Contents

1 Word Swap **3**

2 Aligned Calender **5**

3 Underwater Trip **7**

4 Minion Walk **9**

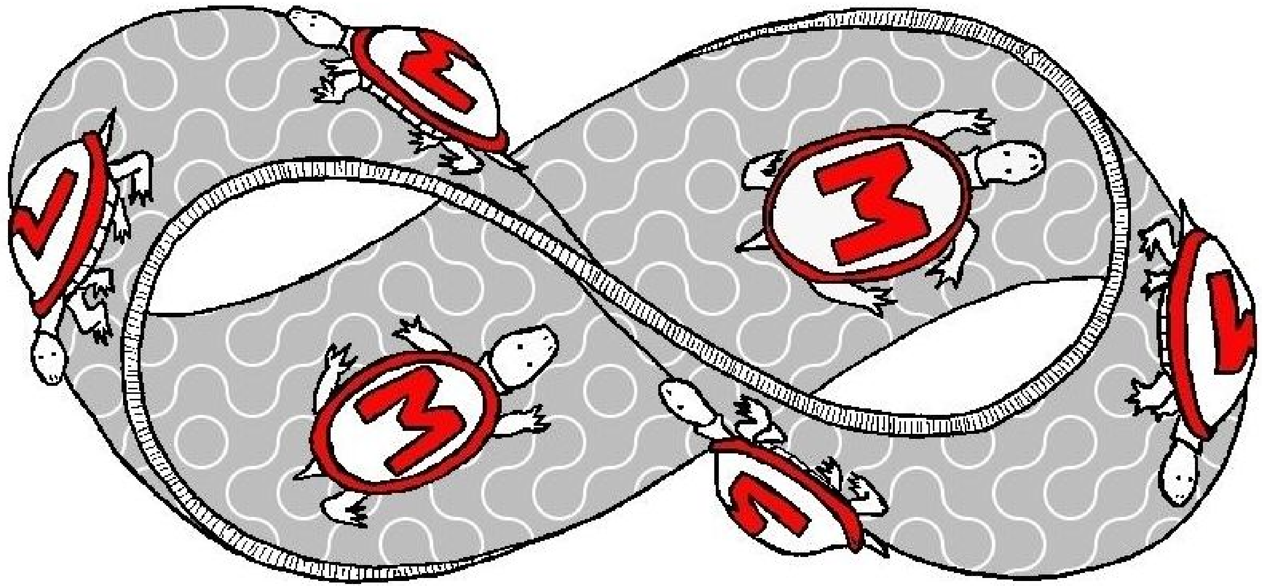
5 Word Ladder **11**

6 Crossing River **13**

7 Shark Tour **15**

8 Jam Factory **17**

9 The Minions Build a Brick Wall **19**



1 Word Swap

Agnes has heard that the carnival is going to have a new game next year. The carnival guy chooses a word and writes it down on a piece of paper, and only tells you the length of the word (lets call it Word 1). You have to guess a word of the same length (lets call it Word 2), and you earn or lose coins depending on how different are the two words (i.e., based on how much effort is required to swap the two words). Agnes, having just turned 6, has barely started reading and can't do math yet, so she wants your help to play this game.

The carnival provided the following rules for earning or paying coins.

- Only letters a-z will be used in the game.
- The difference between words is decided on a position-by-position basis.
- For each position: (1) if the characters in that position in the two words are the same, no coins are paid or earned; (2) if the character in Word 1 at that position appears alphabetically before the character in Word 2, then you have to pay 1 coin for each character between the two characters and 1 additional coin; (3) if the character in Word 1 at that position appears alphabetically after the character in Word 2, then you earn 1 coin for each character between the two characters and 1 additional coin.
- For example: if the carnival guy wrote down `agnes` and you guessed `heard`, then the following table shows the calculation.

	Position 1	**Position 2**	**Position 3**	**Position 4**	**Position 5**	**Total**
Carnival Word	a	g	n	e	s	
Your Word	h	e	a	r	d	
Coins	paid 7	earned 2	earned 13	paid 13	earned 15	earned 10

- You might think this is not particularly interesting game, since one can always guess words like "aal", "abut", etc. (depending on length), and easily earn coins. However, there is a twist: if it turns out you would earn too many coins, you don't earn anything (its a carnival game after all). We will ignore this twist.

You have to write a program that tells Agnes how many coins she will earn or have to pay for a pair of words. The program will read in a series of pairs of words and report the amount paid or earned. The provided skeleton handles the input of the words and the output messages. You need to implement the method costToSwap which returns an integer.

Input/Output Format:

Input: The first line in the test data file contains the number of test cases. After that each line will contain two words separated by a space.

Output: For each test case, the program will display the number of coins paid or earned.

Note: We have provided a skeleton program that reads the input and prints the output based on the costToSwap method that you need to implement.

```
private static int costToSwap(String word1, String word2)
```

If you have to pay, then the function should return a negative number whose absolute value is the number of coins you need to pay. If you earn coins, then it should return a positive number.

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	
agnes heard	Swapping letters to make agnes look like heard earned 10 coins.
unicorn minions	Swapping letters to make unicorn look like minions earned 1 coins.
victor vector	Swapping letters to make victor look like vector earned 4 coins.
sweat waste	Swapping letters to make sweat look like waste was FREE.

