CoE 164

Computing Platforms

Machine Problem 03

Academic Period: 2nd Semester AY 2021-2022
Workload: 3 hours
Synopsis: Faster flights with barcodes!
Submission Platform: Google Forms

Description

Do you remember the 2D barcode made in the 1990s for tracking automotive parts? Two people only created that barcode that is now the QR code! Currently, the company holding the patent does not exercise it, which enabled its widespread use into the late 2010s. All smartphones now have a camera that also enabled its relatively easy adoption.

Fast forward into late 2004, the International Air Transport Association (IATA) has become interested in streamlining the airport boarding process by mandating barcodes in the paper boarding passes of all airlines around the world. That time, they were mulling over several barcode standards, including yours that was developed during your stay in a barcode scanner company around the same time as when the QR code standard was released. You realized that the IATA may have read your 1995 patent for such a system using your barcode. With IATA aiming to release their barcode system around 2005, you are now tasked with providing a sample barcode scanner for them to test.

The barcode you have developed is a 2D barcode that looks like the usual grocery barcodes and is read from top to bottom and left to right. In each row, there can be between 3 and 30 data columns, with each column containing a *symbol character* consisting of 17 white and black spaces in some configuration. The 17 white and black spaces correspond to 929 unique values since only the white-black combinations that yield groups of exactly four contiguous whites and blacks each, arranged in an alternating fashion, are used. Each symbol character is mapped to a *symbol character value* (SCV). One SCV can correspond to one of the three symbol characters depending on which row it is located. This redundancy ensures that scanners can easily determine the location of the data in relation to the barcode, enables reading at any orientation, and reduces errors.

The barcode itself has more symbols, such as row indicators and start and stop symbols, for added redundancy. Fortunately, you have finished writing the module for detection and reading of the barcode from print, so reading of the symbol characters themselves has been taken care of. However, you are not yet finished implementing the decoding and error correction modules for your scanner.

This scanning module returns the SCV list itself, which can contain values from 0 to 928 inclusive. The first element of the list is the *symbol length descriptor*, or the size of the list *excluding* the error correction codewords. The next few values are the data codewords, followed by a variable amount of padding of value 900, and finally the error correction codewords. The padding is needed so that the SCV list fills the whole barcode.

To decode the data from the SCV list, the error corrector should be run first through the list to correct any erroneous SCVs. Then, the data codewords are extracted from the list, and the barcode decoder is run through it to retrieve the encoded data.

Error Correction

The barcode should be able to withstand some environmental exposure while still being readable. As such, it is imperative to have a system to check whether the data read in the barcode is the true one encoded in it. You remembered that the barcode specification used Reed-Solomon error-correcting codes to provide for barcode integrity and correction when printers sometimes smudge ink over the barcodes. The error correction is applied to the partial SCV list containing the symbol length descriptor, data codewords, and padding.

Reed-Solomon error-correcting codes are 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.

The barcode encodes its SCVs between 0 and 928 inclusive, with all of them represented with three different bar-space arrangements. In math jargon, we will do our division and generation of the polynomial over the field F_{929}. Operations on this finite field $F_{929}$are to be done such that for a number $\exists n \in N^0, n \in F_{929} \mid 0 \leq n < 929$. In our "BCH view", addition and multiplication is done as is. An additional modulo operator may also be performed on the result so that its value stays within the range 0 and 929. On the other hand, to subtract in our view, we should first find the complement of the subtrahend by subtracting 929 with it. Then, we add the two numbers modulo 929 to get the result. Finally, to divide in our view, we should first find the complement of the divisor by performing the Extended Euclidean algorithm. Then, we multiply the two numbers modulo 929 to get the result.

