DKC^3 2017 – Computing Problems

1. Mars Rover (75 points)

In previous years your school's teams have tried, and probably failed, to help NASA with problems regarding the rovers they have on Mars. This year your school will have another chance to fail…I mean help NASA. The newest rover, Curiosity, has made a startling discovery. With its ground penetrating sonar it has discovered a series of underground tunnels. They appear to be devoid of life and the tunnels are too precise to be naturally occurring, and so are evidence of the planet having been inhabited by some alien race sometime in the past. The tunnels run underground at a depth of 1 kilometer. The tunnels follow a grid design and have junctions every kilometer. Some junctions act only as a terminator to a tunnel, some only connect two tunnels and some connect three or four tunnels. NASA can't wait to explore these tunnels! There are two problems. The first problem is that there doesn't seem to be any access to the tunnels from the surface. The second problem to the tunnel exploration is that Curiosity is battery operated and recharges its batteries via solar energy. As I'm sure you can understand, there's not a lot of sun in a tunnel 1 kilometer under the surface. To gain access to the tunnels NASA has decided to drill down to the junctions of the grid lines. It wants to drill as few holes as possible as the drill time for even one hole is many days. NASA has calculated that the rover can operate for no longer than 8 hours on a single day's charge while doing its exploration of the tunnels. Taking into consideration travel down to the tunnels and travel time back to the surface, it's been decided to not let Curiosity explore for more than 5 hours at a time. Assuming an average time of 1 kilometer per hour for the rover while in the tunnels, it's up to you to decide at which grid coordinates to drill the fewest holes possible in order for Curiosity to access every kilometer of tunnel.

Input

The first line of each test case will be two whole numbers, separated by a space, representing the size of the grid. (X,Y) where X is the horizontal axis and Y is the vertical axis. X and Y will be greater than 0 and smaller than 101. Each of the next lines of the test case will consists of two pairs of coordinates representing a grid segment that has a tunnel. The lower left corner of the grid will be (0,0). Each test case will be separated by a blank line.

Output

The minimum number of holes that would need to be dug to cover all the tunnels.

Input File: D:\DKC3\MarsIn.txt
Output File: D:\DKC3\MarsOut.txt

DKC^3 2017 – Computing Problems

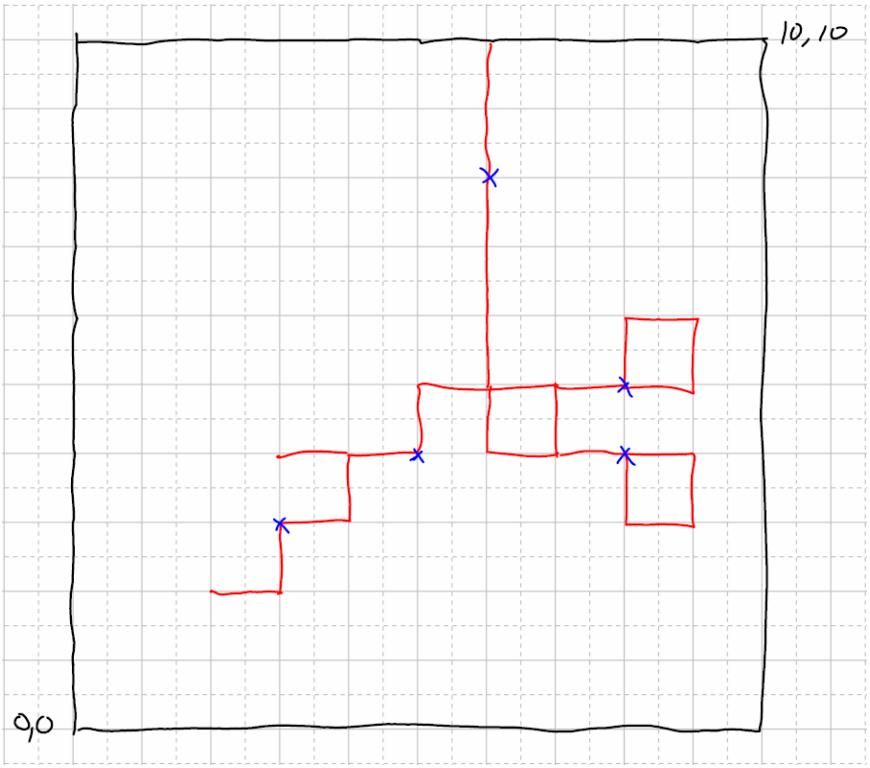
Examples:

Input:

2 2
(0,1) (1,1)
(1,0) (1,1)
(1,1) (1,2)
(1,1) (2,1)

10 10 /* picture representation shown below */
(2,2) (3,2)
(3,2) (3,3)
(3,3) (4,3)
(4,3) (4,4)
(4,4) (3,4)
(4,4) (5,4)
(5,4) (5,5)
(5,5) (6,5)
(6,5) (6,6)
(6,6) (6,7)
(6,7) (6,8)
(6,8) (6,9)
(6,9) (6,10)
(6,5) (6,4)
(6,4) (7,4)
(7,4) (8,4)
(8,4) (9,4)
(9,4) (9,3)
(9,3) (8,3)
(8,3) (8,4)
(6,5) (7,5)
(7,5) (7,4)
(7,5) (8,5)
(8,5) (9,5)
(8,5) (8,6)
(9,5) (9,6)
(9,6) (8,6)

DKC³ 2017 – Computing Problems



Output:

1

5

2. Bouncing Bunnies (75 points)

Connie and Ronnie, the bouncing bunnies, enjoy frolicking in the hills. They are both very adventurous and seek extreme weather changes. Connie loves changes in the temperature, so when she bounces from one hill to another, her happiness during that bounce is equal to |TA – TB| (absolute value), where TA is the temperature of the hill she jumped from, in bunny-degrees, and TB is the temperature of the hill she jumped to, also in bunny-degrees. On the other hand, Ronnie loves changes in humidity, so when she bounces from one hill to another, her happiness during that bounce is equal to |HA – HB| (absolute value), where HA is the humidity of the hill she jumped from, in bunny humidity units, and HB is the humidity of the hill she jumped to, also in bunny humidity units.
Connie and Ronnie are good friends, and would like to travel together across a field full of hills (starting at their home and ending at their favorite tree), but in order to relate to each other as well as possible, they would like Connie's happiness level to be equal to Ronnie's during every bounce (jump) they make.

Given the weather data for a field of hills, determine the minimum number of jumps needed for Connie and Ronnie to get from their home to their favorite tree. Bunnies are so good at bouncing that they can jump from any hill to any other hill, i.e., any hill is within a single bounce's distance of any other hill.

Input:
There will be ten test cases (i.e. fields). The description of each field will start (on a new line) with a positive integer, n (2 <= *n* <= 500,000), denoting the number of hills in the field. The following input line will contain n positive integers – the ith number on this line, Ti (1 <=Ti <= 10^6), will denote the temperature of the ith hill in bunny degrees. The next input line (the last line of each field description) will consist of n positive integers – the ith number on this line, Hi (1 <= Hi <= 10^9), will denote the humidity of the ith hill in bunny humidity units. The bunnies' home is located on hill 1, and their favorite tree is located on hill n.

Output:
For each field, output must consist of a single line of the following form: "Field #f: b", where f is the field number in the input starting from 1 and b is an integer – the minimum number of bounces (jumps) needed for Connie and Ronnie to get from their home to their favorite tree, or the number -1 if such a journey cannot be made by the pair of bunnies.

Input File: D:\DKC3\BunnyIn.txt
Output File: D:\DKC3\BunnyOut.txt

Examples:
Input:
5
1 2 4 7 11
5 12 14 11 3
3
1 5 2
6 2 2

Output:
Field #1: 4
Field #2: -1

3. Molkky (75 points)

A “friend” has “asked” you to create an automated scoring system for a game called Molkky. The players use a wooden throwing pin (also called "mölkky") to try to knock over wooden scoring pins (also called "skittles"). The skittles are marked with numbers from 1 to 12. Knocking over one skittle scores the amount of points marked on the skittle. Knocking 2 or more skittles scores the number of skittles knocked over (e.g., knocking over 3 skittles scores 3 points). After each throw, the skittles are stood up again in the exact location where they landed. The first one to reach exactly 50 points wins the game. Scoring more than 50 will be penalized by setting the player's score back to 25 points. A player will be eliminated from the game if they miss all the skittles three times in a row.
Your “friend” will provide you a file with scoring data that needs to be processed.

Input:

1. The first line of the file will be the number of games in the data file.
2. Each game’s data will begin with a line indicating the number of players in the game. There will always be at least one player for a game.
3. Subsequent lines will represent each throw of each player in the game. Each line will follow these formatting rules, separated by at least one whitespace:
    a. Player number
    b. a list of each skittle knocked down. If all skittles were missed, the list will be empty.
    c. an optional Two asterisks will signal the end of the data for that line, after which no further data should be processed on that line (your “Friend” sometimes likes to put notes here to remind herself about each throw.)
4. Two asterisks on a line by themselves signal the end of the data for the current game.