2 Aligned Calender

The Minions have found that El Macho and other super-villains use a different calendar than the rest of us. Their calendar has 13 months that each have 28 days (thus the 13 months cover a total of 364 days). The remaining 1 or 2 days of the year (depending on whether it is leap year or not) are used as a vacation before starting the next year of evil. The Minions need you to implement a method that will convert a date from the regular calendar into a date on the super-villain calendar.

The Minions have provided you with the following reminders:

- Non-leap years have the following days per month: 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31.
- A leap year has 29 days in the second month and is defined as any year that is evenly divisible by 4 as long as that year is NOT evenly divisible by 100 UNLESS that year is also evenly divisible by 400.

Your task is to write a program that will read in a series of dates from the standard calendar and convert them to the super-villain calendar and print the new date. The provided skeleton handles the input of the dates and the output messages. You need to implement the method **convertDate** which returns a **ConvertedDate** object.

Input/Output Format:

Input: The first line in the test data file contains the number of test cases (≤ 100). After that each line will contain one date from the regular calendar in "YYYY MM DD" format. You can assume that you will only be provided with correct dates.

Output: For each test case, the program should display the date from the regular calendar and the date as it would appear on the super-villain calendar. The exact format is shown below.

Note: We have provided a skeleton program that reads the input and prints the output. You need to implement the method **convertDate** which returns a **ConvertedDate** object.

```
private static ConvertedDate convertDate(int year, int month, int day)
```

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	
2014 01 20	2014/1/20 became 2014/1/20
2014 01 31	2014/1/31 became 2014/2/3
2014 03 30	2014/3/30 became 2014/4/5
2014 12 31	2014/12/31 became a HOLIDAY

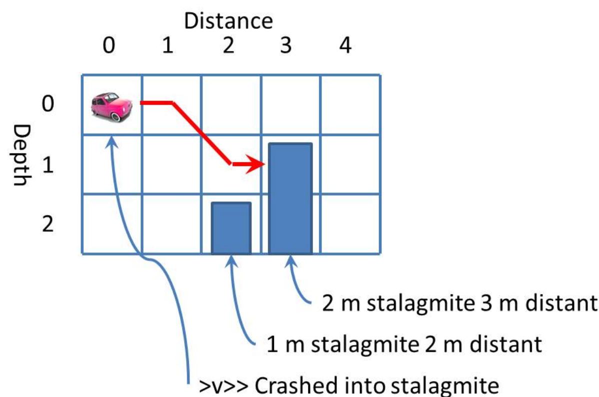
3 Underwater Trip

Lucy has kidnapped Gru and two minions in order to take them to the secret underwater headquarters of the Anti-Villain League. She has converted her car into a submarine for the last part of the trip. Along the way her car passes through an underwater tunnel.

Lucy deploys her car's sonar to detect the depth and length of the tunnel, as well as the size and distance of several stalagmites sticking up from the bottom of the tunnel. There is a strong current flowing through the tunnel that is carrying Lucy's car 1 meter forward each second. She can steer the car to either move up 1 meter, stay at the same depth, or move down 1 meter each second.

You are given several possible action sequences Lucy can follow for steering her car through the tunnel. Your goal is to decide for each sequence whether her car will hit the tunnel roof, tunnel wall, a stalagmite, or safely pass through the tunnel.

Lucy's car starts at the top left corner of the tunnel (distance = 0, depth = 0). It travels to the right, and is considered to have safely reached the end of the tunnel if it travels $n-1$ meters into a tunnel of length n. Each action sequence will thus consist of $n-1$ actions for a tunnel of length n.



In the example above, Lucy is trying to travel through a tunnel of depth 3 and length 5. There are two stalagmites. One is 1 meter high and 2 meters distant. The second is 2 meters high and 3 meters distant. When considering the action sequence >v>>, Lucy's car starts from the top left corner, moves 1 meter right, 1 meter down and to the right, then crashes into a stalagmite when it tries to move 1 meter right. If the second stalagmite was not present then the action sequence would have reached the end of the tunnel.

Input/Output Format:

Input: The first line in the test data file contains the number of test cases. Each test case begins with a description of the depth and length of the tunnel. After that is the number of stalagmites, followed by the size and distance of each stalagmite on a separate line.

Next is the number of action sequences to analyze, followed by each sequence on a separate line. Each sequence is a string composed of the characters ∧, >, and v, indicating whether Lucy should steer the car up (decrease depth by 1 meter), keep it at the same depth, or steer the car down (increase depth by 1 meter).

Output: For each test case, you are to output the input action sequence, followed by the expected outcome of following the sequence. There are four possible outcomes:

- Reached end of tunnel
- Crashed into stalagmite
- Crashed into tunnel floor
- Crashed into tunnel ceiling

The exact desired input/output format is shown below in the examples.

