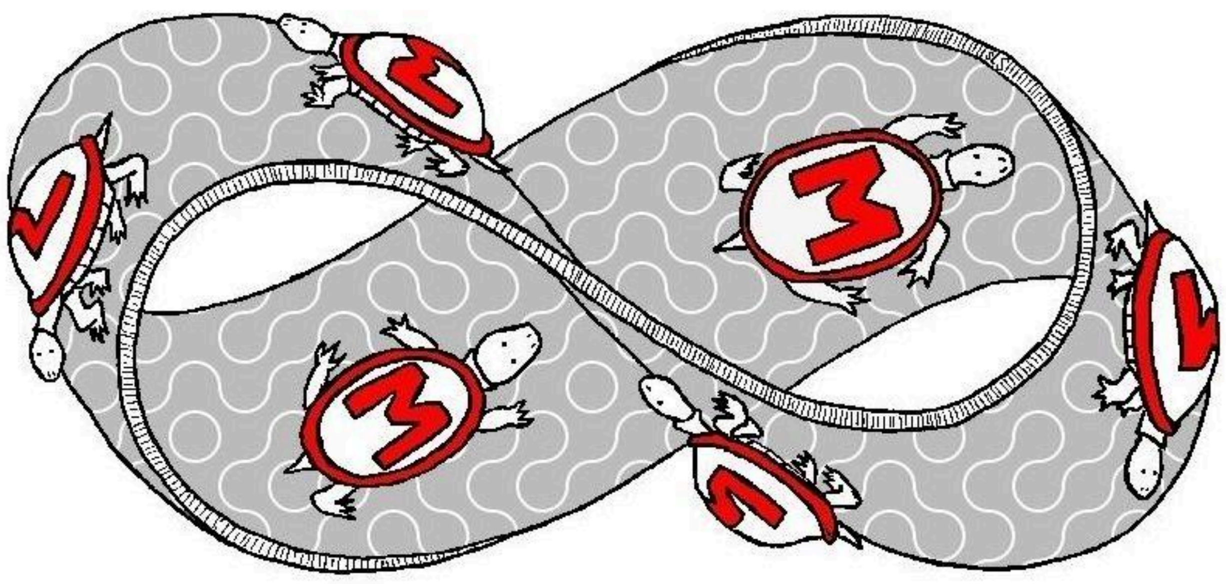
2024 University of Maryland High School Programming Contest



Problem 1: Koopa Rescue Mission

The Mushroom Kingdom is in peril! Princess Peach has been kidnapped by the Koopa King, and it's up to you, the brave programmer, to help Mario and Luigi rescue her. Your mission is to create a program that can navigate through the Koopa King's message and uncover crucial clues for the rescue.

The Koopa King has left a cryptic message, and your task is to analyze it. Write a method that takes in a string of characters and follows these rules:

1. If "Peach" appears in the string, the method should return the number of times "Koopa" appears within the message. This information will reveal the strength of the Koopa King's forces. Note that both of these matches should be “case-sensitive”.

2. If "Peach" is absent from the string, the method should return -1, indicating that the message is a diversion and not a reliable source of information.

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

```
   static int lookForKoopa(String str)
```

which returns an

```
   int
```

as described above.

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 string, consisting of characters `a-zA-Z`, and of length between 1 and 100 (inclusive).

Output:

For each test case, the program should output “-1” (if “Peach” is not present in the string) or the number of times “Koopa” appears in the string.

Note:

We have provided a skeleton program that reads the number of test cases and the inputs one by one. It also prints the output based on the `lookForKoopa()` method that you need to implement.

Examples:

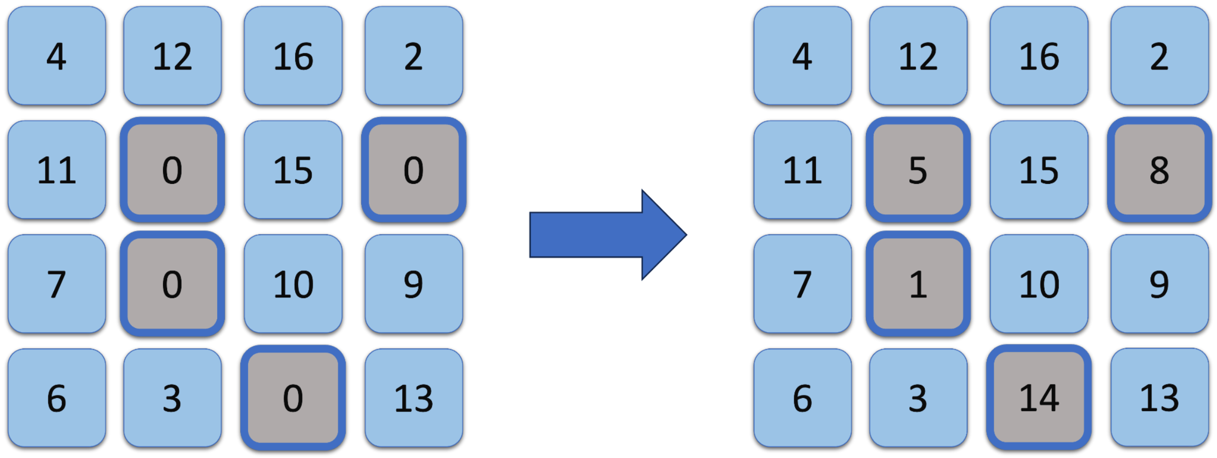
The output is shown with extra blank lines so that the output is shown aligned with the appropriate input line; those blank lines should not be present in the actual output. (The provided skeleton code handles that for you).

Input:	**Output:**
`5`	
`Koopa`	`Result is -1`
`KoopaKoopaPeach`	`Result is 2`
`PeachKoopaMario`	`Result is 1`
`LuigiKoolPeach`	`Result is 0`
`LuigiKoolPeachKoopa`	`Result is 1`

Problem 2: Mario's Magic Matrix

Mario’s latest escapade has led him to a strange part of the Mushroom Kingdom where the blocks he loves to smash for coins are arranged in a peculiar grid. These are no ordinary blocks—they are enchanted with numbers, from 1 to 16, each bestowing different powers and coin values. The grid is a 4x4 matrix, and the enchantment is such that after Mario smashes all the 16 blocks, the coins he receives are the product of the numbers on the blocks in each row plus each column. However, exactly four of the blocks had their numbers erased, and it looks like Mario can fill those in any order he wants (with the missing numbers). Your task is to help Mario maximize his haul.

As an example, on the left below, you can see the original matrix as Mario encountered it, where the erased blocks have 0 written on them. The right matrix shows the best way to complete the matrix so that the coin haul is maximized.



In this case, the maximum coin haul is the sum of:

4 * 12 * 16 * 2 + 11 * 5 * 15 * 8 + 7 * 1 * 10 * 9 + 6 * 3 * 14 * 13 (row-wise product sum)

4 * 11 * 7 * 6 + 12 * 5 * 1 * 3 + 16 * 15 * 10 * 14 + 2 * 8 * 9 * 13 (column-wise product sum)

Giving a total of: 49542 coins.

You need to implement the function:

```
        static int reconstructMatrix(int[][] matrix)
```

This function will receive 16 integers in the form of a 4x4 array (exactly 4 of which will be 0, indicating the erased blocks) and should reconstruct the matrix optimally so that Mario's coin haul is maximized. The function should return an:

```
int
```

which is the maximum coin haul that's possible by filling in the erased blocks.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases. After that, each test case is provided on the next 4 lines containing 4 integers each.

Output:

For each test case, your program should output the maximum coin haul after reconstructing the grid.

Note:

We have provided a skeleton program that reads the number of test cases and the inputs one by one. It also prints the output based on the `reconstructMatrix()` 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).

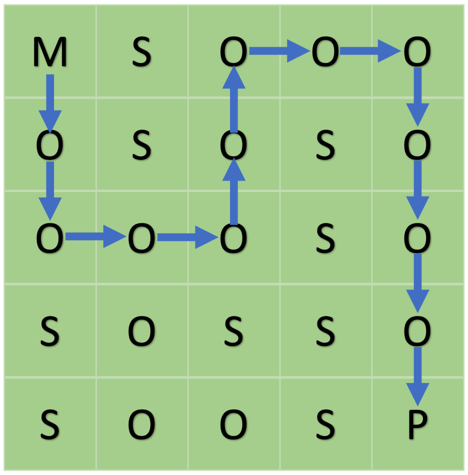
Input:	**Output:**
`2`	
`4  12 16 2`	`The maximum number of coins that can be`
`11 0  15 0`	`collected is: 49542`
`7  0  10 9`	
`6  3  0  13`	
`3  0 10 8`	`The maximum number of coins that can be`
`15 2  0 4`	`collected is: 31458`
`6  0 7  11`	
`9  5  0 1`	

Problem 3: Mario's Perilous Path

Bowser has once again captured Princess Peach, and this time, he has trapped her within a treacherous labyrinth! Mario must navigate through a maze of obstacles and spikes to reach his beloved Princess. Your programming skills are needed to guide Mario through the shortest path to rescue her.