Given a data or message represented as a string of integers with M data and E error-correcting codewords concatenated in that order, for a total of T integers, we aim to check whether there are errors in the message by deriving an error polynomial $e(x)$, such that subtracting the message from this polynomial results in the corrected message. As such, the whole message can be treated as a polynomial $m(x)$ as shown below.

$$m(x) = M_0 x^{m-1} + M_1 x^{m-2} + \ldots + M_m$$

The barcode provides an error correction code (ecc) level, pertaining to how robust the barcode should be. A higher ecc level encodes more error correcting codewords at the expense of being able to encode fewer data. The table below shows the number of error correcting codewords corresponding to the ecc level of the barcode. In general $E = 2^{ecc\_level+1}$. Such error correcting codewords can correct only up to $E/2$ codewords.

ecc level	ecc count
0	2
1	4
2	8
3	16
4	32
5	64
6	128
7	256
8	512

Next, the value of $m(x)$ for $x \in [\alpha^1, \alpha^E]$ is then evaluated to get the *syndromes* S_i of the message, where $\alpha = 3$. 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 6 are still within 0 and 928 since, for example, $\alpha^7 = 3^7 = 2187$ under normal math. Hence, to calculate α^d, we multiply α d times, but every time the value becomes 929 or greater, the resulting value should be reduced to modulo 929 before multiplying again. Therefore, in our math world, $\alpha^7 = 329$ and $\alpha^8 = 58$, 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 929 if the resulting exponent is 929 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.

If all of the syndromes are zero, then the message does not have any errors. Otherwise, we have to correct the message by first finding the *error locator polynomial* $\Lambda(x)$. The syndromes and the error locator polynomial relate to the error polynomial that we aim for. There are various ways of computing for $\Lambda(x)$, but the most common ones are the Petersen-Gorenstein-Zeirler (PGZ), Berlekamp-Massey (BM), and Sugiyama algorithms. $\Lambda(x)$ can be expressed in terms of the following equation:

$$\Lambda(x) = \Pi^{r}_{i=1}(1 - xX_i) = 1 + \sum_{i=1}^{r} \lambda_i x^i$$

Note that r is the number of roots, $m - 1 - d$ the index of the jth error in the message derived from $X_j = \alpha^d$, and X_i^{-1} the ith zero of the equation.

After having the error locator polynomial, we compute for its zeros such that $\Lambda(x) = 0$. Since we are computing over F_{929}, the zeros can only be of values between 0 and 928 inclusive. We can use trial substitution evaluating $\Lambda(x)$ for $x = \alpha^0, \alpha^1$, etc and checking whether the polynomial evaluates to zero for such value. If $\alpha^x | \, x \leq 0$ is a zero of this polynomial, then an error exists at the xth codeword. An optimization of this task can be performed using *Chien search*, which lets you evaluate $\Lambda(\alpha^{x+1})$ by multiplying the coefficients of $\Lambda(\alpha^x)$ with the corresponding coefficients of the polynomial $1 + \alpha^2 x + \alpha^3 x^2 + \ldots$

Finally, after finding the errors, or *roots*, of the error locator polynomial, we can now derive the error polynomial. We can use the *Forney algorithm* to find the coefficients of the error polynomial at the locations provided by the $\Lambda(x)$. First, we compute for the *formal derivative* of $\Lambda(x)$, which computationally is the same as that of the derivative but the coefficients are reduced modulo 929 *after* normal multiplication. Then, we derive the *error evaluator polynomial* $\Omega(x) = (S(x)\Lambda(x)) \bmod x^{2t}$, where $2t$ in the expression corresponds to the ecc

count. Then, we can compute for the jth coefficient of $e(x)$ by computing for $e_j = \Omega(X_j^{-1}) / \Lambda'(X_j^{-1})$. The resulting error polynomial can be subtracted from the message to get the corrected message.

Barcode Decoding

Once we have the corrected SCV list, we can extract the data codewords in it to get a new SCV list. You can think of this new list as sequential commands to some decoder that can either change its own mode or submode of processing or output a character or byte data. This decoder can change into one of the three modes of processing - numbers, alphanumeric characters, and byte data. For our scanner, it will work only in the *alphanumeric mode*. This mode consists of four submodes - alpha, lower, mixed, and punctuation. Each submode encodes up to 30 characters or submode commands.

Let us say we have the SCV list [87, 447, 146, 841, 184] resulting from reading the whole barcode and conversion of the read symbol characters. Each value actually contains *two* data values H and L, both having a value between 0 and 29 inclusive.

$$SCV = 30H + L$$

Decomposing each SCV in the example list yields [02, 27, 14, 27, 04, 26, 28, 01, 06, 04]. Note that H comes first before L. Also, if the last value of this list is 29, it is discarded, and hence, makes the length of the list odd. This list can now be used to decode the data.