Note: We have provided a skeleton program that reads the input. You need to implement the body of the following procedure to print out the correct outcome for each input action sequence:

```
private static void tunnelTrip(
    int tunnelDepth,
    int tunnelLength,
    int stalagmites[][],
    char[][] actions)
```

Examples:

Input:

```
2
Tunnel depth 3 length 5
     0 stalagmites
     4 sequences
           >>>>
           >v>>
           >v∧∧
           v>vv
Tunnel depth 3 length 5
     2 stalagmites
           1 meter stalagmite 2 meters distant
           2 meter stalagmite 3 meters distant
     3 sequences
           >>>>
           >v>>
           v>vv
```

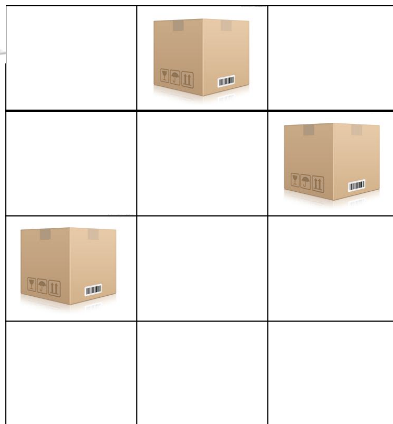
Output:

```
Case:  1
Sequence >>>> Reached end of tunnel
Sequence >v>> Reached end of tunnel
Sequence >v∧∧ Crashed into tunnel ceiling
Sequence v>vv Crashed into tunnel floor
Case:  2
Sequence >>>> Reached end of tunnel
Sequence >v>> Crashed into stalagmite
Sequence v>vv Crashed into stalagmite
```

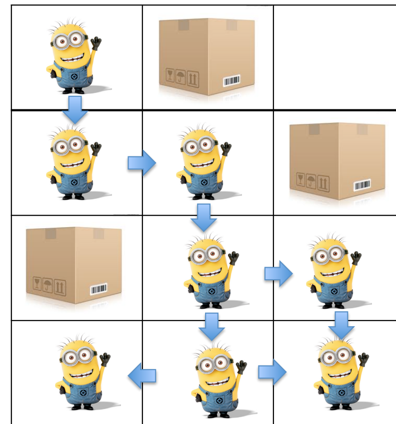
4 Minion Walk

Doctor Nefario is leaving Gru because he misses being evil. While he is packing and preparing to leave, boxes become scattered all over the laboratory floor. In fact, there are so many boxes that minions may no longer be able to cross the room!

Your goal is to determine which parts of the lab minions can still reach. In particular you want to find whether minions can cross the room from the top left corner to the bottom right corner. Minions can only make 90 degree turns. I.e., they may move up, down, left, or right one location at a time, but not diagonally.



Layout



Locations that minions can reach

In the example above, minions need to cross a room of height 4 and width 3. Three boxes are scattered in the room. Minions start from the top left corner and try to reach the bottom right corner, making only 90 degree turns. It turns out that minions can reach every part of the room except the top right corner.

Input/Output Format:

Input: The first line of the input file contains the number of test cases. Each case has the following format. First, a line containing the height H and width W of the room. Next, a sequence of H lines, each of containing W characters. Each character is either '`X`' (occupied) or '`O`' (clear).

Output: The output is the layout of the room for each test case, where locations are marked as '`M`' if it can be reached by minions, followed by one of the the following lines: "Minions can cross the room" or "Minions cannot cross the room".

Note: We will provide a procedure for reading the input and setting it up as a two dimensional array of characters, with H rows and W columns (where $1 \leq H, W \leq 100$). The entry '`X`' (upper-case 'X') means that there is box at the location, and '`O`' (upper-case 'O') means that the location is clear.

All you need to provide is the body of the following function, which takes the input from the character array `wall[H][W]`, and stores the result by modifying this array.

```
private static boolean canReach(char[][] wall)
```

Starting from the top left corner, each location that can be reached by a minion making only 90 degree turns should be marked with ‘M’. The function should return true if minions can start in the top left corner of the room and reach the bottom right corner of the room, and false otherwise.

You may assume that the input is valid and H and W each lie between 1 and 100.

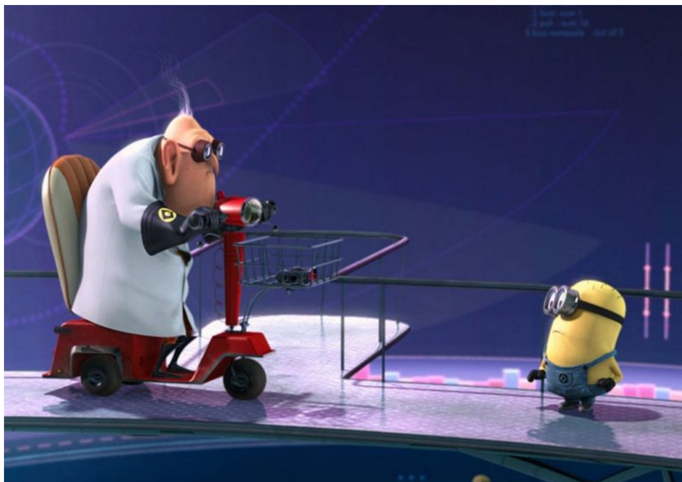
Examples:

Input:

```
2
4 3
OXO
OOX
XOO
OOO
4 3
OOO
OOX
OXO
XOO
```

Output:

```
Case:  1
+---+---+---+
| M | X |   |
+---+---+---+
| M | M | X |
+---+---+---+
| X | M | M |
+---+---+---+
| M | M | M |
+---+---+---+
Minions can cross the room
Case:  2
+---+---+---+
| M | M | M |
+---+---+---+
| M | M | X |
+---+---+---+
| M | X |   |
+---+---+---+
| X |   |   |
+---+---+---+
Minions cannot cross the room
```



5 Word Ladder

As minions continue to learn English, they start playing various games to improve their vocabulary. One of the games they play is *word ladder*. Word ladder (also known as Doublets, word-links, or Word golf) is a word game invented by Lewis Carroll. A word ladder puzzle begins with two words, and to solve the puzzle one must find a chain of other words to link the two, in which two adjacent words (that is, words in successive steps) differ by one letter.