The labyrinth is provided in the form of a 2-dimensional character array representing its layout. Each cell in the array has a specific meaning:

- 'M': Mario's starting position
- 'P': Princess Peach's location
- 'O': Unobstructed path
- 'S': Spikes (obstacles Mario cannot pass through)



Your method should return the minimum number of steps Mario needs to take to reach Princess Peach, adhering to these rules:

- Mario can only move up, down, left, or right.
- He cannot move through spikes or step outside the grid's boundaries.

If there is no path that satisfies the constraint, then your method should return -1.

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

```
static int findPath(char[][] grid)
```

which returns an

```
int
```

as described above.

<u>Input/Output Format</u>

Input:

The first line in the test data file contains the number of test cases. After that, for each test case, the next line specifies the size of the grid (*m n)*, followed by *n* lines of *m* characters each (the characters must be ‘M’, ‘P’, ‘O’, or ‘S’).

Output:

For each test case, the program should output the length of the path for Mario following the rules described above, or say that there is no such path.

Note:

We have provided a skeleton program that reads the number of test cases and the inputs one by one. It also prints the output based on the `findPath()` 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`	
`5 5`	`Path length is 8`
`MOOOO`	
`OOOOO`	
`OOOOO`	
`OOOOO`	
`OOOOP`	
`5 5`	`Path length is 12`
`MSOOO`	
`OSOSO`	
`OOOSO`	
`SOSSO`	
`SOOSP`	
`5 5`	`Path is blocked.`
`MSOOO`	
`OSOSO`	
`OSOSO`	
`SOSSO`	
`SOOSP`	

Problem 4: Princess Peach's Garden Party

In the heart of the Mushroom Kingdom, Princess Peach decides to host a delightful garden party at her castle. To add a touch of magic to the event, she invites a wise old wizard named Ramsey, known throughout the land for his intriguing prophecies and conjectures. Wizard Ramsey shares with the guests his most famous conjecture: "In any gathering of nine in the Mushroom Kingdom, there will always be either four who have gone on a quest with each other, or three who have never done any quests with each other." This conjecture has piqued the curiosity of the kingdom for years and has become a popular topic among the residents.

To make the party more interesting, Princess Peach and her eight guests – including various inhabitants of the kingdom – decide to test this conjecture. They aim to see if among them, they can find either four people who have done quests with each other (i.e., each pair among them has done a quest together), or three people who have never done a quest with each other.

For instance, assuming the participants are *A, B, C, …, I,* and we have that following pairs of folks have done quests together:

(A, B), (A, C), (A, E), (A, I), (B, C), (B, I), (C, I)

Then, we have a group of 4 (A, B, C, and I) who have all done quests with each other, as well as several triplets (e.g., (C, D, G)) such that no two of them have done a quest together.

Specifically, your goal is to write a function that takes as input the pairs as above, and return a 4-set of people who have gone on quests together, and a 3-set of people who haven't. If either does not exist, you should return **null** for that output. Remember that, Wizard Ramsey claims you will always find at least one of those (and in many cases both), so at most one of those should be set to **null**. The sets that you return should be ordered (i.e., your method should return the 4-set above in the order A, B, C, I, and not, e.g., B, C, A, I).

If there are multiple groups of 3 or 4 participants that satisfy the respective condition, you should return the first in lexicographical order. For instance, in the above example, there are a large number of triples that satisfy the second condition – your method should return (A, D, F).

Your function should have the signature:

```
    private static Solution solveRamsey(String[] pairs)
```

which takes in input as a string of pairs (of the form: "AB", "AC", …), and return an:

```
    Solution
```

which is a class that we have provided for returning the 4-set and the 3-set (these should be set to null as required).

