CoE 164

Computing Platforms

Machine Problem 01

Academic Period: 2nd Semester AY 2020-2021
Workload: 3 hours
Synopsis: Make your 2D matrix barcode more robust!
Submission Platform: UVLe

Description

It is 1992, and you are working for a division that needs to keep track of thousands of automotive parts. The division uses EAN barcodes to track them, which you've come to realize as a pain in the back. You need to fill out several forms every day, and scan probably the same barcode multiple times to automatically fill them.

One day, your division head suddenly had a wild idea - create a new barcode! "Since EANs only encode a 13-digit number, why can't we have barcodes having all the information we need?", your head said. Your attention was then turned to a game of Othello, where white and black markers are arranged on a square board. These markers can stand for a bit, and information can be encoded to fill the whole board. Your head then decided to work with you on a barcode specification similar to a game of Othello.

Months went by, and after a year, your two-man team has developed a 2D barcode specification! This barcode consists of black and white square "modules" arranged on a square grid, which can be read by a relatively cheap camera on any orientation. It can even encode three different data types - numbers, alphanumeric characters, and byte data. There is one hiccup though - you want the barcode to sustain some kind of while still being

readable. Since this is a grid-type (or matrix) barcode, it is possible for the barcode to be accidentally cut, spliced, or drawn over during transport. You have read something about Reed-Solomon error-correcting codes, which were used in NASA space missions, and have decided to apply these codes into your barcode. These codes will be appended to the data to be encoded into the barcode, and generation of these codes will be written as a module to a larger program that encodes data into this grid-type barcode.

Reed-Solomon error-correcting codes are actually a family of codes that can detect both erasures and errors in a message. The most common and easier implementation of generating such codes is through the “BCH view”, named after BCH (Bose–Chaudhuri–Hocquenghem) codes. BCH codes can be constructed by using a *generator polynomial* over a *finite field*. The message is expressed as a polynomial over this field, and it gets divided by the generator polynomial in each iteration. The coefficients of the remainder polynomial of this operation become the error-correcting codes associated with that message.

Your barcode will have error-correction codes between 1 and 255 inclusive, with all of them can be represented with a byte, or eight bits. In math jargon, we will do our division and generation of the polynomial over the field F_{256}. Operations on this finite field $F_{256}$are to be done such that for a number $\exists n \in N^0, n \in F_{256} \mid 0 \leq n < 256$. In our “BCH view”, addition and subtraction of two numbers in this field is actually a single operation - an XOR operation. An additional modulo operator may also be performed on the result so that its value stays within the range 1 and 255.

Given a data or message represented as a string of bytes, error-correcting codes are generated by first determining the number of codewords needed. Your barcode has a version, pertaining to its size, and an ecc level, pertaining to how robust the barcode should be. A higher ecc level can encode more codewords at the expense of being able to encode only fewer data bytes. The table below shows the codewords that need to be generated given an error-correction code (ecc) level and version of the barcode.

	ecc level			
version	L	M	Q	H
1	7	10	13	17
2	10	16	22	28

After determining the number of codewords to generate, a generator polynomial should be built corresponding to c codewords. The polynomial, still over F_{256}, is given by

$$g(x) = (x - \alpha^0)(x - \alpha^1)\ldots(x - \alpha^{c-1})$$

$$= \prod_{i=1}^{c-1} (x - \alpha^i)$$

where $\alpha = 2$, mirroring the fact that the message is encoded in bits. α is called the *primitive root*. Since we are operating in a finite field, we would like to make sure that the powers of α above 8 are still within 1 and 255, especially that, for example, $\alpha^8 = 2^8 = 256$ under normal math. Hence, to calculate α^d, we multiply α d times, but every time the value becomes 256 or greater, it should be reduced to "modulo" (or XOR more accurately) 285 before multiplying again. 285 is just a random number you came up with. Therefore, in our math world, $\alpha^8 = 29$and $\alpha^9 = 58$, which looks odd when α is directly substituted with its value. Additionally, when multiplying two powers of α, the exponent should be reduced to modulo 255 if the resulting exponent is 256 or greater. As the multiplication takes place, working with the logs and antilogs of α may be an essential step in making the process tenable.

