2016 University of Maryland High School Programming Contest

1. Jedi Crystals 3

2. Aligned Gaps in Asteroid Field 5

3. Forced Cards 7

4. Legal Moves in Go 9

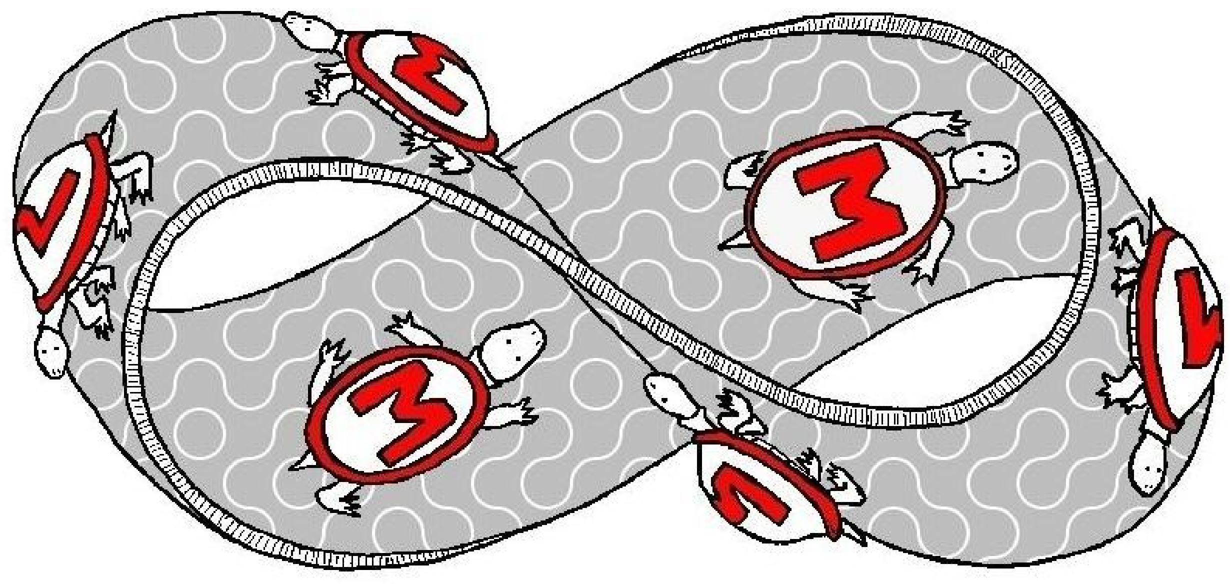
5. Scavenger One 12

6. Tree Ragnorianism 14

7. Combination Puzzle 17

8. Opening Doors 19

9. Winning a Cards Game 21



1. Jedi Crystals

A new Kyber crystal mine has been started on Dantooine. These are the crystals for lightsabers, and the mine owner has been mining red, blue, and green crystals. Periodically, the mine owner pauses the mining to put together all the crystals found till then, and ships them out to lightsaber factories (yes, they exist!). For reasons of superstition, he only does this when an equal number of crystals of each color has been collected from the mine (i.e., when there is “balance”). However, since he has no control over which color crystal is found next, he wants your help to decide when balance is reached.

The program will read in a list of letters representing the color of each crystal that comes out of the mine. When a balance (greater than 0,0,0) has been achieved, the program will stop reading and print "Balance achieved at " followed by the number of red crystals that came out (which of course would be the same as the number of blue and the number of green crystals).

The provided skeleton handles the input of the values and the output messages. You need to implement the method

```
achieveBalance(Scanner sc)
```

which returns an

```
int
```

when balance has been achieved. The entire input may not be read in the first call -- instead, the function is called repeatedly in order to read all the input.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases. After that, for each test case there will be one letter (R,G,B) per line representing the color of the crystal that just came out.

Output:

For each test case, the program will display the "balance" number. You can assume that for the given input file, balance will be achieved exactly as many times as the number of test cases listed, and also that there will be no more lines in the input after the last balance.

Note:

We have provided a skeleton program that reads the number of test cases and opens the input stream. It also prints the output based on the `achieveBalance()` method that you need to implement.

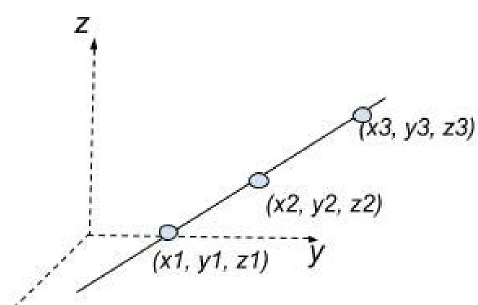
Examples:

The output is shown with extra blank lines so that the output is shown aligned with the appropriate input line; those blank lines should not be present in the actual output. (The provided skeleton code handles that for you).

Input:	Output:
`3`	
`R`	
`G`	
`B`	`Balance achieved at 1`
`R`	
`R`	
`G`	
`G`	
`B`	
`B`	`Balance achieved at 2`
`R`	
`G`	
`R`	
`G`	
`R`	
`G`	
`B`	
`B`	
`B`	`Balance achieved at 3`

2. Aligned Gaps in Asteroid Field

Poe and Finn have escaped the Finalizer and are trying to run away, chased by several enemy ships. Poe notices that they are approaching an asteroid field at a very fast pace, the pace at which their spaceship is incapable of maneuvering and must go in a straight line. There are gaps in the asteroid field, but Poe can't tell whether they are in a straight line for him to be able to go through all the way. You are to help Poe decide whether a particular set of gaps is aligned (collinear) or not.