Input/Output Format

Input:

The first line in the test data file contains the number of test cases, *n*. After that, each line contains one test case, starting with the number of pairs, *M*, and then the pairs themselves as shown below.

Output:

For each test case, you are to output the 4-set and the 3-set that you found (or report that there are none).

Note:

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

Examples:

The input/output are 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:**
2	
7 AB AC AE AI BC BI CI	These 4 people have gone on quests with each other : A B C I These 3 people have not gone on quests with each other : A D F
19 AE AF AH BC BD BE BG BH CD CF CI DF DI EF EG EI FH FI HI	These 4 people have gone on quests with each other : C D F I These 3 people have not gone on quests with each other : A B I

Problem 5: Maximizing Harmony

Boo has created blockades that block paths throughout the Mushroom Kingdom, making it hard to get around. Each blockade has a different "strength", making some easier or harder to break than others. The Kingdom in this problem can be modeled as an undirected tree on the vertices [0, N-1] (see the end of this problem for an example tree), where the strengths correspond to edge weights.

When a blockade is broken, all vertices on either side of the blockade that were previously blocked by this blockade (but not by any other blockade) magically gain 1 unit of harmony due to the removal of the blockade!

Mario decides to use a greedy strategy to break the blockades. His strategy works by breaking the blockades in order of increasing strength (starting from the lowest strength blockade). Your task is to help Mario determine the maximum amount of harmony obtained by any vertex as Mario applies his greedy strategy.

More formally, here are the rules:

- The Mushroom Kingdom is modeled as an undirected tree.
- Each edge of the tree initially has a blockade on it, with a *unique integer strength*.
- Mario destroys the blockades in order of increasing strength.
- When a blockade on an edge (u,v) is destroyed, all vertices that are currently reachable from either u or v *without passing through any other not-yet-removed blockade* have their harmony increased by 1.

The task is to determine the maximum harmony obtained by any vertex.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases (*N*). After that, each test case contains the following: an integer representing a random seed used for generating the input, followed by an integer representing the number of vertices in the tree for this test case.

Output:

One line per test case printing out the maximum harmony obtained by any vertex.

Note:

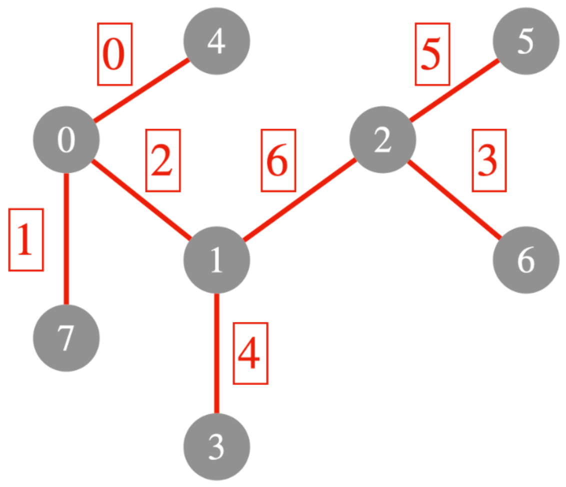
We have provided a skeleton program that reads the number of test cases, and parses a test case. It generates the input tree representing the kingdom and passes the tree to a function that you need to implement called `maxHarmony(int numVertices, ArrayList<Edge> edges)`.It also prints the output based on the `maxHarmony()` method.

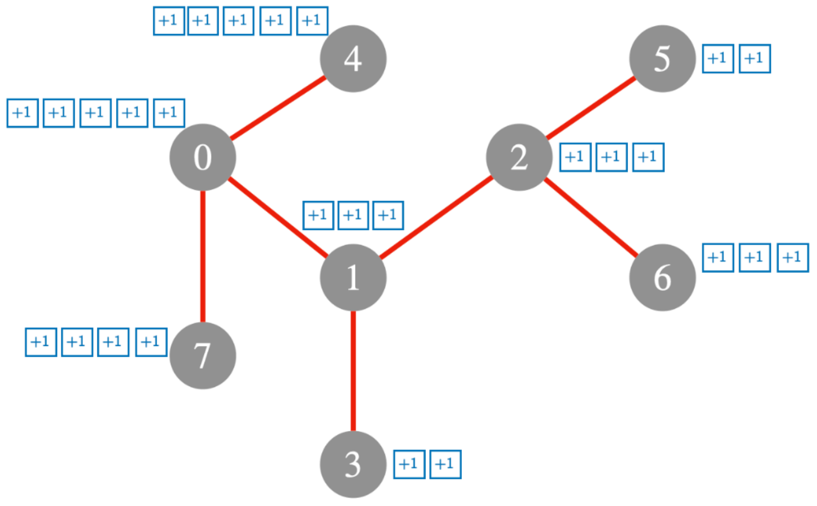
Examples:

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:**

`1`
`42 8`
	
	`Max Harmony: 5`





For the seed and the input length above, the following tree is generated (shown on the left).

