINSIGHT	Ciphertext: NIOZSECPQETPGCGYMKQFE

4 Shut The Box I

Their pursuit of the Collector has taken the Avengers on a long voyage through the galaxies, and they are looking for ways to pass time on their ship. They discover an old game, called *shut the box*, that used to be popular with sailors. Each player starts with 9 cards, numbered 1 to 9, laid face-up (i.e., the number showing) on the table, and continues playing till he or she cannot make any moves.

The game is played with a pair of dice (with a small twist that, only one die is used if the total of the open cards is ≤ 6). Initially, all the cards are open (i.e., face-up). The player then roles the dice. Let the total dice value be m. The player selects any set of open cards whose face values sum to m, and closes (shuts) them by turning them face-down. For instance, say the player rolls a 6 and 2 the first time, with a total of 8. The player can shut either the card 8, or the cards 1 and 7, or the cards 2 and 6, or the three cards 1 and 2 and 5, and so on. As another example, if the current cards face up are: 1, 2, 6, and the player rolls a 4, then there are no cards that he can shut, and his turn is over.

The final score for the player is the sum of the numbers of the cards still face-up, and the aim is to get as low score as possible (and ideally to shut the box, i.e., have no cards left facing up). In the above case where the game ends with face up cards 1, 2, and 6, the final score is: $1 + 2 + 6 = 9$.

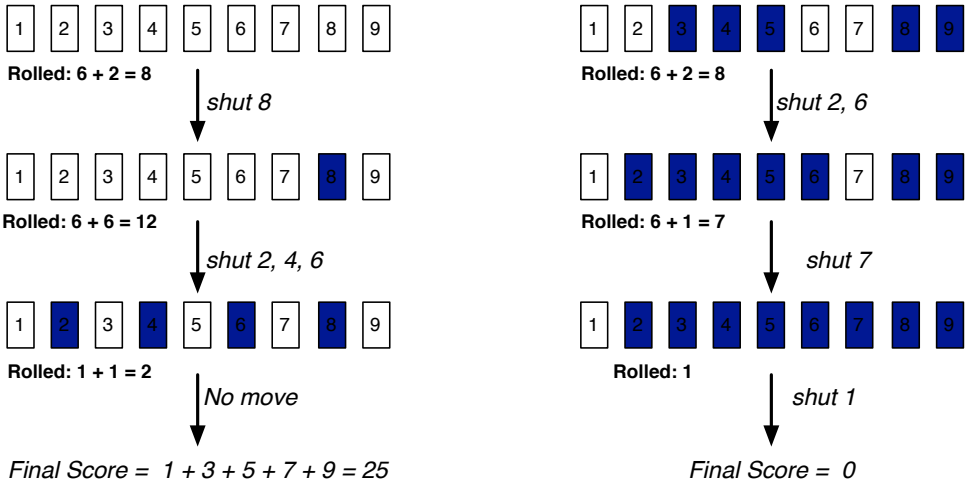

Figure 1: Illustration of Shut the Box game in two cases (in the right example, only the last few moves are shown). Note that, we only use 1 die when the total of open cards is less than or equal to 6 (in the right example)

Dr. Banner (The Hulk) has heard of the following strategy. Consider all the valid alternatives for shutting cards, given the current roll of the dice. If there is only one, do it. If there are more than one, take the one that maximizes the smallest face value. (For example, given the options {1,7} and {2,6}, you would take {2,6}.) If there are many that maximize the smallest face value, take the one that maximizes the second smallest face value. (For example, given the options {2,4,6} and {2,3,7}, you would take {2,4,6}.) In general, among the valid options, select the one that maximizes the lexicographically ordered sequence of face values.

Your goal is to write a program to help Dr. Banner make this choice and, hopefully, win (you know the consequences of angering the Hulk).

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 entry on each line is the rolled total (called *target*), and the next entry is the number of open cards n. The following n entries denote the open cards in the ascending order.