The location of each gap is given in the form of its (x, y, z) coordinates. Your program should compute whether the set of gaps is collinear, i.e., whether they lie on a single straight line. Any two points are collinear since we can just draw a straight line between them. One approach for deciding whether three points (x1, y1, z1), (x2 ,y2, z2), and (x3, y3, z3) are collinear, is to check whether there exists a number *r* such that all of the following are true:

$$(x3 - x1) = r (x2 - x1), \quad (y3 - y1) = r (y2 - y1), \quad (z3 - z1) = r (z2 - z1)$$

Specifically, we will follow the following steps to decide if a set of *n* distinct points is collinear.

(1) Compute *xdiff = x2 - x1, ydiff = y2 - y1*, and *zdiff = z2 - z1*.
(2) Say *xdiff* has the largest *absolute value* among those three (i.e. |xdiff| > |ydiff|, |xdiff| > |zdiff|)
(3) For i = 3 to n:
    (a) Compute *r = (x3 - x1)/xdiff*
    (b) Check whether *|(y3 - y1) - r * ydiff | < EPSILON*, where EPSILON is very small value to handle errors in dealing with floating point numbers (use EPSILON = 0.00001). |x| denotes the absolute value of x.
    (c) Similarly: check if *|(z3 - z1) - r * zdiff | < EPSILON*.
    (d) If both of them are true, then point 3 is collinear with points 1 and 2.
(4) If *ydiff* or *zdiff* are the largest, the above steps change accordingly.
(5) The above steps fall in the degenerate case when all of *xdiff, ydiff,* and *zdiff* are 0 (i.e., points 1 and 2 are identical), because of a divide-of-zero. But since we assume all the *n* points are distinct, this is not an issue for us.
(6) Return true if all the other points are collinear with points 1 and 2; return false otherwise.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases, and then the test cases are listed one by one. The first line of a test case contains the number of gaps *n*, and after that the coordinates of the *n* gaps/points are listed (as 3 doubles each) on one line each. Assume *n <= 100*.

Output:

For each test case, your program should output whether the gaps are aligned (collinear) or not using the inequalities listed above.

Note:

We have provided a skeleton program that reads the input and prints the output based on the following `areGapsAligned()` method.

```
private static boolean areGapsAligned(double coordinates_x[],
                              double coordinates_y[], double coordinates_z[])
```

Examples:

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

Input:	**Output:**
2 3 1.0 1.0 1.0 2.0 2.0 2.0 3.0 3.0 3.0 3 1.0 1.0 1.0 2.0 2.0 2.0 3.0 3.0 4.0	 Gaps are aligned. Gaps are NOT aligned.

3. Forced Cards

Poe Dameron and BB-8 are in a little hot water. They need to win money to buy parts for Poe's X-Wing. There's a local card game where a random group of cards (at least one with a positive value, but some could have negative values) are selected and then shuffled. The player begins by giving a number. The casino then starts to deal out the cards face down. The player watches the cards coming out and says "start" on one card and then "stop" on another card. Then the casino adds up all the card values between where the player called *start* and where the player called *stop* (including the two cards)*;* if this total is identical to the number that the player called out, the player wins that amount of money.

BB-8 is able to scan the deck while it is face down, and knows all of the values of the cards and the sequence. What he doesn't have is a method where he can pass in the values and get back the start/stop points that would give the biggest win possible, i.e., the start/stop points that will give the highest total.

The program will read in list of card values. It will then print the start and stop points with the highest total, as well as the total between those points.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases. After that, for each test case there will be an integer saying how many cards there are, and then each line after that will be the value of a card. Assume the number of cards in any test case is <= 100.

Output:

For each test case, the program will display the start and stop points as well as the total between those points for that deck. If there are multiple answers (i.e., multiple start/stop pairs with highest total), your code should return the one with the earliest start point; if there are multiple start/stop pairs with highest total that start at the same point, it should return the one with the smallest stop point.

Note:

We have provided a skeleton program that reads the input and prints the output based on the `bestPlay()` method that you need to implement. The method takes as input the sequence of card values, and returns an array of 3 integers (start/stop/total).

```
        private static int[] bestPlay(int[] deck)
```

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.

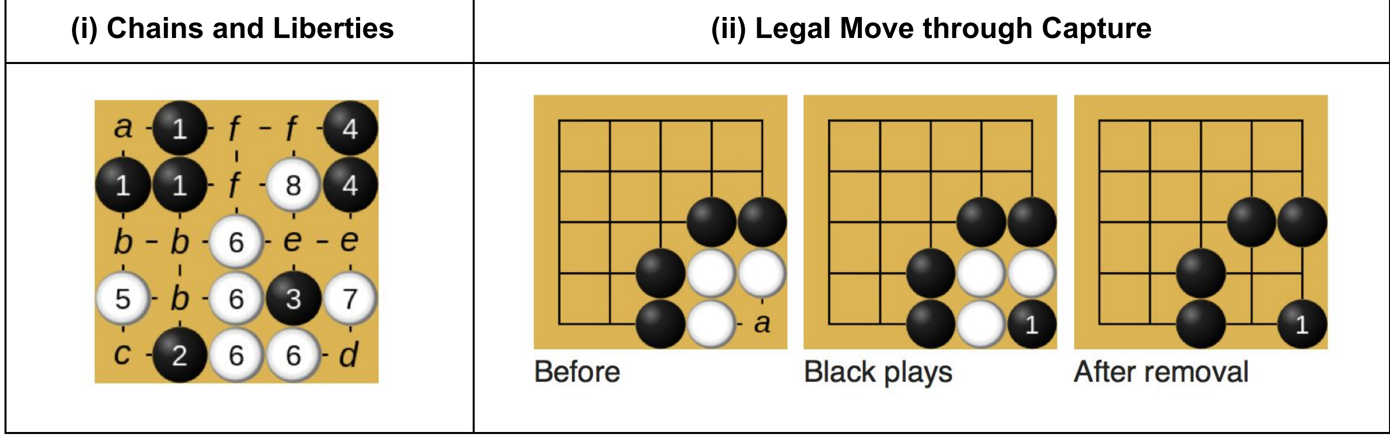
Input:	Output:
1 7 99 3 -17 19 -96 2 2	 Start/Stop/Value: 0/3/104