Output:

When someone wins the game or when the input data runs out for a game, stop all further processing of that game’s data and summarize the game score on a single line. This summary should follow this format, ordered first by score descending, then by player number ascending:
1 (2,3) (2,3) (…) 4
This format corresponds to the following:

1. Game number
2. Player number
3. Score.
    - Repeat 2 and 3 for each player.
    - Because your “friend” likes to overcomplicate things, if a player was eliminated their score should be treated as a negative to represent their elimination.
    - If they were eliminated with a zero score, output this as negative zero. Your “friend” doesn’t care that there’s no such thing as ‘negative zero.’ She does feel that ‘negative zero’ is less than zero for sorting purposes, though, so that’s something.
4. If someone won this game, their player number. Otherwise, a capital X.

Input File: D:\DKC3\MolkkyIn.txt
Output File: D:\DKC3\MolkkyOut.txt

DKC^3 2017 – Computing Problems

Examples:
Input:
2
2
1 12 3 **Nice throw, bad bounce
2 12 **whoa, how'd steve snipe that skittle
1 1 2 3 **Frank got a bad sliver trying to show off
2 12
1 ** that sliver is messing with Frank's concentration
2 12 ** steve is on FIRE
1 ** focus, frank...
2 12 1 **
1 ** aww, Frank.
2 12 ** Yay steve.
1 1 2 3 4 5 6 7 8 9 10 11 12 ** come on frank, don't be a poor sport and kick over all the pins after the game is over :(
**
3
1 7 9 8 **crazy backspin
2 ** 5 nearly went over
3 10 3 7 1 9 8 **That'll leave a mark **
1 1 2 ** ending early because Frank is still bleeding from game 1
**

Output:
1 (2,50) (1,-5) 2
2 (3,6) (1,5) (2,0) X

DKC^3 2017 – Computing Problems

4. QR Code (2D Barcode) (75 points) Tim, Peter and Elizabeth

The QR Code (Quick Response Code) was created with the purpose of tracking vehicles through the manufacturing process, but quickly grew beyond its original purpose due to its quick readability and increased storage capacity over standard UPC Barcodes which are commonly used on merchandise. One of the earliest versions of these codes was a simple 21x21 matrix, but they have grown quite complex over the years with the current version which is a 177x177 matrix (Figure 1).

Figure 1

Reading these codes, at least the early versions, can be completed by hand while following a specific set of steps, but first you must understand the basic structure of one of these codes. The following sections will lay out the following information; Orientation, Data, Version Info, Error Correction, and Format Info. These sections are laid out below (Figure 2), but for the purpose of this programming problem, we will be ignoring the Version Info and the Error Correction section. The un-colored sections are ignored entirely.

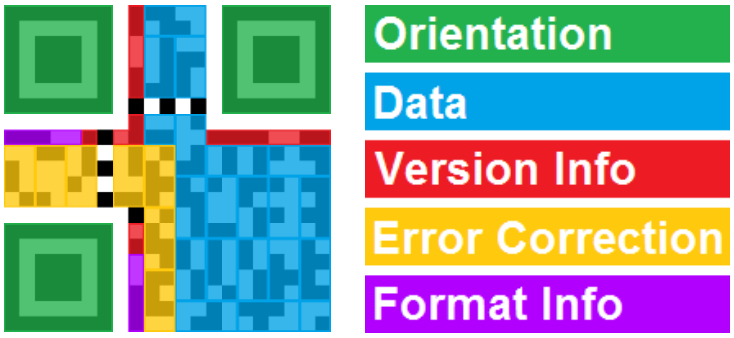


Figure 2

Orientation:
Orientation is determined by the large boxes on 3 corners of the QR Code, highlighted above in **GREEN**. When reading in the image, a user may not know the proper orientation, so it is up to the program to read in the image and orientate it in a way it can understand. For proper orientation the squares should be located in the top left, top right, and bottom left.

DKC^3 2017 – Computing Problems

Data:
The data section, highlighted above (Figure 2) and below (Figure 3) in **BLUE**, consists of a START which is 4 BITS in the lower-right corner. These 4 bits indicate the storage type, which can be 0001 (numeric), 0010 (alphanumeric), and 0100 (8-bit). **For the purpose of this problem, we will be strictly using the 8-Bit storage**. Following the flow in the below image, the next BYTE of data directly above the START is the LENGTH of the overall data contained in the QR Code with a maximum of 17 BYTES. The length is important since some of the bits may be flipped due to the layer masks which will be discussed later.