For example, given words COLD and WARM, here is a word ladder between the two:

COLD → CORD → CARD → WARD → WARM

Another solution is:

COLD → WOLD → WORD → WARD → WARM

Whichever player finds the shortest such word ladder between the given two words (if one exists), wins the game. If two players find different shortest word ladders, we look at the first word in the word ladder, and if they are different, the player who uses the word that comes earlier in lexicographical order wins. For example, comparing the above two word ladders, the player who plays the first word ladder wins (since CORD comes before WOLD in lexicographical order). If the two ladders use the same first word, then we look at the second word, and so on.

Your task is to help Kevin the Minion win at this game.

Input/Output Format:

Input: The first line in the test data file contains the number of test cases (< 10). Each test case contains one dictionary, and pairs of words for which you have to find a word ladder using only the words in the dictionary. Each test case begins with the number of words in the dictionary, followed by the list of the words in the dictionary (e.g., in the first example below, the dictionary contains 6 words). The next number is the number of pairs of words for which you have to find the word ladder using that dictionary, followed by the pairs themselves. You can assume that the words are provided in all lowercase letters (i.e., "gru", not "Gru").

Output: For each test case, for each pair of words provided, you have to find a word ladder using only the words in that dictionary. You are to output the word ladder itself. If there are multiple word ladders between two words, you are to output the one containing fewest words; further, if there are multiple shortest word ladders, you are to output the one that comes lexicographically first. The exact format is shown below in the examples.

As an example: in the second test case below, for the pair of words `man` and `how`, there are several shortest word ladders (e.g., man → can → con → cow → how, is another word ladder that only uses 3 words). Since "ban" comes lexicographically before "can", you should output the one that uses "ban", "bon", "bow" (as 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<String> solveWordLadder(String[] dict, String start, String end)
```

Your code should return a Vector of Strings that form the word ladder (set to null if there is no word ladder between the two words). The first entry in the return vector must be the "start" word and the last entry must be the "end" word. As an example, for the second test case below, you should return a Vector of Strings containing "man", "ban", "bon", "bow", and "how", in that order.

Examples:

```
Input:
2
6 abc abd acd bcd bdd xyz 2 abc bdd abc xyz
10 man can con bow cow mow han ban bon how 1 man how
```