4. Legal Moves in Go

BB-8 is continuing to learn the game of Go so he can win against Chewbacca by employing the strategies of AlphaGo, the neural network-based Go engine that was recently developed. The rules of Go are very confusing though, and BB-8 would like your help.

Go is played on an *n-by-n* board (typically 19-by-19, but 5-by-5 is used for teaching) as shown below, and the two players (one playing white stones, and the other playing black stones) take turns placing stones on the board. Stones cannot however be placed arbitrarily. To begin with, a stone can only be placed at a position if the position is empty. Further, the position must have an adjacent "liberty", i.e., another open position that it is connected to. The simplest case is when the one of the immediately adjacent positions is also open (e.g., either of the two positions labeled "e" below satisfy this condition).

However, two stones of the same color that are adjacent to each other "share" their liberties. To define this more formally: two stones are called "connected" if they are of the same color, and they are on adjacent locations. A "*chain"* is a set of one or more stones (necessarily of the same color) that are all connected to each other. In the example figure (i) below, the different chains are shown by using numbers, i.e., all stones in the chain are given the same number. For example, there is a chain of 3 black stones (numbered 1) at the top-left corner.



All the stones in a chain share liberties. For example, the position marked "d" (bottom-right) is considered a "liberty" of all stones numbered 6, even though several of those are not adjacent to it.

Given this, we can now more fully define a "legal" move in Go. A stone can be placed in a position only if that stone (after placing it) will have a liberty (either directly adjacent or shared through a chain). For example, a white stone can be placed in the position labeled "d" because it has an adjacent liberty through either of the two chains numbered 6 or 7. Similarly, a black stone can be placed in position labeled "a".

However: considering this rule alone, in the second example (Figure (ii)), a black stone is not allowed to be placed in position "a". However, Go has a notion of "capturing" stones. If a chain is enclosed on all sides by stones of the other color, then the stones in that chain are "captured" and removed from the board. In the example (ii), after placing a black stone in position "a", the three white stones will be removed, thus satisfying the "adjacent" liberties rule (in other words: the position "a" was the only liberty for those three white stones, and hence it was a legal placement for a black stone).

Input/Output Format

Input:

The first line in the test data file contains the number of test cases, and then the test cases are listed one by one. The first line of a test case contains the size of the board, *n*; a single character *B* or *W,* indicating whether BB-8 is playing black stones or white stones; and a pair of numbers indicating the position of interest. The position *(0, 0)* refers to the top left corner of the board, and *(1, 0)* refers to the position to the right of it. After that, the current state of the board itself is listed on *n* lines, each of which contains a single string of length *n.* The string uses characters *B, W,* and *. (dot)* to indicate the state of any specific position. Assume *n <= 20*.

Important note: Do not assume that the provided board is valid according to these rules; e.g., it may contain captured stones that haven't yet been removed.

Output:

For each test case, your program should output whether the intended move is "Legal" or "Illegal", along with an brief explanation (as shown below).

Note:

We have provided a skeleton program that reads the input and prints the output based on the following `checkLegality()` method. `checkLegality()` is passed the *n* strings as an array, a character ('B' or 'W'), and the position of interest as two numbers. It should return one of the four constants defined in the file (NOT_EMPTY, LEGAL_ADJACENT_LIBERTY, LEGAL_THROUGH_CAPTURE, ILLEGAL).

```
private static int checkLegality(String board[], char c,
                                 int pos_x, int pos_y)
```

Examples:

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

Input:	**Output:**

Input	Output
5	
3 W 0 0 B..	Move W in position (0, 0) is illegal - the position is not empty.
3 W 1 1	Move W in position (1, 1) is legal - there is an adjacent (shared) liberty.
3 B 1 1 .W. W.W .W.	Move B in position (1, 1) is illegal.
3 W 1 1 .W. W.W .W.	Move W in position (1, 1) is legal - there is an adjacent (shared) liberty.
3 B 2 2 .BB BWW BW.	Move B in position (2, 2) is legal through capture of opponent's stones.

5. Scavenger One

Rey has a variety of scavenged items she can bring to Plutt each day. On any given day she will have no more than around 30 or 40 such items. She knows her speeder's weight capacity; each morning she weighs herself and knows exactly how many pounds of scrap she can bring back. If she can't bring back exactly that much, she'll go into the ships and find more things to try to get a combination of items that's just the right weight.

The program will read in a weight capacity for the day and a list of item weights based on what's available to bring back from the ships. It will then either print a list of the weights of the items that she should load up or a message letting her know that the exact capacity can't be reached. The provided skeleton handles the input of the values and the output messages. You need to implement the method

```
bestItems
```

which returns a

```
int[]
```

Input/Output Format

Input:

The first line in the test data file contains the number of test cases. After that, for each test case there will be an integer capacity, an integer value saying how many items there are, and then each line after that will be the weight of an item.