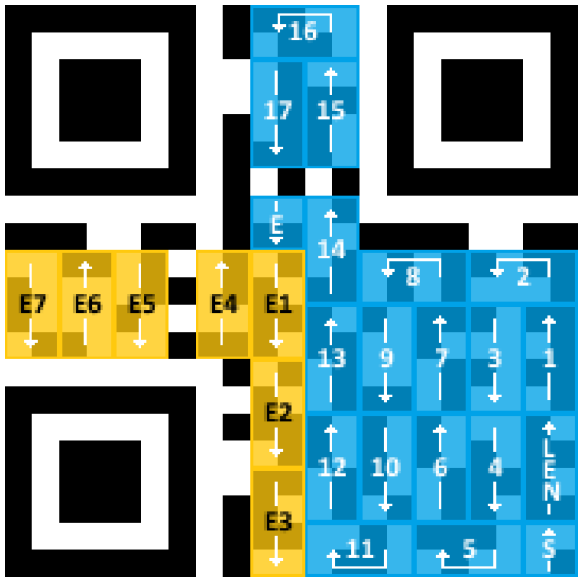


Figure 3

The bits are read in the following order (Figure 4), and once unmasked, the value of those bits are mapped to the Extended ASCII Character Table (0-255).

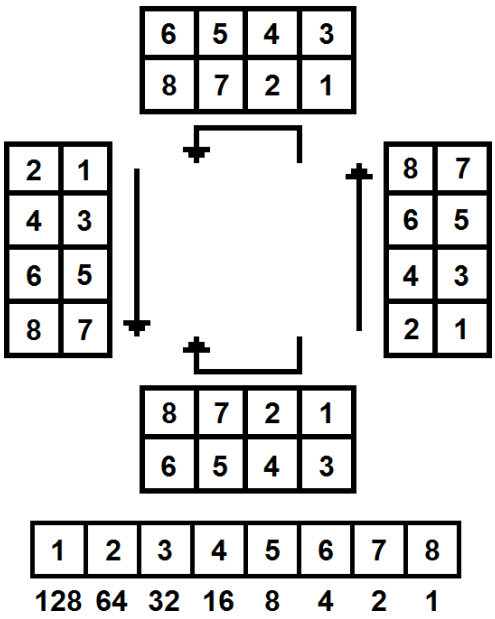


Figure 4

As mentioned above, the data is masked. This is done to avoid large groupings of bits in the QR Code and help to obfuscate the data while making it simply easier to look at. There are 8 different masks that can be applied, which one is determined by the Format Information section of the QR Code which is colored purple in the layout diagram (Figure 2). Below (Figure Group 5) is a rundown of the different masks available as well has how they are achieved. In general, any bits located within the highlighted portion of the masks are flipped, 0 to 1 and 1 to 0. **An important thing to note is that only the data and error correction sections are masked, not the orientation, format or version information as those are needed to tell you what mask to use.**