Output: For each test case, you are to output the best move according to the above strategy, or output that there is no legal move. The exact form is shown below.

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 void solveShutTheBoxI(int[] values, int target)
```

Your code should set the variables `solution` and `solution_found` appropriately.

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:
5	
10 9 1 2 3 4 5 6 7 8 9	The best move is: 4 6
12 9 1 2 3 4 5 6 7 8 9	The best move is: 5 7
7 6 1 2 3 4 8 9	The best move is: 3 4
8 7 1 2 3 4 5 6 7	The best move is: 3 5
8 4 1 2 3 9	No move found.

5 Battleground Preservation

While the people of Earth are grateful to Iron Man, Black Widow, the Hulk, and all the rest of the superheroes, the damage that their battles wreak on the cities they are protecting is severe. In the nearly 100 years of these battles, there have been many victories and losses in individual fights. When a new villain appears, it is conceded that a new battle must be fought. However, it has been decided that so many battles being re-fought is not safe or economical. For this reason, a task force has been formed to explore the use of a database of battles to predict who will win an upcoming battle, and then declare a winner before the battle takes place (with the inevitable citywide destruction that would follow) if at all possible.

The rules they have come up with are as follows:

1. Each battle in our database says who was the winner, who was the loser, and the cost of victory as an integer.
2. We will use the database of battles to infer outcomes of other battles.
   - If `X` has defeated `B` and `B` has defeated `Y` then we will treat that as if `X` defeated `Y` and set the cost of battle as the sum of the `X` -vs- `B` and the `B` -vs- `Y` battles.
   - We will allow chains like this to grow very long, so we could have `X` defeat `A` and `A` defeat `B` and `B` defeat `C` and `C` defeat `Y` and treat that as if `X` defeated `Y` (with the sum of the individual battle costs as the cost of victory).
   - If there are two chains between `X` and `Y`, we will use the one with the lower total cost of victory.

   From now on, when we say `X` defeated (or beat) `Y`, it could either be a direct battle between the two or it could be through a chain of battles as described above.
3. If `X` has defeated `Y` and `Y` has not defeated `X`, we will declare that `X` will defeat `Y` again.
4. If `X` beat `Y` but `Y` also beat `X` we will decide the winner based on the cost of victory, where the person who won with the lower cost of victory will be declared the winner in this new battle.

   For example, say (a) `Loki` beat `Thor` with a cost of 27, and (b) `Thor` beat `Loki` with a cost of 18. If `Thor` and `Loki` were to say they wanted to do battle again, we would declare `Thor` the winner of that new battle.
5. If there have never been any battles or battle chains between `X` and `Y` then they will have to actually do battle.
6. If the lowest victory costs for `X` beating `Y` and `Y` beating `X` are identical, then they will have to actually do battle (see 3rd test case below).

Input/Output Format:

Input: The first line in the test data file contains the number of test cases (< 100). Each test case is specified on two lines: the first line starts with the number of combatants (< 100), the number of previous battles (< 1000), and then information about the previous battles as triples: "Combatant1 Combatant2 CostOfVictory"; the second line has the new battle's combatants as: "Combatant1 Combatant2".

Output: For each test case, you are to output the name of the winner of the new battle, or the word "FIGHT!" if the two will need to actually do battle. The exact form is shown below.

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 solveBattle(int numCombatants, ArrayList<SingleBattle> pastBattles,
                               String combatant1, String combatant2)
```

The procedure should return 1 if `combatant1` wins, 2 if `combatant2` wins, and 0 otherwise (i.e., if they should fight).