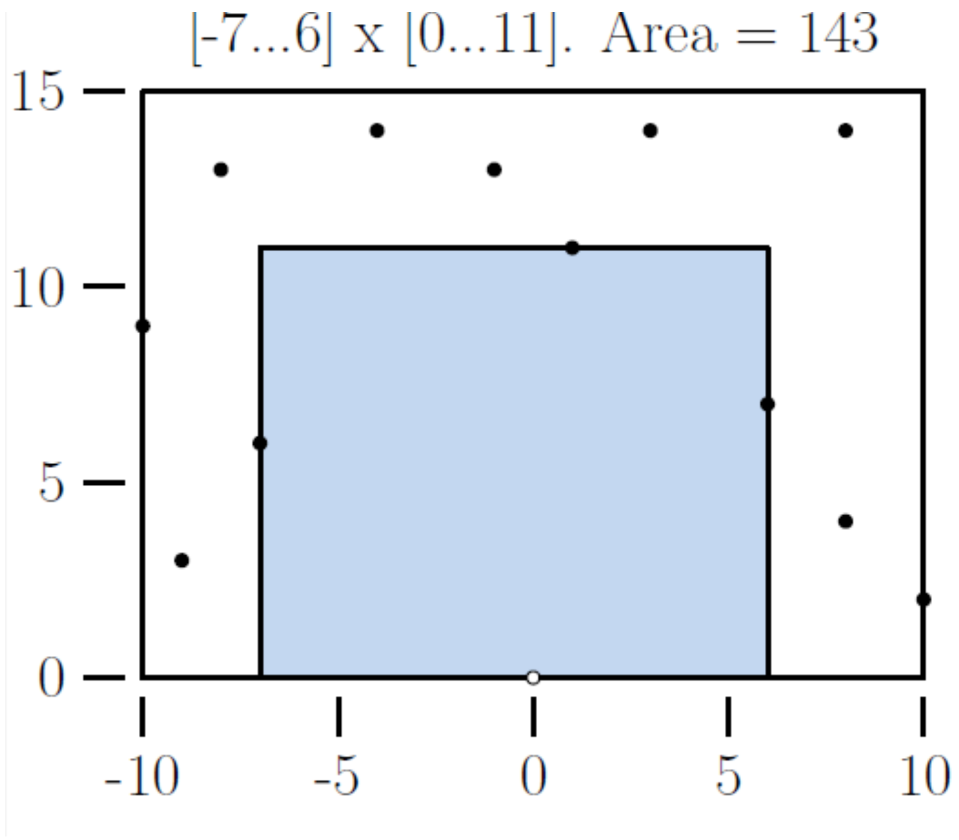
The red boxes show the unique integer strengths (think of these as edge weights). On the right, the total harmony incurred by each vertex as Mario runs his greedy strategy are shown as blue boxes, with a +1 for each unit of harmony accrued.

To illustrate how the strategy works, when Mario breaks the blockade with weight 0, its endpoints, vertices 0 and 4 gain 1 unit of harmony. Next, the blockade with weight 1 is broken, and 0, 4, and 7 all gain 1 unit of harmony. Next, the blockade with weight 2 is broken, and 0, 4, 7, and 1 gain 1 unit of harmony. Next, the blockade with weight 3 is broken, and only 2 and 6 gain 1 unit of harmony. (And so on).

Problem 6: Magical Platform Planning for the Star Festival

Mario has been tasked with a very special mission by Princess Peach. To celebrate the annual Star Festival, a grand event that lights up the night sky with dazzling star showers, Mario needs to create the largest possible magical platform in a designated area of the kingdom. This platform will serve as the main stage for the festivities, where inhabitants from all corners of the kingdom can gather to enjoy the spectacle. The designated area for the platform is a vast, open field near Peach's Castle, but there's a twist: the field is dotted with magical spots where Star Flowers grow. These flowers are rare and vital to the kingdom's magic, so the platform must be constructed without disturbing any of these precious blooms. The Star Flowers' locations are known, each marked by its coordinates in the kingdom's map, which fortunately, all have nonnegative y-coordinates.