Output:

For each test case, the program will display the list of item weights for the run, or a message saying the exact capacity can't be reached.

Note:

We have provided a skeleton program that reads the input and prints the output based on the `bestItems()` method that you need to implement.

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.

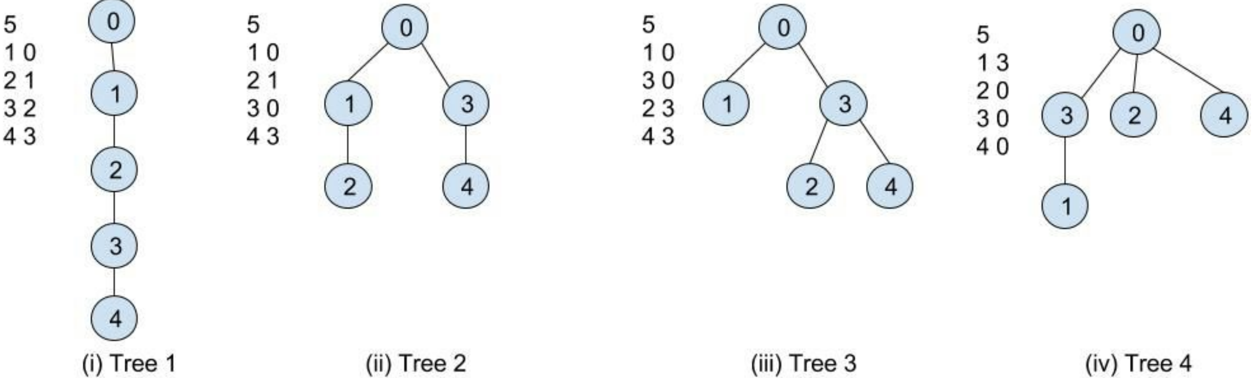
Input:	**Output:**

2	
100 5 99 3 96 2 2	This run is: 96, 2, 2
7 5 2 4 6 8 10	Cannot fill to capacity.

6. Tree "Ragnorianism"

BB-8 is carrying a partial map that supposedly leads to Luke Skywalker. But unfortunately the map is not only incomplete but has also been scrambled to make it difficult to interpret. Han Solo thinks that he can figure out which part of the galaxy corresponds to the incomplete map by examining structural features of the map. Specifically: he converts the map into a "tree", where the "nodes" of the tree correspond to landmarks on the map and there is an edge between two nodes if they are connected by a road. (He only keeps enough edges to ensure that the structure he draws is a valid tree). Similarly he takes a part of the galaxy and constructs a similar tree for that. However, in order to decide if those two are equivalent, he needs to find a "Ragnorianism" between them, i.e., a one-to-one mapping between the nodes that allows converting one tree into the other. Two trees are called Ragnorianic if such a mapping exists. See examples below.

Han would like your help with finding such a mapping, if one exists. Your program should take as input two rooted trees. The nodes in each tree are labeled from *0* to *n-1*, with *0* always being the root. For each non-root node, the "parent" of the node is provided. The figure below shows 4 trees on 5 nodes, and the input representations of those trees.



The first two of these trees are Ragnorianic -- the root of the first tree maps to either node 2 or node 4 in the second tree. Tree 3 and Tree 4 are similarly Ragnorianic -- root of the first tree maps to node 3 in the second tree. However, there is no such correspondence between Tree 1 and Tree 3 (or any other pair of the trees).

Input/Output Format

Input:

The first line in the test data file contains the number of test cases, and then the test cases are listed one by one. The first line of a test case contains the number of vertices, *n* (assume *n <= 500)*. The next *n-1* lines contain the first tree; specifically, each line contains two numbers *a b*, where *a* is the id

of a vertex, and *b* is its parent. The next *n-1* lines contain the second tree. The roots of both the trees are vertex 0. You are guaranteed that each of the provided inputs is a valid tree.

Output:

For each test case, your program should output whether the trees are Ragnorianic or not. If they are Ragnorianic, you are also to list which node in the second tree maps to the node labeled 0 in the first tree (i.e., the root of the first tree). If there are multiple different mappings, you should return the smallest of such nodes (e.g., 2 in the example above, where there two different mappings).

Note:

We have provided a skeleton program that reads the input and prints the output based on the following `findRagnorianism()` method. `findRagnorianism()` is passed the 2 trees as int arrays. It should return -1 if the trees are not Ragnorianic; otherwise, it should return the node from the second tree to which the root of the first tree maps to in the Ragnorianism (smallest one if there are multiple mapping).

```
    private static int findRagnorianism(int[] parents_tree1,
                                        int[] parents_tree2)
```

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.