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:
4	
3 2 Thor Loki 18 Loki Catwoman 12	
Thor Catwoman	Thor
4 2 Thor Loki 18 Loki Catwoman 12	
Thor Firefly	FIGHT!
3 3 Thor Loki 1 Loki Catwoman 1 Catwoman Thor 2	
Thor Catwoman	FIGHT!
3 3 Thor Loki 1 Loki Catwoman 1 Catwoman Thor 1	
Thor Catwoman	Catwoman

6 Breaking Vigenère Cipher

Nick Fury has gotten wind of the secret communications between Black Widow and Hawkeye (who are using a Vigenère Cipher), and is worried about a possible rebellion. He started reading the description of the Vigenère cipher on Wikipedia, and was discouraged to find in the first paragraph that it is considered *le chiffre inchiffrable* ('the indecipherable cipher'). He now wants your help to break it, in return promising not to tell The Hulk about how you ate his lunch.

You are however smarter than Nick, and continue reading the description of Vigenère cipher, and discover that it is considered quite insecure and easy to break. In fact, Charles Babbage, considered a "father of the computer", decrypted a sample of encrypted ciphertext in 1846, using a technique later published by Kasiski, and called "Kasiski examination".

Without getting into too many specific details, Kasiski examination relies on guessing the *length* of the keyword by looking for repeated groups of letters in the ciphertext. For example, consider the following plaintext, keyword (repeated sufficiently), and ciphertext.

```
Keyword:     ABCDABCDABCDABCDABCDABCDABCD
Plaintext:   CRYPTOISSHORTFORCRYPTOGRAPHY
Ciphertext:  CSASTPKVSIQUTGQUCSASTPIUAQJB
```

The plaintext contains a repeated text `CRYPTO`, and because of the fortuitous (for you) alignment, both the occurrences were encrypted to the same ciphertext `CSASTP`. We will denote such a pair of repeated text in the ciphertext by a triple: $\langle text, position1, position2 \rangle$. The above repetition is denoted by: $\langle$`CSASTP`$, 1, 17\rangle$.

For each such triple, we can compute the distance between the two occurences (in this case 17 - 1 = 16) and guess the keyword to be of a factor of that distance (in this case, the possibilities are: 1, 2, 4, 8, or 16). Since 1 and 2 are too small, best guesses are 4, 8, or 16.

The repeated group of letters may also be a coincidence, so a keyword length-guessing algorithm needs to be aware of that possibility.

Your goal is to try to find possible key lengths given a ciphertext. For that purpose, you have to find every repeated group of letters of length exactly 3 in the ciphertext. Say you find n such pairs of repeated groups:

$$\langle text_1, pos1_1, pos2_1 \rangle, \langle text_2, pos1_2, pos2_2 \rangle, \cdots, \langle text_n, pos1_n, pos2_n \rangle$$

Let $x_1 = pos2_1 - pos1_1, \cdots, x_n = pos2_n - pos1_n$ be the distances between the repeated occurences. The possible guesses are found as follows: a number k is a guess for the length of the keyword if and only if it is a factor of (i.e., it divides) at least 90% of the numbers in $\{x_1, \cdots, x_n\}$. Further, we will assume that the key length is between 4 and 20 (inclusive).

For example, consider the ciphertext:

```
Ciphertext:  VHVSSPQUCEMRVBVBBBVHVSURQGIBDUGRNICJQUCERVUAXSSR
```

There are two repeated group of length 4, giving us a total of 4 groups of length 3: VHV, HVS, QUC, and UCE. We will count these VHV and HVS as separate groups even though they overlap, naturally giving higher weight to longer repeated groups. The sequence of lengths is 18, 18, 30 and 30, respectively, and the only possible guess that satisfies the conditions above is 6.

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 a single string, the *ciphertext*. You can assume that the ciphertext contains only capital letters (from A to Z) – all numbers or punctuation marks (including white spaces) are stripped out.

Output: For each test case, you are to output the guesses for keyword length between 4 and 20 (inclusive). The exact form of the output is shown below.