Mario's challenge is to calculate the optimal dimensions for the magical platform. It must be a large, rectangular area that includes the central point of the kingdom (the origin, i.e., (0, 0)) but avoids overlapping with any Star Flower's location (a star flower can be at the edge of the platform – that is acceptable – see example below where "." indicates locations of star flowers). The platform's design is constrained by the maximum width and height specified by Princess Peach to ensure it fits perfectly within the designated area and maintains the harmony of the landscape.



The kingdom's scholars have provided Mario with a map detailing the coordinates of all Star Flowers, along with the maximum allowable dimensions for the platform. Mario needs to find the x-coordinates (xMin and xMax) and a single y-coordinate (yMax) that define the platform's dimensions. These coordinates must satisfy the conditions that:

```
-maxWidth <= xMin < 0 < xMax <= maxWidth
0 < yMax <= maxHeight
```

Additionally, no Star Flower can be located within the interior of the planned platform (but one can be at the edge of the platform), preserving the magic of the kingdom.

You need to help Mario complete the following function:

```
static ArrayList<Integer> getMaxPlatform(int maxWidth,
                                          int maxHeight, int[] x, int[] y)
```

that returns an

```
ArrayList
```

containing the four critical numbers: xMin, xMax, yMax, and the platform's final area

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 three numbers, maxWidth, maxHeight, M, where M is the number points. Next M lines contain coordinates of one point each. There may be up to 10000 input points that are in the range [-50,000,+50,000] x [0,+50,000].

Output:

For each test case, the program will output the largest platform that can be built.

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). We have also compressed the input onto fewer lines.

Input:	**Output:**
`1`	`Test case: 1`
`10 15 12`	`  Max Width: 10 Max Height: 15`
`-9 3 4 14 1 11`	`  Points: (-9,3) (4,14) (1,11) (6,7) (8,14) (-7,6) (-10,9)`
`6 7 8 14 -7 6`	`(-8,13) (8,4) (10,2)`
`-10 9 -8 13 8 4`	`          (-1,13) (3,14)`
`10 2 -1 13 3 14`	`  Final platform dimensions [-7...6] x [0...11]. Area = 143`

Problem 7: Chompy Chain Clusters

Bowser has captured Princess Peach and has scattered fragments of her essence across the Mushroom Kingdom! Mario must navigate a dangerous landscape to reunite fragments of her essence, but he's facing an intriguing challenge that he needs your help with. Chompy Chains, usually confined to their posts, have broken free and are now mysteriously manifesting throughout the Mushroom Kingdom. The Kingdom for this problem can be viewed as an unweighted $W \times W$ grid (or mesh graph) containing W^2 vertices.

Each Chompy Chain is initially contained at some vertex in the grid, and is initially dormant. Chompy Chains have the following unique properties:

- Activation Time: Each Chompy Chain has a distinct *start time*. They remain dormant until this time.
- Growth: Once activated, the Chompy Chain length grows outward at a rate of one unit per second, forming a circular cluster containing all vertices that the chain can reach.
- Collision: If two growing clusters collide at a vertex, then the cluster that reaches the vertex first (taking into account the start times of the clusters) captures that vertex.

Luckily, Mario has found the list of start times and starting locations of all Chompy Chains that will appear. Armed with this information, your task is to help Mario *determine the number of vertices that each Chompy Chain covers in its cluster.*

Note that every vertex will eventually be contained in some cluster since the number of Chompys is non-zero and the grid is connected.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases (*N*). After that, each line contains a test case first specifying the size of the grid (W) and the number of chains (*numChains).* These two numbers are followed by *numChains* lines containing edge tuples representing the location that a Chompy Chain will appear, and the start time of the Chain (see below for an example). The location on the grid is specified using the standard index notation for a 2D matrix, i.e., 3 2 corresponds to row 3, column 2.

Output:

One line per test case printing out an array containing the number of vertices that will be captured by each Chompy Chain's Cluster.