000
(i + j) % 2 = 0

001
i % 2 = 0

010
j % 3 = 0

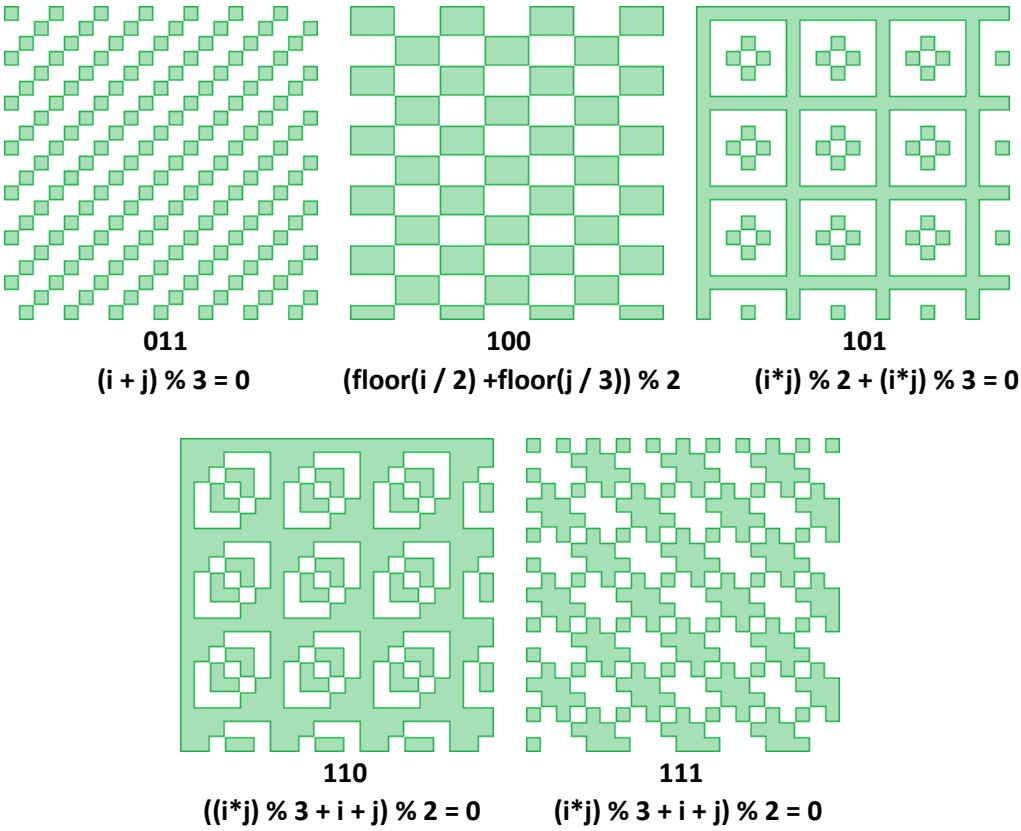


Figure Group 5

Format:

The format of the QR Code is located just below the top left orientation marker (read left to right) and a duplicate of this section is located to the right of the bottom left orientation marker (read bottom to top). We have highlighted this section in **PURPLE** in the sectional image (Figure 2). This section consists of 5 bits which contain the level or error correction and which mask to use. More specifically, the first two bits are the level of error correction, while the last three specify the mask. While this section is not masked, **it has been XOR'ed with 10101**, and must be returned to its original state in order to be read.

Reading the QR Code:

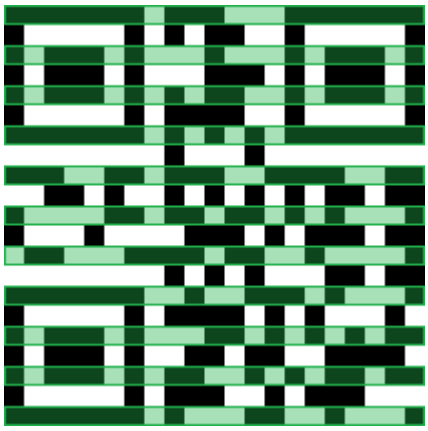
Given the above information, we will try to read the following QR Code which is already been orientated.

QR Code

Figure 6

First, we will grab the format information. This was the **PURPLE** below the top left orientation marker, or to the right of the bottom left one. We have 11100 visible. When performing an XOR with 10101, we get the following: 11100 ^ 10101 = 01001. And since we are ignoring the error correction for this problem, we can omit the 2 leading bits and are left with 001, the second mask.

Second, we apply our mask to the QR code in order to determine which bits to flip and how to read them. The below images are the original code masked, and unmasked. **Remember, the orientation, format and version bits are not masked and should be ignored when unmasking the code.**



Masked (with Mask Highlight)

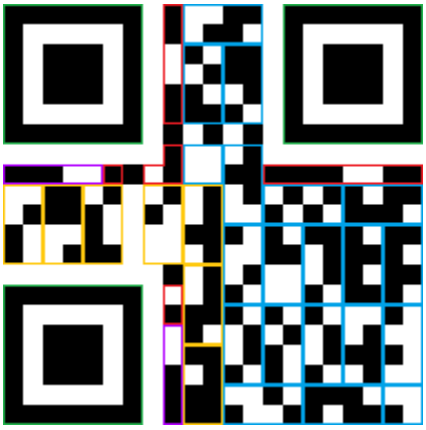
Figure 7

Unmasked

Figure 8

Third, we read the data! We will need to follow the proper flow of the data as described in the data description above (Figure 4) using the newly unmasked QR Code.

DKC^3 2017 – Computing Problems



Unmasked (Key Outline)

Figure 9

START: Reading from the lower right, we have 0100 which is our data type, which is always going to be 8-Bit for the purpose of this problem.