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 Vector<Integer> solveVigenereII(String cipherText)
```

The procedure should return a vector that contains the possible key lengths in increasing order.

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:
3 CSASTPKVSIQUTGQUCSASTPIUAQJB VHVSSPQUCEMRVBVBBBVHVSURQGIBDUGRNICJQUCERVUAXSSR NIOZSECPQETPGCGYMKQFE	 Possible key lengths between 4 and 20: 4 8 16 Possible key lengths between 4 and 20: 6 No guesses found.

7 Shut The Box II

The Avengers are still bored on their way to the Collector's hiding place, and are continuing to play Shut the Box. Unfortunately for Dr. Banner, the strategy that he was using with your help is not turning out to be a very good one, and he is losing most of the games. He is halfway to transforming into the Hulk, and you need to calm him down by finding the *best move* at any given point, defined to be the one that maximizes the *probability* of eventually shutting the box.

Assume that the dice are fair (i.e., with two dice, all of the 36 possibilities are equally likely). Keep in mind the twist mentioned earlier: when the total of the open cards is ≤ 6, only one die is used.

For example, say the current open cards are: 1, 2, and 3, and the rolled total = 3. There are 2 moves:

- **Shut 1 and 2:** In this case, we are left with 3. The probability that we will shut the box is then equal to the probability that we roll a 3 on the next try. Assuming fair dice (and keeping in mind we are only using 1 die), the probability = 1/6 = 0.166666.
- **Shut 3:** Then we are left with 1 and 2, and the analysis is a bit more complicated. However, we can show the probability of eventually shutting the box to be somewhat higher in this case. We state the computation here without derivation: 1/6 + 2/6 * 1/6 = 0.2222222. We should choose this move.

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 entry on each line is the rolled total (called *target*), and the next entry is the number of open cards n. The following n entries denote the open cards in the ascending order.

Output: For each test case, you are to output the move that maximizes the probability of eventually shutting the box, or output that there is no legal move. If there is a move, you also need to list out the probability of shutting the box moving forward. The exact form is shown below.

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 void solveShutTheBoxII(int[] values, int target)
```

Your code should set the variables `solution`, `solution_found` and `probability` appropriately.

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:
4	
3 3 1 2 4	The best move is: 1 2, and probability of shutting = 0.1667
3 3 1 2 3	The best move is: 3, and probability of shutting = 0.2222
6 3 1 2 3	The best move is: 1 2 3, and probability of shutting = 1
8 4 1 2 3 9	No move found.

8 Game of Stones

Captain America has just caught up with the supervillain Loki to retrieve the Cosmic Cube, and they are about to begin one of their epic battles. Loki is however tired of getting his butt kicked, and instead challenges Captain America to a battle of wits. Captain America is confident in his abilities (more precisely his trusty sidekick's, i.e., your, abilities) and accepts the challenge.

Loki then explains the game. He begins by drawing an acyclic directed graph (as shown in the figure), and places a number of stones on each of the nodes. The two players take turns moving a stone from any node to one of its neighbors, following a directed edge. So the stone on node 0 (in either of the examples below) can be moved to node 1, but a stone could not be moved from node 1 to node 0. Multiple stones may be at the same node at any time. The player that cannot move any stone loses the game, and the Cosmic Cube. Let a *sink node* be a node that has no outgoing edges. As long as at least one stone is at a non-sink node, a move can be made. Hence, the goal is to be the person who moves the last such stone from a non-sink node to a sink-node.