Note:

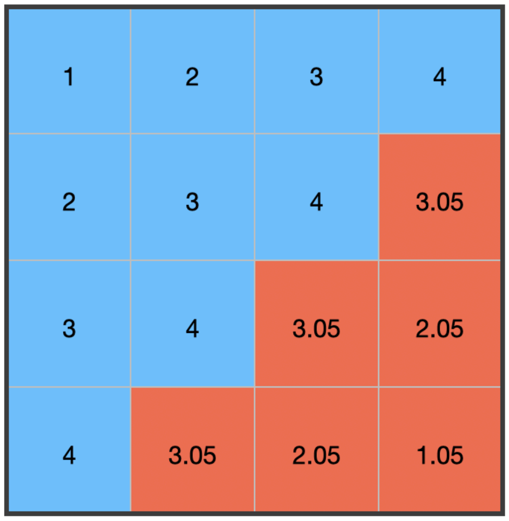
We have provided a skeleton program that reads the number of test cases and the input numbers. It also prints the output based on the `computeChompyChainClusterSizes()` method that you need to implement. You can assume that the starting time of each Chompy Chain is **distinct**.

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`	
`4 1`	`Cluster Sizes: 16`
`3 2 1.025`	
`4 2`	`Cluster Sizes: 16 0`
`3 2 1.025`	
`0 0 100`	
`4 2`	`Cluster Sizes: 10 6`
`0 0 1`	
`3 3 1.05`	

In the first example, we have a 4 x 4 grid containing 16 vertices and a single Chompy Chain. This Chain's cluster will cover all vertices, so the size of the cluster is 16 irrespective of its start time. In the second example, we added a second Chain, but its start time is extremely late. By the time it wakes up, it is already contained in the first Chain's cluster, hence its cluster size is 0.



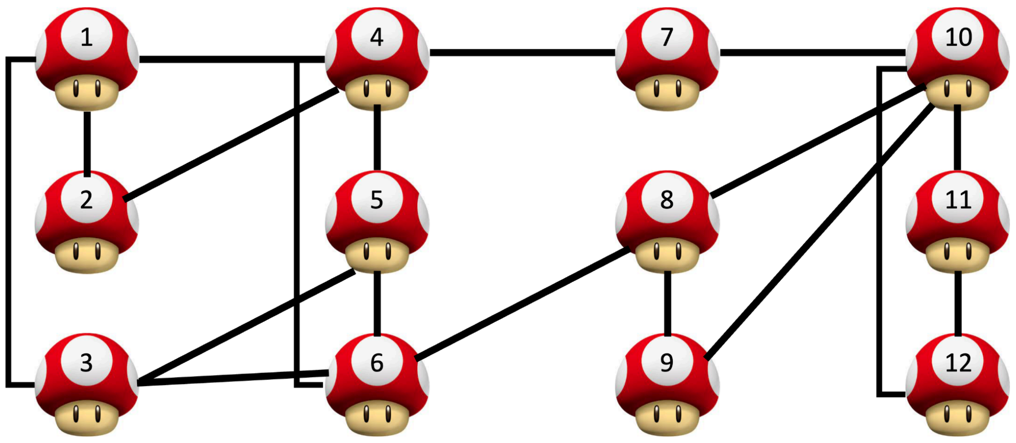
In the third example (illustrated here), we again have a 4 x 4 grid with 16 vertices, but now have two chains. The first chain starts at the top left (location 0 0) at time 1, and the second starts at the bottom right (location 3 3) at time 1.05. The clusters for both Chains are illustrated in different colors. The size of the location 0 0 Chain's cluster (colored blue) is 10, and the size of the other cluster (colored red) is 6. Each vertex (grid cell) is labeled with the time that a Chompy Chain reaches the vertex.

Problem 8: Colorful Chaos in the Mushroom Kingdom!

Bowser's mischievous Magikoopas have cast a chaotic spell, scrambling the colors of the Mushroom Kingdom! Mario and his friends have started restoring order, but they need your help to complete the task.

All the landmarks in the Mushroom Kingdom are colored with one of 6 colors, **red**, **green**, **blue**, **purple, orange,** and **yellow**. To avoid monotony, the King of the Mushroom Kingdom requires that any two landmarks that are connected to each other by a path must be of different colors. However, he also wants to use the minimum number of colors possible. Luckily for our heroes, the landmarks are arranged in a particular order making this a somewhat manageable task. Specifically, the landmarks are arranged in a grid-like structure with exactly 3 landmarks in each line. Furthermore, any landmark can only have paths to the landmarks in its group or with landmarks in the two groups adjacent to it (except for the first and the last group, which only have one group adjacent each).