After a long way building the generator polynomial, it is now time to generate the codewords themselves by dividing the original message with the generator polynomial. If the message consists of Nbytes, then the message polynomial has degree x^{N-1}, and is defined as the message bytes as a separate term, with the first byte paired with the highest degree. For example, if the message bytes are [1, 6, 4], then the message polynomial is therefore $m(x) = x^2 + 6x + 4$. Next, the generator polynomial $g(x)$ corresponding to c codewords is multiplied by $x^{N-1}$and $m(x)$ by $x^c$to make sure that the exponents don't get too small (or negative) during division. Finally, $\frac{m(x)}{g(x)}$ is performed to obtain the codewords, which are the coefficients of the remainder.

To perform long division, we first get the coefficient of the highest term of $m(x)$ and multiply it with $g(x)$ to get $d(x)$. Then, we "add" (or XOR more accurately) this and the message polynomial ($d(x) \oplus m(x)$) to get the first running remainder polynomial $r_1(x)$. Now, we perform these series of operations c times, with the previous $r(x)$ becoming the "new" $m(x)$. After the long process, we get the final running remainder $r_c(x)$, whose c coefficients will be the error-correcting bytes that we need.

With only a few weeks to spare before the next general division meeting, you are inclined to finish this module fast so that you can demo the whole system successfully. Your division head counts on you, too, as he is now busy with managerial work and cannot make time to help you in the meantime.

Input

The input to the module starts with a number T on a line indicating the number of messages. Then, each message is provided in two separate lines. The first line consists of a number Vindicating the version of the barcode, a letter M indicating the error correction level needed for the message, and a number N indicating the size, or number of bytes, of the message. The next line consists of Ndecimal numbers denoting the bytes of the message itself.

Output

The output should consist of N lines, with each line i consisting of E numbers. These numbers denote the decimal numbers of the bytes of the codewords corresponding to the i th message.

Example

Input

2
1 M 16
32 91 11 120 209 114 220 77 67 64 236 17 236 17 236 17
1 H 9
32 50 52 78 228 72 236 17 236

Output

196 35 39 119 235 215 231 226 93 23
135 83 157 250 127 45 83 50 167 179 104 18 214 60 194 94 77

Additional Description/Requirements

The module will only accept these range or set of variables:
$T \leq 10$
$V \in \{1, 2\}$
$M \in \{'L', 'M', 'Q', 'H'\}$
$1 \leq N \leq 34$

Since you are developing a barcode specification, you decided to write a short journal so that you can refer to it when the time comes to formalize and standardize it. You decided to

write which programming language you used to write the algorithm and a short description of the algorithms in that language 1) to obtain α^n, 2) to obtain $g(x)$, and 3) to obtain the codewords via long division. As you have access to the Future Web Archive™, you can use any programming language of the future to develop your error-correcting codeword generator module.

You luckily maintain a digital notebook of your general algorithm musings at work. Attached with this paper is a file of your notes depicting your algorithm formulation. For more information about the future of your error-correcting algorithm, you've also decided to visit the Archive™ and found this link:

https://www.thonky.com/qr-code-tutorial/error-correction-coding

The website will exist in the year 2001 and is last updated in 2021, around 28 years after your predicament! Further scrolling showed that your barcode specification will become world-famous and ubiquitous starting 8 years after in 2002.

Upload both your module (as a single source code file; in TXT if the system does not support the file extension) and your short journal (PDF or TXT file) to your remote repository in your division.

Grading Rubric

5% Input handling - able to read the input specifications
10% Algorithm to compute α^n
30% Algorithm to build a generator polynomial
35% Algorithm to divide and output the codewords
20% Short journal