```
Output:
Word ladder from abc to bdd:  abc --> abd --> acd --> bcd --> bdd
No word ladder from abc to xyz using the input dictionary.
Word ladder from man to how:  man --> ban --> bon --> bow --> how
```

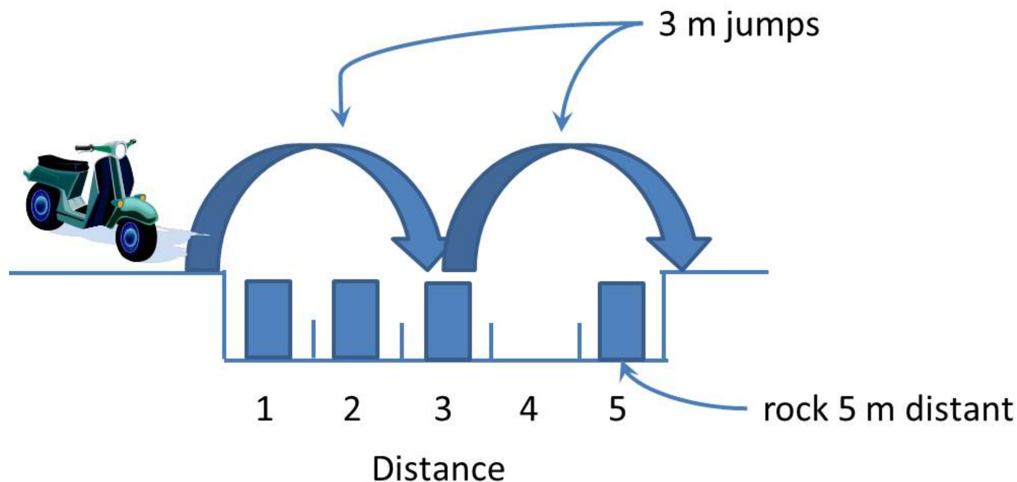
6 Crossing River

Doctor Nefario is planning a trip after spending a lot of time doing research in his lab. During the trip he needs to cross a river. Fortunately there are N rocks in the river lying on a straight line across the river that may be used to make the crossing. The width of the river (i.e., the total distance that needs to be crossed) is L.

Doctor Nefario's scooter can hover in the air, but to make the river crossing he wants to build a rocket booster that can make longer jumps. The cost of a rocket booster depends on its quality. More precisely, a rocket booster that can boost jumps up to distance R costs R^2. The booster may be used multiple times, but each jump costs an additional amount C.

For example, if Doctor Nefario builds a rocket booster that can boost jumps up to distance 10 and uses it to make 5 jumps, the total cost would be $10^2 + 5 \times C = 100 + 5C$.

Given the width of the river L, the cost of each jump C, and the location of N rocks, your goal is to find the minimum cost M for Doctor Nefario to cross the river, the number of jumps J needed, and the range R selected for the rocket booster.



For instance, in the figure above Doctor Nefario needs to cross a river of width 6. There are 4 rocks at distances 1, 2, 3, and 5. One possible solution is to build a rocket booster with range 3, which would allow the scooter to cross the river with two jumps.

The parameters of this problem may be quite large: $1 \leq L \leq 10^9$, $0 \leq C \leq 10^6$, and $0 \leq N < 1000$. Your implementation should represent these values using *long* instead of *int*. Exhaustively checking all possible values for range R will not work in the time allowed.

Input/Output Format:

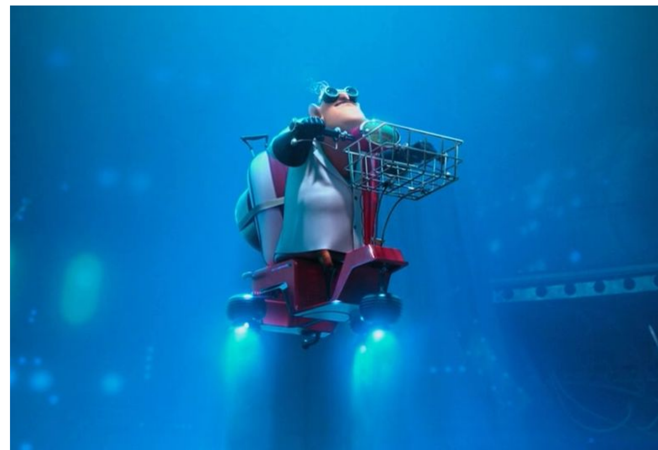
Input: The input consists of multiple test cases. The first line of each test contains three integers L, C and, N: the width of the river, cost of each jump, and the number of rocks, respectively. Each of the next N lines contains an integer specifying the distance of a rock. The input terminates with a line containing just 0.

Output: For each test case, print the minimum cost of crossing the river in the following format:

Minimum cost M achieved with J jumps of range R

Examples:

Input:	Output:
6 2 4	Minimum cost 12 achieved with 4 jumps of range 2
1	Minimum cost 49 achieved with 2 jumps of range 3
2	
3	
5	
6 20 4	
1	
2	
3	
5	
0	



7 Shark Tour

The minions who rode in Lucy's car for an underwater trip were very impressed by the many sharks they saw on the trip. When they got home they told the other minions. Now everyone wants to go back and see the sharks!

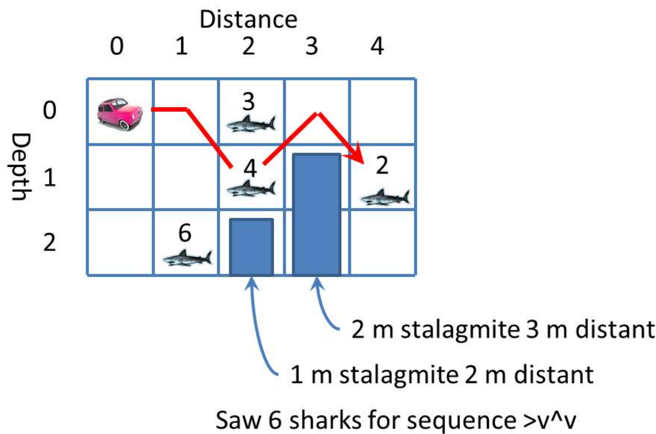