As an example, in the figure below, landmarks are denoted by mushrooms. The first group contains landmarks 1-3, and these can only be connected to each other or to 4, 5, or 6. The second group (4-6) can only have connections to first group (1-3) or to third group (7-9), and so on.



Your goal is to figure out if it is possible to color the landmarks using less than 6 colors. It is always possible to color them using 6 colors, but the King will be quite unhappy if you use all of the colors.

Specifically, you should implement the function that takes as input the number of landmarks, M, (always a multiple of 3), and a list of paths (each path represented by two numbers between 1 and M):

```
   static int colorLandmarks(int M, int[][] paths)
```

which returns the minimum number of colors required between 1 and 5 as a:

```
   int
```

And -1 if it is not possible to color using less than 5 colors.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases (*N*). After that, for each test case, the first line contains *M,* and the number of paths, *E.* The next *E* pairs of numbers indicate the paths between landmarks for that test case.

Output:

For each test case, the program should output whether it is possible to color it using less than 6 colors.

Note:

We have provided a skeleton program that reads the number of test cases and the inputs. It also prints the output based on the `landmarkColors()` 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:**

`2`
`12 17`
`1 2 1 3 1 4 2 4 3 5 3 6`
`4 5 4 7 5 6 6 8`
`7 10 8 10 8 9 9 10`
`10 11 10 12 11 12`
	
	`Possible to color the landmarks with`
	`3 colors.`

`6 15`
`1 2 1 3 1 4 1 5 1 6`
`2 3 2 4 2 5 2 6`
`3 4 3 5 3 6`
`4 5 4 6`
`5 6`
	`Not possible to color the landmarks`
	`with less than 5 colors.`

Problem 9: Bowser's Diabolical Parentheses

Princess Peach finds herself on yet another critical mission to rescue Mario, who has been ensnared by Bowser's minions in a distant castle. As she hastens to his aid, she encounters an unexpected blockade—a magical gate sealed by a spell that can only be unlocked by balancing a string of unbalanced parentheses that are imprinted on the gate. Peach's arrival at the gate also started a countdown (of 10 seconds) after which the gate will be shut permanently. However, if Peach makes a move, i.e., swaps one ")" with "(" as described below, then the countdown will reset.

Usually Peach is in a hurry to get wherever she is going, and knows how to balance the parentheses in the lowest number of moves. However, she wants to delay opening the gate and facing Bowser's minions until Luigi arrives. So she would like your help to keep resetting the countdown timer as long as possible.

The string she is looking at contains exactly the same number of ")" and "(" characters, but is unbalanced. For example, the starting string might look like:

`))(())((`

Each move involves swapping a ")" at position *i* with a "(" at position *j,* where *i* < *j* (i.e., you are not allowed to try to make the string even more unbalanced). The minimum number of moves to balance this string is just 1 – swapping first and last positions gives us a balanced string.

```
    ))(())((  == 1-8 ==> ()(())()
```

However, Peach is looking to maximize the number of swaps before the string is balanced. For example, the following sequence takes longer:

```
    ))(())((  == 1-2 ==> ()(())((  == 7-8 ==> ()(())()
```

And in fact, this one is even longer:

```
    ))(())(( = 1-4 => ()(())(( = 2-3 => (()())(( = 3-4 => ((()))(( = 4-7 => (((())))
```

Formally, a string S is balanced if:

1. S = empty
2. S = S1 S2, where both S1 and S2 are balanced
3. S = (S1) where S1 is balanced

Specifically, you should implement the function that takes as input the string:

```
   static int longestSequenceOfSwaps(String str)
```

which returns the maximum number of swaps possible before the string becomes balanced:

```
   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 a single string on one line. The maximum number of characters in a string will be <= 100000.

Output:

For each test case, the program will output the maximum number of swaps possible before the string becomes balanced.

Note:

We have provided a skeleton program that reads the number of test cases and the input strings. It also prints the output based on the `longestSequenceOfSwaps()` 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`	
`()()`	`The maximum number of swaps possible is 0`
`))((`	`The maximum number of swaps possible is 3`
`)(()()`	`The maximum number of swaps possible is 2`