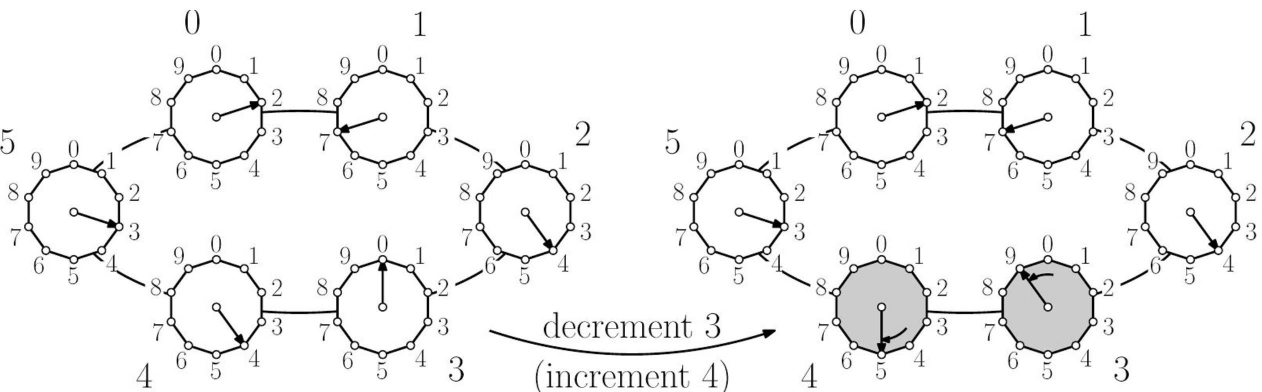
Input:	**Output:**
3	
5	The trees are Ragnorianic -- root of the first tree maps to node 2 of the second tree.
1 0	
2 1	
3 2	
4 3	
1 0	
2 1	
3 0	
4 3	
4	The two trees are not Ragnorianic.
1 0	
2 0	
3 0	
1 0	
2 1	
3 2	

5 1 0 2 0 3 2 4 2 1 0 2 0 3 0 4 3	The trees are Ragnorianic -- root of the first tree maps to node 3 of the second tree.

7. The Combination Puzzle

On planet Takodana, Rey meets with Maz Kanata who shows her a case containing Luke's lightsaber, but the case is locked! The locking mechanism consists of *n* dials, where each dial can be set to a digit from 0 to 9. Maz explains that the unlocking process consists of a number of steps. At each step, any dial can be turned one position either clockwise (increasing the dial's value by one) or counterclockwise (decreasing its value by one). The dial values wrap around, so increasing a value of 9 results in 0 and decreasing a value of 0 results in 9.

There is a complication in how the dials move. Let us number the dials from 0 to *n*-1. When dial *i* is turned clockwise one position, dial *i+1* automatically turns counterclockwise one position. Similarly, if dial *i* is turned counterclockwise one position, dial *i+1* automatically turns clockwise one position. Dial indices wrap around, so turning dial *n-1* affects the value of dial *0*. For example, the figure below shows the effect of turning dial 3 counterclockwise, decreasing its value and increasing dial 4's value.



In order to unlock the case, all the dials must be set to 0. Rey is in a hurry, and because the dials turn very slowly, she needs to determine the minimum number of steps needed. Note that it may be impossible to unlock the case.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases, and each line after that contains a test case. The first number is *n*, the number of dials (at most 50). This is followed by *n* digits, giving the initial dial settings. The instance in the above figure left would be expressed as "6   2 7 4 0 4 3".

Output:

For each test case, your program should output a string that encodes a minimum length sequence of steps in order to unlock the case. If it is not possible to unlock the case, then the string contains the

single character “X”. Otherwise, the string consists of a sequence of steps, separated by spaces. Each step consists of a ‘+’ or ‘-’ (indicating whether to turn clockwise or counterclockwise) followed by a dial number from 0 to *n*-1 (indicating which dial is to be turned). For example, the move shown in the above figure would be expressed as “-3”.

For example, if *n* = 8, the string “+1 +5 -3 -3 -7” means that dials 1 and 5 are each incremented by one (resulting in dials 2 and 6 each being decremented by one), then dial 3 is decremented twice (resulting in dial 4 being incremented twice), and dial 7 is decremented (resulting in dial 0 being incremented). Note that there may be many valid solutions, and your output will be judged correct if it sets all the dials to the desired final setting and involves the minimum number of steps.

Note:

We have provided a skeleton program that reads the input and prints the output based on a method `SolvePuzzle(),` which you will provide. This method is passed an integer array containing the *n* initial dial settings. It should return a string that encodes your solution as described above.

```
   private static String SolvePuzzle(int[] dial)
```

We have also provided validation code that checks whether your answer is correct or not.

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. Note that this shows *just one* of many possible correct outputs for each input.

<table>
<tr><th>Input:</th><th>Output:</th></tr>
<tr><td><pre>3
2   9 1

2   1 1

3   4 7 9</pre></td><td><pre>
Initial dial settings = [9 1]
Raw output = "-1"
 Analysis:
  >> Valid solution. Total cost = 1

Initial dial settings = [1 1]
Raw output = "X"
 Analysis:
  >> No solution

Initial dial settings = [4 7 9]
Raw output = "-0 -0 -0 +2"
 Analysis:
  >> Valid solution. Total cost = 4</pre></td></tr>
</table>

8. Opening Doors

Rey and her friends find themselves in a long corridor with many (say, *n)* doors heading in different directions, and a switchboard with a number of switches (*m)* on them. Both the doors and the switches are numbered from *0* to *n-1* and *0* to *m-1* respectively. A switch however, does not open or close a single door; instead, each switch toggles the state of a subset of the doors. In the example below, the first switch (Switch 0) will toggle the states of the doors 0, 1, 3, and 4; if one of those doors is open, it will be closed, and if one of them is closed, it will be opened. Similarly, the second switch will toggle the states of the doors 1, 2, and 4. The doors are all initially closed, and Rey needs to open a specific subset of the doors (they are going to split up to do the different tasks), and no others (to ensure no stormtroopers can intercept them). They only have one shot at going through the corridor, so she can't open the doors one after the other; instead she must toggle a subset of the switches so the desired configuration is reached, and then they will all start running towards their respective doors.