As before, Lucy's car's sonar can detect the depth and length of the tunnel, as well as the size and distance of several stalagmites sticking up from the bottom of the tunnel. For the shark tour Lucy has modified her car's sonar to also report how many sharks can be seen from each tunnel location. Because of viewing conditions, any shark can only be seen from exactly one location in the tunnel.

There is a strong current flowing through the tunnel that is carrying Lucy's car 1 meter forward each second. She can steer the car to either move up 1 meter, stay at the same depth, or move down 1 meter each second.

Your goal is to analyze all possible paths Lucy can take through the tunnel (while avoiding tunnel ceiling, floor, and stalagmites), and select the action sequence for the path that will allow the greatest number of shark sightings. Assume there always exists at least one path through the tunnel.

Lucy's car starts at the top left corner of the tunnel (distance = 0, depth = 0). It travels to the right, and is considered to have safely reached the end of the tunnel if it travels $n - 1$ meters into a tunnel of length n.

A path through the tunnel will consist of $n - 1$ actions for a tunnel of length n. Each action is represented by one of the characters $\wedge$, >, and v, indicating whether Lucy should steer the car up (decrease depth by 1 meter), keep it at the same depth, or steer the car down (increase depth by 1 meter) as it travels through the tunnel.



In the example above, Lucy is trying to travel through a tunnel of depth 3 and length 5. There are two stalagmites. One is 1 meter high and 2 meters distant. The second is 2 meters high and 3 meters distant. There are 4 shark sightings at 2 meters distant (0 and 1 meter down), 4 meters distant (1 meter down). and 1 meter distant (2 meters down).

From the location 2 meters distant and 1 meter down, 4 sharks may be seen. From the position 4 meters distant and 1 meter down, 2 sharks may be seen. The action sequence >v$\wedge$> takes Lucy's car through the tunnel along a path that passes through both of these locations, allowing a total of 6 sharks to be seen.

The parameters of this problem may be quite large. The tunnel may be long, with several thousand shark sightings. Exhaustively checking all possible paths through the tunnel will not work within the time allowed.

Input/Output Format:

Input: The first line in the test data file contains the number of test cases. Each test case begins with a description of the depth and length of the tunnel. After that is the number of stalagmites, followed by the size and distance of each stalagmite on a separate line. Next is the number of shark sightings, followed by the number and location of each shark sighting on a separate line.

Output: For each test case, you are to output the maximum number of sharks possibly seen, and the action sequence required to see them using the following format:

"Saw n sharks for sequence *actions*"

The exact desired input/output format is shown below in the examples.

Note: We have provided a skeleton program that reads the input. You need to implement the body of the following procedure to find the action sequence that leads to the maximal number of shark sightings:

```
private static void findPath(
    int tunnelDepth,
    int tunnelLength,
    int stalagmites[][],
    int sightings[][])
```

Examples:

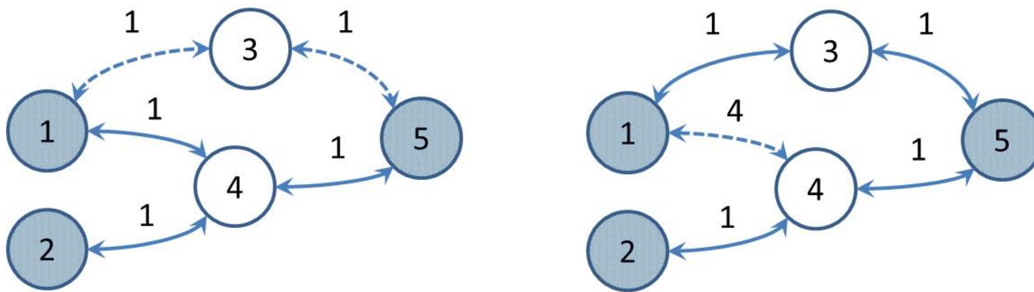
Input:	Output:

2
Tunnel depth 3 length 5
     0 stalagmites
     1 sightings
           10 sharks 2 meters distant 1 meters down
Tunnel depth 3 length 5
     2 stalagmites
           1 meter stalagmite 2 meters distant
           2 meter stalagmite 3 meters distant
     4 sightings
           4 sharks 2 meters distant 1 meters down
           3 sharks 2 meters distant 0 meters down
           2 sharks 4 meters distant 1 meters down
           6 sharks 1 meters distant 2 meters down
	Case: 1
	Saw 10 sharks for sequence >v>>
	Case: 2
	Saw 6 sharks for sequence >v∧v

8 Jam Factory

Gru is making jam in his underground laboratory in an attempt to start a new legitimate career. Minions crush fruit in a number of giant vats. To make jam, these vats must be connected together by large pipes. After the failure of his last batch of jam, Gru wants to try making something new by combining two types of fruit.

You are given the costs of connecting pairs of vats with pipes. Your goal is to find the least expensive way to connect together two vats v_1, v_2 with the vat v_d leading to the bottling machine. Fruit from both vats v_1, v_2 must be able to reach vat v_d. They may share part of the piping, though it is not required. In some cases it may not be possible to connect both vats to the vat leading to the bottling machine.



The figure above shows two examples, each example contains 5 vats and 5 possible pipes for connecting the vats. Vats 1 and 2 need to be connected to vat 5, which leads to the bottling machine. In the example on the left, the cost of each pipe is 1, so the minimal cost of connecting vats 1 and 2 to vat 5 is 3, achieved by building pipes from 1 to 4, 2 to 4, and from 4 to 5 (as shown by the solid lines).

In the example on the right, the cost of a pipe between vats 1 and 4 has been increased to 4. Now the minimal cost of connecting vats 1 and 2 to vat 5 is 4, achieved by building pipes from 1 to 3, 3 to 5, 2 to 4, and 4 to 5 (as shown by the solid lines).

The parameters of this problem may be quite large. There may be several hundred vats with several thousand possible pipes. Exhaustively checking all possible connections between vats will not work within the time allowed.

Input/Output Format:

Input: Input consists of multiple test cases. Each test case begins with a single line consisting of five numbers: the number of vats v, the number of possible pipes connecting vats p, the numbers of the two vats to be connected (v_1 and v_2), and the number of the vat connected to the bottling machine v_d.

There are then p additional lines consisting of three numbers: the two vats (v_x and v_y), and the cost of connecting them with piping. Following the last test case is a line containing only 0.

Output: For each test case, print the lowest cost on as single line in the following format:

Cost of connecting v_1 and v_2 to v_d is c

If either vat cannot be connected to the vat leading to the bottling machine, print out the following line:

Cannot connect v_1 and v_2 to v_d

Examples:

Input:	Output:

`5 5 1 2 5`
`1 3 1`
`1 4 1`
`2 4 1`
`3 5 1`
`4 5 1`
`5 5 1 2 5`
`1 3 1`
`1 4 4`
`2 4 1`
`3 5 1`
`4 5 1`
`0`
	`Cost of connecting 1 and 2 to 5 is 3`
	`Cost of connecting 1 and 2 to 5 is 4`

9 The Minions Build a Brick Wall

Because of major damage to his laboratory (from an explosion involving fart-generating weaponry), Gru needs his minions to replace a large brick wall in their lab. The wall is of size $H \times W$, that is, H squares high and W squares wide (both positive integers). Each brick is of size 1×2 and may either horizontally oriented (1×2) or vertically oriented (2×1). The brick-laying process is complicated by the fact that there are certain squares, called *forbidden squares*, that must *not* be bricked over. The minions' job is to determine a pattern of bricks in order to cover as many of the non-forbidden squares as possible. (Fig. 1(a) shows an example of a wall with forbidden squares shaded, and Fig. 1(b) shows one possible solution for this input.)

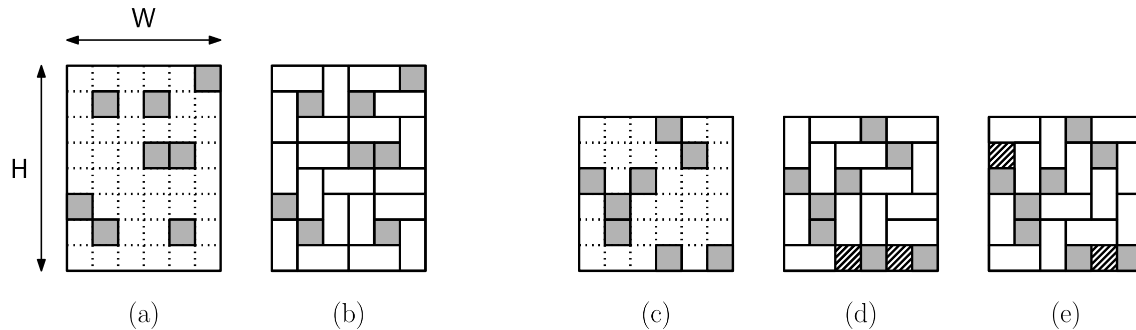


Figure 1: Sample input walls ((a) and (c)) with forbidden squares shaded, and possible brick layouts ((b), (d), and (e)) with uncovered squares shown with hatched lines.

Note that it may not always be possible to cover all the non-forbidden squares. (You can convince yourself that no pattern of bricks can cover all the squares in Fig. 1(c). In Fig. 1(d) and (e) we show two optimal solutions, each of which leaves two squares uncovered.)

The minions are given as input a two dimensional array of characters, wall, with H rows and W columns (where $1 \leq H, W \leq 100$). The entry 'X' (upper-case 'X') means that the square is forbidden and 'O' (upper-case 'O') means that the square can be covered.

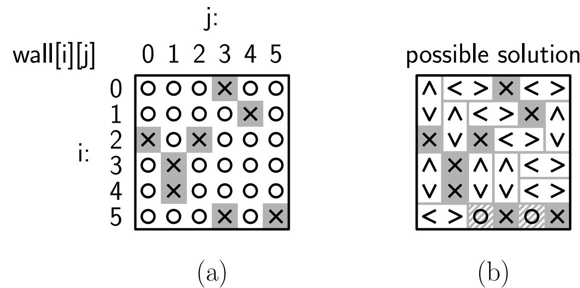


Figure 2: (a): Sample input array corresponding Fig. 1(c). (b): sample output array corresponding to the solution of Fig. 1(d).

The output is written into the same array. Each horizontally oriented brick is indicated with two horizontally adjacent array entries '<' and '>'. Each vertically oriented brick is indicated with two vertically adjacent array entries '^' (a caret or Shift-6 on most keyboards) and 'v' (lower-case 'v'). The output is considered correct if it has the proper format and covers the maximum possible number of non-forbidden squares.

Input/Output Format:

Input: The first line of the input file contains the number of test cases. Each case has the following format:

- A line containing the height H and width W of the wall array.
- A sequence of H lines, each of containing W characters. Each character is either '`X`' (forbidden) or '`O`' (coverable).

We will provide a procedure for reading the input and setting it up as an array. All you have to provide is the body of the following function, which takes the input from the character array `wall[H][W]`, and stores the result by modifying this array.