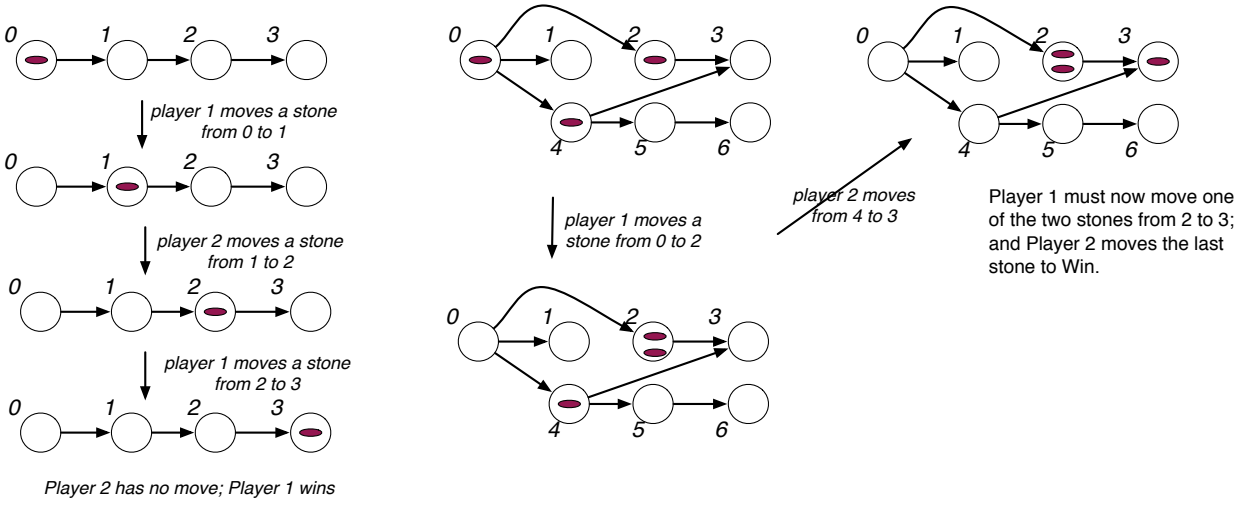

Figure 2: (1) In the first case, the moves are forced (there is no choice) and player 1 wins; (2) In the second case, no matter what Player 1 does in the first move, Player 2 can win.

Loki doesn't think you or Captain America are a match for him, and offers you the choice of moving first or second. You have to figure out what is the right choice.

Input/Output Format:

Input: The input consists of a number of test cases. Each test case begins with a line containing two integers n and m, the number of nodes and the number of edges respectively ($1 \leq n \leq 1000, 0 \leq m \leq 10000$). Then, the next line contais the m edges, each edge given by two integers a and b: the starting and ending node of the edge (nodes are labeled from 0 to $n-1$). The test case is terminated by n more integers $s_0, \cdots, s_{n-1}$ on the next line, where s_i represents the number of stones that are initially placed on node i ($0 \leq s_i \leq 1000$). Input is terminated by a line containing '0 0' which should not be processed.

Output: For each test case output a single line with either the word "First" if the first player will win, or the word "Second" if the second player will win (assuming optimal play by both sides).