Switch 0	Toggles doors 0, 1, 3, and 4
Switch 1	Toggles doors 1, 2, and 4
Switch 2	Toggles doors 1 and 3
Switch 3	Toggles door 1
Switch 4	Toggles doors 2, 3 and 4

For example, if the goal was to have exactly doors 1, 2, 3, and 4 open, then we can use the switches 1, 2, and 3. Switch 1 will open doors 1, 2, and 4. Switch 2 will close door 1 and open door 3, and then Switch 3 will again open door 1. Another way to achieve that would be to use switches 3 and 4. On the other hand, there is no way to get the doors 2 and 3 open, without keeping any other doors open.

You have to help Rey decide if it is possible to achieve the desired configuration, and a set of switches to toggle to achieve that configuration. Any solution would be accepted -- it is not required that the solution be minimal.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases, and the test cases are listed after one by one. The first line contains two numbers, *m* and *n*, the number of switches and the number of doors respectively. After that, the next *m* lines tell us which doors each of the switches will toggle. The first number of each line is the number of doors that the switch will toggle, and then the doors themselves are listed one by one. After those *m* lines, the next line indicates which doors we want to

keep open at the end. (The first number is again the number of doors we want open, followed by the specific doors). Assume *n <= 500, m <= 500*.

Output:

For each test case, your program should output a set of switches to be toggled to ensure that all the required doors are open, and no other doors are open. Any solution would be fine. If there is no solution, then your program should say so.

Note:

We have provided a skeleton program that reads the input and prints the output based on the following `chooseSwitches()` method. `chooseSwitches()` is passed the *m+1* arrays of integers. It should return an ArrayList containing the switches to be toggled, and *null* if the desired configuration is not achievable..

```
private static ArrayList<Integer> chooseSwitches(int numSwitches, int numDoors,
                    ArrayList<Integer> switchOpens[], ArrayList<Integer> needOpen)
```

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. For the third test case, the answer “Toggle switches [3, 4]” would also be accepted.

Input:	Output:
3 3 3 1 0 2 0 1 3 0 1 2 2 0 2	 Toggle switches [0, 1, 2]
3 3 1 2 2 0 1 3 0 1 2 1 1	No combination of the switches can result in the desired configuration.
5 5 4 0 1 3 4 3 2 4 1 2 1 3 1 1 3 2 3 4 4 1 2 3 4	Toggle switches [1, 2, 3]

9. Winning a Card Game

Poe Dameron and BB-8 haven't been able to win enough money to buy parts for Poe's X-Wing using the simple card game, and are instead going to try winning at another card game with higher stakes. In this game, there are two decks of cards, a *player* deck and a *dealer* deck. The game goes in rounds, which continue until the *player* deck is exhausted.

In each round, the casino starts dealing from the player deck until the player says "stop". All the cards that have been dealt in the round are then summed up (at least one player card must be dealt in each round). Then the top card from the dealer stack is opened, and its value is compared with that total. The player wins $10 if the total is higher than the dealer card, $5 if the two numbers are equal, and nothing otherwise. All the cards dealt in the round (including the dealer card) are set aside, and the next round continues with the remainder of the decks. The last round will end with the last player card, i.e., when the last card from the player deck has been dealt. Note that there may still be remaining cards in the dealer deck at that point.

As an example, let the values of the two sequences of cards be:
Player Deck: 0 5 9 2 3 4
Dealer Deck: 3 8 3 2 1 1

Case 1: The player says "stop" after every card is dealt. In that case, the player will lose the first round (0 vs 3), lose the second round (5 vs 8), win the third round (9 vs 3), tie the fourth round (2 vs 2), and win the remaining two. So the total winnings will be $35.
Case 2: The player does not say "stop" till the very end. In that case, there is only one round; the total would be: 0+5+9+2+3+4 = 23, and he wins against the dealer card of 3. However, the total winnings will be just $10.
Case 3: The player says "stop" after cards 0 and 5 are dealt (round 1), then after card 9 is dealt (round 2), then after 2, 3 are dealt (round 3), and finally at the end (after 4 is dealt). He wins in each case (against dealer cards: 3, 8, 3, and 2 respectively). The total winnings are $40, which is highest possible.

BB-8 is able to scan both the stacks and knows all the cards that will be dealt and the sequence. Your job is to find the best points at which to stop to maximize the winnings. If there are multiple possible strategies to win, you should find the one that requires the minimum number of stops.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases. After that, for each test case there will be an integer indicating the number of cards, and then the two sequences (player, and dealer stacks) are listed on one line each. Assume the number of cards in any test case is <= 50.

Output:

For each test case, the program will display the stop points as well as the maximum total winnings.