```
private static void layBricks(char[][] wall)
```

You may assume that the input is valid and H and W each lie between 1 and 100.

Output:

You do not need to generate any output. We will provide a main program that will check that the array that your function returns is of the proper form (although it does not check for optimality), and if so, it will print the tiling layout. Here is an example corresponding to Fig. 2(a) and (b).

Input: (One of possibly many test cases.)

```
6 6
OOOXOO
OOOOXO
XOXOOO
OXOOOO
OXOOOO
OOOXOX
```

Output: (Generated by our program based on your modified array.)

```
Solution:
+---+---+---+---+---+---+
|   |       | X |       |
+   +---+---+---+---+---+
|   |   |       | X |   |
+---+   +---+---+---+   +
| X |   | X |       |   |
+---+---+---+---+---+---+
|   | X |   |   |       |
+   +---+   +   +---+---+
|   | X |   |   |       |
+---+---+---+---+---+---+
|       | O | X | O | X |
+---+---+---+---+---+---+

Uncovered squares remaining = 2
```

Practice 1 – Longest Subsequence

Gru and Lucy are trying to break into El Macho's lair. They arrive at a hallway that they need to walk across, which is likely booby-trapped. The hallway is a single line of squares, each of which is annotated either X or O. Gru believes that the only "safe" squares are the ones that comprise the longest subsequence of X's, i.e., the longest contiguous set of squares that are all annotated X.

Your goal is to help Gru find the safe squares to jump on.

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 number of entries in the sequence, n (provided as an `int`), and the next n Strings (which can only be "X" or "O") are the sequence itself.

Output: For each test case, you are to find the longest subsequence of consecutive X's, and output the number of X's in that sequence. 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 solveLongestSubsequence(ArrayList<String> sequence)
```

The procedure should return the number of O's in the longest contiguous subsequence of O's.

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	
10 X O O O O X O O O X	The longest contiguous subsequence of X's is of length 1
5 O O O O O	The longest contiguous subsequence of X's is of length 0
4 X X X X	The longest contiguous subsequence of X's is of length 4

Practice 2 – Count Vowels

Gru has been trying to teach the minions English (they are born knowing only Minionese, which Lucy can't stand). He is currently stuck trying to teach them the difference between vowels and consonants, which the minions are always confused about, especially "y", which they argue about endlessly (it is not a vowel). To help them practice, Gru starts a contest where the minions have to count the number of vowels in a word, i.e., given a word, they have to tell Gru what is the number of vowels in it. The winner gets ice cream, which the minions are crazy about. Kevin the Minion wants your help to cheat and eat a lot of ice cream.

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, a word, w, (provided as a `String`). You can assume that the words are provided in all lowercase letters (i.e., "gru", not "Gru").

Output: For each test case, you are to output the number of vowels in that word. The vowels are: "a", "e", "i", "o", and "u". 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 countVowels(String s)
```

Examples:

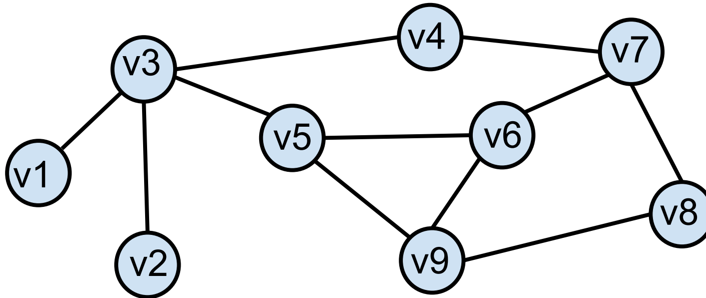
The output is shown with an extra blank line so that each test case input is aligned with the output; the first blank line should not be present in the actual output.

Input:	Output:
3	
despicable	The number of vowels in despicable is 4.
gru	The number of vowels in gru is 1.
nth	The number of vowels in nth is 0.

Practice 3 – Neighborhoods in Graphs

Gru and Lucy put together a graph of super-villains to see if they can see any interesting connections, in order to find who stole the Arctic Circle Laboratory. In this graph, there is a node (vertex) for every super-villain, and there is an edge (connection) between two nodes if they have ever concocted a sinister plot together. The figure below shows an example of such a graph over 10 villains (to simplify things for you, we will use made-up simple names for the super-villains, of the form $v1$, $v2$, ..., and so on).

Given such a graph and a super-villain, your goal is to count to the number of super-villains in that super-villain's 2-hop neighborhood in the graph, i.e., all the nodes that are within a distance of 2 from the given node. (This is also called "friends of friends query" in a social graph setting.) For example, for $v1$, this includes: $v3$, $v2$, $v5$, and $v4$ (so the answer would be 4), whereas for $v5$, the 2-hop neighborhood includes: $v1, v2, v3, v4, v6, v7, v8, v9$, i.e., all the nodes in the graph (so the answer would be 8). Note that, we don't count $v5$ to be in its own 2-hop neighborhood.



Input/Output Format:

Input: The first line in the test data file contains the number of test cases (< 100). After that each line contains a test case. The first two numbers in each test case represent the number of supervillains in the graph, n ($n < 100$), and the number of edges in the graph, e ($e < 1000$). After that, the next $2e$ Strings describe the edges. The last String on each line contains the id of a supervillain, vx.

Output: For each test case, you are to output the number of nodes in the 2-hop neighborhood of vx. 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 solveGraphs(Arraylist<Edge> edges, String vx)
```

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

Examples:

```
Input:
4
9 11 v1 v3 v2 v3 v3 v4 v3 v5 v4 v7 v5 v6 v5 v9 v6 v7 v6 v9 v7 v8 v8 v9 v1
9 11 v1 v3 v2 v3 v3 v4 v3 v5 v4 v7 v5 v6 v5 v9 v6 v7 v6 v9 v7 v8 v8 v9 v5
5 4 v1 v2 v2 v3 v3 v4 v4 v5 v1
5 0 v1
```

```
Output:
The number of supervillains in 2-hop neighborhood of v1 is 4.
The number of supervillains in 2-hop neighborhood of v5 is 8.
The number of supervillains in 2-hop neighborhood of v1 is 2.
The number of supervillains in 2-hop neighborhood of v1 is 0.
```