LENGTH: This section we have 00010001, which is 17. Our data is going to be 17bytes long.

DATA:

Section 1: 01110111 = 64 + 32 + 16 + 4 + 2 + 1	= 119	= w
Section 2: 01110111 = 64 + 32 + 16 + 4 + 2 + 1	= 119	= w
Section 3: 01110111 = 64 + 32 + 16 + 4 + 2 + 1	= 119	= w
Section 4: 00101110 = 32 + 8 + 4 + 2	= 46	= .
Section 5: 01110111 = 64 + 32 + 16 + 4 + 2 + 1	= 119	= w
Section 6: 01101001 = 64 + 32 + 8 + 1	= 105	= i
Section 7: 01101011 = 64 + 32 + 8 + 2 + 1	= 107	= k
Section 8: 01101001 = 64 + 32 + 8 + 1	= 105	= i
Section 9: 01110000 = 64 + 32 + 16	= 112	= p
Section 10: 01100101 = 64 + 32 + 4 + 1	= 101	= e
Section 11: 01100100 = 64 + 32 + 4	= 100	= d
Section 12: 01101001 = 64 + 32 + 8 + 1	= 105	= i
Section 13: 01100001 = 64 + 32 + 1	= 97	= a
Section 14: 00101110 = 32 + 8 + 4 + 2	= 46	= .
Section 15: 01101111 = 64 + 32 + 8 + 4 + 2 + 1	= 111	= o
Section 16: 01110010 = 64 + 32 + 16 + 2	= 114	= r
Section 17: 01100111 = 64 + 32 + 4 + 2 + 1	= 103	= g

END: The final 4 bits are all blank, which is NUL, and the end of our string of bytes.

After reading all 17 bits, we get “www.widipedia.org” as our decoded message!

There will be 10 test cases for this problem with each test case consisting of a 21x21 matrix of blanks spaces and hashtags (#). Each test case will be separated by a line with a single asterisk (*). The output will consist of a single line formatted string with labels that shall contain the mask bits, the length of the encoded message in decimal form, and the actual message. Using the above example, the output should be the following: ***Note the spacing and casing, and punctuation.**

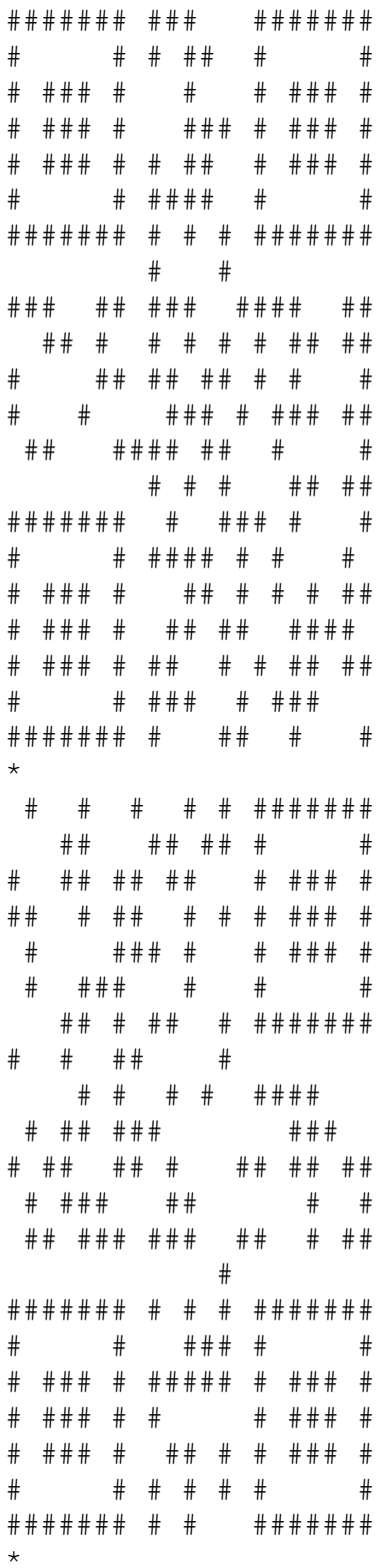
Mask: 001, Length: 17, Message: www.wikipedia.com

Input File: D:\DKC3\QRCodeIn.txt
Output File: D:\DKC3\QRCodeOut.txt

DKC^3 2017 – Computing Problems

Examples:

Input:



Output:

Mask: 001, Length: 17, Message: www.wikipedia.com
Mask: 111, Length: 8, Message: *Abc123$