Note:

We have provided a skeleton program that reads the input and prints the output based on the `bestPlay()` method that you need to implement. The method takes as input the two sequences of card values, and an arraylist. Your code should fill in the arraylist with the stop points (in order), and return the total winnings.

```
private static int bestPlay(int[] cards_player,
                            int[] cards_dealer, ArrayList<Integer> solution)
```

The sequences should be treated as 0-indexed; i.e., a stop point of 0 means stopping after cards_player[0] has been dealt. Since the player always has to call "stop" at the last card, the last entry in the solution ArrayList should always be *cards_player.length-1.*

Examples:

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

Input:	Output:
3 4 5 8 1 3 8 0 2 1 4 2 3 5 2 1 4 5 2 6 0 5 9 2 3 4 3 8 3 2 1 1	 Best value 30 is obtained by stopping at positions: [1, 2, 3] Best value 20 is obtained by stopping at positions: [0, 3] Best value 40 is obtained by stopping at positions: [1, 2, 4, 5]

Practice 1: Monotonic Sequences

To pass time on board the freighter Eravana, BB-8 starts teaching Finn about interesting number sequences (turns out teaching Math to stormtroopers is not a priority for the First Order). BB-8 starts with the most basic of sequences, i.e., monotonic sequences. A sequence of numbers: $a1, a2, \ldots, a_k$, is called monotonically increasing if:

$$a1 <= a2 <= \ldots <= a_k$$

Similarly, it is called monotonically decreasing if:

$$a1 >= a2 >= \ldots >= a_k$$

It is called monotonic if either is true, and non-monotonic otherwise. A sequence with all identical numbers is both monotonically increasing and decreasing by this definition. You are to help Finn better understand this concept by telling him whether a particular sequence satisfies either of the properties.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases, and each line after that contains a test case. The first number in each line, *n*, is the length of the sequence, and the *n* numbers after that are the sequence itself.

Output:

For each test case, your program should output whether the sequence is monotonically increasing, or monotonically decreasing, or neither. A sequence may contain repeated elements, but you can assume that it has at least two different numbers, so it cannot be both monotonically increasing and monotonically decreasing at the same time.

Note:

We have provided a skeleton program that reads the input and prints the output based on the following `findMonotonicity()` method. `findMonotonicity()` is passed the sequence as a int array. It should return -1 if the sequence is monotonically decreasing, 1 if the sequence is monotonically increasing, and 0 otherwise.

```
private static int findMonotonicity(int seq[])
```

Examples:

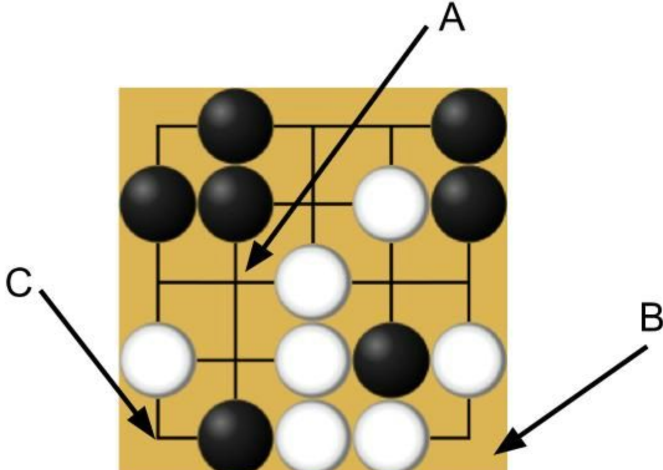
The output is shown with an extra blank line so that each test case input is aligned with the output; the first blank line should not be present in the actual output.

Input:	**Output:**
`5`	
`3 1 2 3`	`The sequence is monotonically increasing.`
`4 3 2 1 4`	`The sequence is neither monotonically increasing or decreasing.`
`4 10 8 7 1`	`The sequence is monotonically decreasing.`
`4 -4 -3 -2 -1`	`The sequence is monotonically increasing.`
`2 0 1`	`The sequence is monotonically increasing.`

Practice 2: Learning to Play Go

On board of Eravana, the freighter ship of Han and Chewbacca, Chewbacca boasts to the droid BB-8 about how he has never lost a game of "Go" to a computer, which is known to be very difficult for computers to play. However, Chewbacca has not heard about the latest developments on a planet far far away (called "Earth") where a team of researchers had created a program called "AlphaGo", which had finally beaten one of the best Go players on Earth. BB-8 decides to emulate AlphaGo's techniques to try to beat Chewbacca; however, since it doesn't actually know anything about the game, it starts by trying to read the rules. The rules of Go are very confusing though, and BB-8 would like your help.

Go is played on *n-by-n* board (typically 19-by-19, but 5-by-5 is used for teaching) as shown below, and the two players (one playing white stones, and the other playing black stones) take turns placing stones on the intersections on the board. Stones cannot however be placed arbitrarily. To begin with, a stone can only be placed at a position (i.e., an intersection) if the position is empty. Further, the position must have an adjacent "liberty", i.e., another open position. The simplest case is when the one of the immediately adjacent positions is also open. For example, in the example below, the position pointed to by A has two immediately adjacent liberties: one to the left of it, and one below it. Note that: the empty position diagonally above and to the right of A, is not considered adjacent. On the other hand, the position pointed to by B has no immediately adjacent liberties; location pointed to by C also does not have any immediately adjacent liberties (again the diagonally adjacent location does not count).



BB-8 would like your help to make sure it understands the basic rules correctly. Specifically, it wants your help in finding the number of immediately adjacent liberties for any given position on a board.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases, and then the test cases are listed one by one. The first line of a test case contains the size of the board, *n*; and two numbers, *(a, b),* indicating the position of interest. The position *(0, 0)* refers to the top left corner of the board, and *(1, 0)* refers to the position to the right of it. After that, the current state of the board itself is listed on *n* lines, each of which contains a single string of length *n.* The string uses characters *B, W,* and *.* to indicate the state of any specific position (*black stone, white stone, empty)*. Assume *n <= 100*.

Output:

For each test case, your program should output the current state of the listed position as: “Black”, “White”, “Open - m adjacent liberties”, where *m* is the number of adjacent positions that are also open.