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 boolean solveStones(int numNodes, ArrayList[] graph, int[] numStones)
```

Your code should return `true` if the first player will win, and `false` otherwise.

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:
4 3 0 1 1 2 2 3 1 0 0 0 7 7 0 1 0 2 0 4 2 3 4 5 5 6 4 3 1 0 1 0 1 0 0 0 0	First Second

9 DNA Manipulator

Bruce Banner has noticed that the Hulk has been putting on a few extra pounds. (The other Avengers are teasing him, calling him the "Incredible Bulk".) Bruce's dietician recommends that the Hulk get outdoors and exercise more and spend less time playing on his XBox. (Yeah, like that's going to happen.) Instead, Bruce is opting for DNA therapy, in an attempt to boost the Hulk's sputtering metabolism.

The resourceful Banner has found an old DNA splicing machine. The machine can be programmed to make certain limited changes to a DNA string. These are called production rules. Your job is to help Bruce determine whether he can build up the complex DNA strings that he'll be needing for his weight-loss program.

A *production rule* is of the form "$a \rightarrow bc$", where a, b, and c are each characters. A *production system* is a collection of production rules. For example, the following is a production system:

- Rule 0: A $\rightarrow$ GC
- Rule 1: A $\rightarrow$ AA
- Rule 2: G $\rightarrow$ TA

Given this production system we want to know whether certain strings can be generated from an initial starting symbol. For example, given the above system, is it possible to generate "TAGCC" from "A"? The answer is YES, and the derivation is given below:

1. $\underline{\text{A}} \Rightarrow \underline{\text{GC}}$ (apply rule 0 to A at position 0)
2. $\underline{\text{G}}\text{C} \Rightarrow \text{T}\underline{\text{A}}\text{C}$ (apply rule 2 to G at position 0)
3. $\text{T}\underline{\text{A}}\text{C} \Rightarrow \text{T}\underline{\text{AA}}\text{C}$ (apply rule 1 to A at position 1)
4. $\text{TA}\underline{\text{A}}\text{C} \Rightarrow \text{TA}\underline{\text{GC}}\text{C}$ (apply rule 0 to the second A at position 2)

Observe that each step of the derivation can be described by two numbers, the number of the *rule* to be applied and the *position* at which it is to be applied. (All indexing starts at 0).

You are to write a program that, given a production system P, a start symbol s, and a target string t, determines whether it is possible to generate t from s by applying the production rules. If so, it will return a derivation. A *derivation* involving m steps is given by a list of $2m$ integers, giving the rule number and position of each substitution. For example the above 4-step derivation can be expressed by the 8-element list $\langle 0, 0, 2, 0, 1, 1, 0, 2\rangle$.

If it is not possible to derive t from s, your program should return an *empty* derivation list. Also, if a trivial derivation exists (because $s = t$) your program should return an *empty* derivation. In general, many derivations may be possible. Your program may produce any one valid derivation.

Input and Output

The input file consists of a number of test cases, where each test case consists of a production system followed by a series of derivations. Each production rule is given on a new line and the system is terminated by a single line containing "$". This is followed by a sequence of source/target pairs terminated by a single line containing "#". The symbols used in the productions may be any alphabetic character or underscore ("a–z", "A–Z" or "_").

Input:	**Explanation:**
`1`	Number of test cases
`A GC`	Rule 0: A → GC
`A AA`	Rule 1: A → AA
`G TA`	Rule 2: G → TA
`$`	(end of production rules)
`A TAGCC`	start = A, target = TAGCC
`A A`	start = A, target = A
`A AB`	start = A, target = AB
`#`	(end of derivations)

We will provide a program that handles all the input and output. All you need to do is to provide the body of the following procedure:

```
private static ArrayList<Integer> computeDerivation(
  ArrayList<Character> lhs,       // production rule left-hand-sides
  ArrayList<String> rhs,          // corresponding right-hand-sides
  Character start,                // start symbol
  String target)                  // target string
```

- A character array `lhs`, where `lhs[i]` contains the left-hand side of the ith rule
- A string array `rhs`, where `rhs[i]` contains the right-hand side of the ith rule
- A character `start`, which is the start symbol
- A string `target`, which is the target string

This procedure returns an ArrayList of integers containing the derivation, as described above. Our program will check that your derivation is valid and, if so, it will generate the output shown below. If the derivation is not valid (e.g., it applies a rule improperly), it will produce an error message.

For the first derivation, suppose that your program returns an ArrayList $\langle 0, 0, 2, 0, 1, 1, 0, 2\rangle$. For the other two, it returns an empty derivation list. Here is the output that would result.

Sample output: (as produced by our output procedure)

```
Test case: 0
  Productions: (lhs --> rhs)
    [0]: A --> GC
    [1]: A --> AA
    [2]: G --> TA

  Derive "TAGCC" from "A"? YES!
    Derivation: (0,0) (2,0) (1,1) (0,2)
      Apply[0] at 0: S ==> GC
      Apply[2] at 0:   ==> TAC
      Apply[1] at 1:   ==> TAAC
      Apply[0] at 2:   ==> TAGCC
  Derive "A" from "A"? YES! (trivially)
  Derive "AB" from "A"? NO!
```