Initially, the decoder will start in the alpha submode. Each submode contains *control values* that can make the decoder *shift* or *latch* to another submode. When shifting, the encoder will temporarily change its submode to the target and treat the immediate next value as encoded in that mode. After this value, the encoder reverts to its submode before the shift. Note that consecutive shifts are not allowed. On the other hand, when latching, the encoder will permanently change its submode to the target unless it will be changed again later on. This submode change can be modeled by a state machine shown in Annex B. Also, the table in Annex A shows the corresponding values and characters in each submode of the encoder. Using the two references, we can deduce that the example list encodes the string “CoE 164”.

Putting it All Together

Since the program is a bit hefty, you sought the help of another person that will write the module that you do not have the time to do so. Of course, if you finished your own module early, you can help your partner with theirs if they are having trouble with it. With limited time left, will you be able to finish the program in time for the IATA demo?

Input

The input to the module starts with a number T on a line indicating the number of barcodes. Then, each barcode is provided in two separate lines. The first line consists of a number E

indicating the error correction level in which the barcode is encoded, and a number N indicating the size of the scanned data. The next line consists of N integers denoting the SCV list itself.

Output

The output should consist of T blocks. Each block starts with a line containing the string "Case #T:". The next line contains a string of the format $N_E\ S$ with N_E denoting the number of corrected SCVs and S the corrected SCV list itself. Then, another line follows containing the decoded data from the corrected SCV list.

Example

Input
3
1 9
5 237 269 900 900 64 152 316 398
0 9
7 87 447 146 841 184 905 879 523
2 18
10 813 864 477 749 739 196 844 393 900 822 22 716 545 596 130 458 0

Output
Case #1:
0 5 237 269 900 900 64 152 316 398
Hi
Case #2:
1 7 87 447 146 841 184 900 879 523
CoE 164
Case #3:
4 10 893 864 877 749 739 496 844 393 900 822 22 716 545 596 130 458 768
(^_^)/--END

Additional Description/Requirements

You will be demonstrating this to the barcode group of IATA. Hence, you have the liberty to set-up the environment according to your desires. But for the purposes of easy set-up and demonstration, you have decided that the program should accept input from the standard input and print to the standard output (e.g. terminal) and that it should not contain any imports to libraries that need to be downloaded off the internet (e.g. through a package manager). After all, broadband was still relatively expensive for most households at that time.

You have decided that you only need to handle erroneous codewords and not erasures. Additionally, all of the data that will be read are correctable (i.e. the number of erroneous

codewords will not exceed the maximum allowable for its corresponding ecc level). The decoded data will not contain any other whitespace aside from the space character (ASCII code 32/040/0x20).

The barcodes that you will be reading will only accept these range or set of variables:
$T \leq 20$
$E \in [0, 8]$
$3 \leq N \leq 1000$

Upload your module(s) (as a single or multiple source code files; in TXT if the system does not support the file extension) to your remote repository in your division. The repository seems to hate compressed files, so make sure to not encase your codes in one.

The one who made the barcode decoder module will upload their module and acknowledge their partner on the submission notes and source code (by writing the name and student number of the partner). The one who made the error corrector module will also do the same for their module. If the modules have been merged into one program, just upload this merged program to your respective bins.

Grading Rubric

Barcode Decoder
5% SCV to high and low values
25% Value to character mapping
25% Submode latching and shifting

Error Corrector
5% Syndrome computation
25% Error locator polynomial computation
10% Root-finding of error locator polynomial
15% Computation for the error polynomial
5% Retrieval of original message

*Joint***
5% Input handling - able to read the input specifications
5% Output handling - able to output according to specifications
10% Self-documenting code
10% Acknowledgement of respective partner*
15% Both modules in one program

- 5% if they are in one module
- 15% if they are in one module and work together

* 5% if unacknowledged without or non-solid explanation, 0% if unacknowledged with solid explanation

** For sudden solo work due to other circumstances, there will either be a new partner assigned or a separate rubric will be applied. Solo workers can choose to implement either one or both modules.