Note:

We have provided a skeleton program that reads the input and prints the output based on the following `findStatus()` method. `findStatus()` is passed the *n* strings as an array, and the position of interest as two numbers. It should return -2 if the position contains a white stone, -1 if the position contains a black stone, or a number between 0 and 4, indicating that the position is open and listing the number of adjacent positions that are also open.

```
private static int findStatus(String board[], int pos_x, int pos_y)
```

Examples:

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

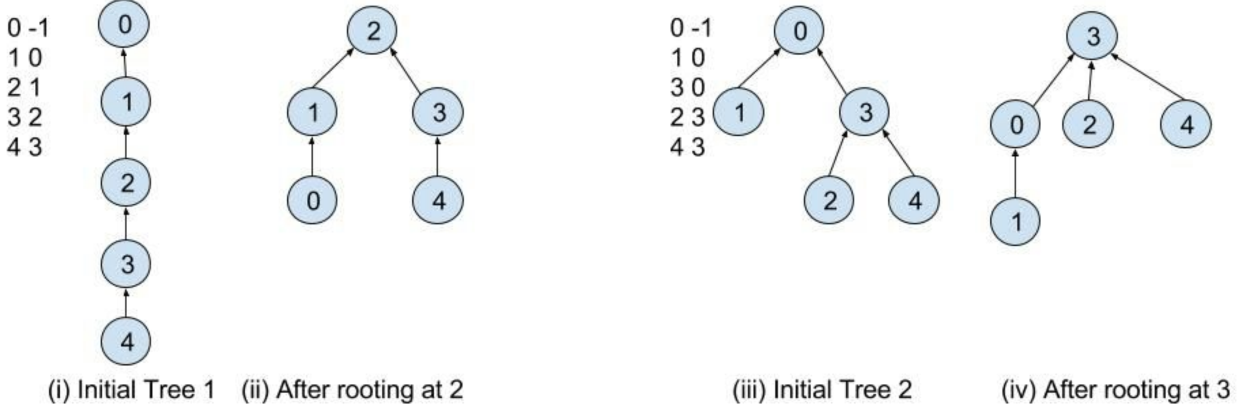
Input:	**Output:**
5	
3 1 1	Open - 4 adjacent liberties
...	
...	
...	
3 1 1	Open - 0 adjacent liberties
.W.	
W.W	
.W.	
3 0 1	White stone

.W. W.W .W.	
3 1 0 .W. .W. ...	White stone
3 1 2 B.. ... .B.	Black stone

Practice 3: Tree Re-orientation

Finn has advanced beyond basic Mathematics, and is starting to learn about graph theory. A graph represents the interconnection structure among a set of objects. Specifically, a graph consists of a set of *vertices (*also called *nodes),* and *edges* connecting pairs of vertices. A graph is called *connected* if there is a path connecting any two vertices, and a connected graph without cycles is called *tree.*

Since trees are the simplest type of graphs, Finn starts with learning about them. Most trees that are studied are *rooted,* i.e., one of the nodes is designated as the *root* of the tree. This allows us to more easily manipulate and draw trees. Figure below shows 2 different trees over 5 nodes labeled 0 to 4, with two different roots each. Figure (i) shows a tree rooted at 0, where Figure (ii) shows the same tree rooted at 2. A rooted tree is fully described by specifying the root, and the parents for all the non-root nodes. (For convenience, the parent of the root is listed as -1). On the left of Figure (i), this representation is shown for the first tree (since parent of 0 is given as -1, 0 is the root; the next line tells us that parent of 1 is 0, and so on).



(i) Initial Tree 1 (ii) After rooting at 2 (iii) Initial Tree 2 (iv) After rooting at 3

You are to write a program that “re-roots” a tree, i.e., changes the root of the tree, and counts the number of nodes in each of the “subtrees” below the root. For example, in the first case: after re-rooting the tree at 2, the new root has two subtrees, each of which contains 2 nodes. In the second case, after re-rooting the tree at 3, the new root has three subtrees, containing 2, 1, 1 nodes respectively.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases, and then the test cases are listed one by one. The first line of a test case contains the number of vertices, *n*; the current root, *r*, and the new root, *r'*. Both *r,* and *r'* are numbers between *0* and *n-1.* The next *n-1* lines contain the current structure of the tree; specifically, each line contains two numbers *a b*, where *a* is the id of a vertex, and *b* is its parent. You are guaranteed that the provided input forms a valid tree.

Output:

For each test case, your program should reroot the tree (i.e., reorient the tree such that the tree is rooted at *r'*), and output the number of nodes in each of the subtrees rooted at the children of *r'* (as discussed above). These numbers should be output in an increasing order.

Note:

We have provided a skeleton program that reads the input and prints the output based on the following `findNewSubtreeSizes()` method. `findNewSubtreeSizes()` is passed the current root, the new root, and the list of edges as a parents array (with the parent for the root set to -1). It should return an ArrayList<Integer> that contains the required sizes, in the required order.

```
   private static ArrayList<Integer> findNewSubtreeSizes(int
currentRoot,
                                   int newRoot, int[] parents)
```

Examples:

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

Input:	Output:
3	
5 0 0	After re-orienting, the subtree sizes under the new root: [4]
1 0	
2 1	
3 2	
4 3	
5 0 1	After re-orienting, the subtree sizes under the new root: [1, 3]
1 0	
2 1	
3 2	
4 3	
5 0 2	After re-orienting, the subtree sizes under the new root: [2, 2]
1 0	
2 1	
3 2	
4 3	