1 Introduction and Project Goals

You and $n - 1$ of your friends have videotaped each other’s birthday parties since you were 5 years old. Now you have decided to store all of your tapes in digital format, but none of you has enough storage to do so! Luckily, you think of breaking the database of videos into fragments — each fragment having $1/n$ of the total data — and having each friend store one fragment on his or her laptop.

However, you and your friends tend to travel a lot! Naturally, when any one of you travels, that person takes their laptop along and the data on it is unavailable to the rest of you. This is just unacceptable! You never know when you will need to see one of those old home movies.

Luckily, you know that at any given time $m$ of the $n$ friends will be in town. Given this condition, we would like a way to give each of the friends some data that is only $1/m$ of the original database size, but also be able to reconstruct the entire database from any $m$ of the $n$ pieces of data given out. This requirement is met by Rabin’s Information Dispersal Algorithm (IDA).

In this project, you will implement Rabin’s Information Dispersal Algorithm. Along the way, you will be learning about the algorithm itself as well as refreshing your memory about rudimentary linear algebra and modular arithmetic. While it is not required that you look at it, Rabin’s paper on the algorithm can be found in The Journal of the ACM, Volume 36, Issue 2, pp. 335–348.

2 The Information Dispersal Algorithm

Fix integers $n$ and $m$, as well as a prime $p$. We will use $\mathbb{Z}_p$ to denote the integers modulo the prime $p$ (i.e., $\{0, 1, 2, \ldots, p - 1\}$).

IDA uses certain linear algebra operations done modulo a prime ($p$). The discussion sections will remind you how to do modular arithmetic and the matrix operations. We will restrict the rest of this section to simply covering the Information Dispersal Algorithm.

Typically, a file is stored as a sequence of bytes, and hence may be viewed as a sequence of integers, each between 0 and 511. In IDA, it is more convenient to work with data that is encoded as elements of $\mathbb{Z}_p$. You need not worry about how to translate data back and forth from a byte sequence to a sequence of elements in $\mathbb{Z}_p$, since we will provide code to do that.

For the sake of brevity, in this assignment we refer to a sequence of $\ell$ elements of $\mathbb{Z}_p$ as an $\ell$-sequence. In the following three sections of the algorithm description, all arithmetic operations (addition, multiplication, division, exponentiation) are done in $\mathbb{Z}_p$.

2.1 Encoding in IDA

In IDA encoding, the input is an arbitrary $\ell$-sequence — we call this $\ell$-sequence the “message”. The encoder outputs a sequence of size $n$ whose elements are $[\ell/m]$-sequences — we call these $[\ell/m]$-sequences “fragments of the message”. The $i$th fragment in the output sequence is said to have label $i$. The fragments have the property the message can be recovered from any $m$ of the $n$ fragments.
The encoding algorithm uses an \( n \) by \( m \) matrix, call it \( A \), with the property that any \( m \) rows of the matrix are linearly independent. (We will show you how to construct such a matrix in a later section.)

IDA looks at the message as a sequence of row vectors, \( \text{message} = c_0, c_1, c_2, \ldots, c_{\lceil \ell/m \rceil - 1} \), where each \( c_i \) has length \( m \). Since \( \ell \) may not be a multiple of \( m \), the encoder may have to pad the message with as many as \( m - 1 \) zeros to make the length a multiple of \( m \). Let \( c_i = (c_{i,0}, c_{i,1}, \ldots, c_{i,m-1}) \). The encoding process outputs the \( i \)th coordinate of \( A \cdot c_i^T \) as the \( j \)th element of the \( i \)th output fragment.

In other words, if

\[
A \cdot \begin{bmatrix} c_{i,0} \\ c_{i,1} \\ \vdots \\ c_{i,m-1} \end{bmatrix} = \begin{bmatrix} f_{0,i} \\ f_{1,i} \\ \vdots \\ f_{n-1,i} \end{bmatrix}
\]

then, the \( i \)th output fragment of the encoding process is \( (f_{0,i}, f_{1,i}, \ldots, f_{\lceil \ell/m \rceil - 1,i}) \).

### 2.2 Decoding in IDA

In IDA decoding, the input is a set of \( m \) distinct labeled fragments of some message \( c \), and the output is \( c \).

Suppose the decoder receives fragments \( f_{0,k}, f_{1,k}, \ldots, f_{\lceil \ell/m \rceil - 1,k} \). The decoder creates an \( n \) by \( m \) matrix \( B \), where the \( j \)th row of \( B \) is row \( j_i \) of \( A \) (the matrix used in encoding). The decoder also constructs \( \lceil \ell/m \rceil \) column vectors \( r_0, \ldots, r_{\lceil \ell/m \rceil - 1} \), each of size \( m \), such that \( r_k = (f_{0,k}, f_{1,k}, f_{2,k}, \ldots, f_{n-1,k}) \). The decoder outputs the \( j \)th element of \( B^{-1} \cdot r_k \) as the \( (mk + j) \)th element of the reconstructed message.

In other words, if

\[
B^{-1} \cdot r_k = B^{-1} \cdot \begin{bmatrix} f_{0,k} \\ f_{1,k} \\ \vdots \\ f_{n-1,k} \end{bmatrix} = \begin{bmatrix} c_{km} \\ c_{km+1} \\ c_{km+2} \\ \vdots \\ c_{km+m-1} \end{bmatrix}
\]

then the reconstructed message is \( c = (c_0, c_1, c_2, \ldots, c_{\lceil \ell/m \rceil - 1}) \).

### 2.3 Constructing the Encoding Matrix

There are several methods to construct an \( n \) by \( m \) matrix with entries drawn from \( \mathbb{Z}_p \) such that any \( m \) rows are linearly independent. In this project we will use what is quite possibly the simplest such method, a Vandermonde-type construction. Let the rows of matrix \( A \) be indexed from 0 to \( n - 1 \). For the Vandermonde-type construction, the \( i \)th row of matrix \( A \) is defined as

\[
1, (i + 1), (i + 1)^2, (i + 1)^3, \ldots, (i + 1)^{m-1}.
\]

Of course, the values in the preceding sequence are all taken modulo \( p \) since we are working over \( \mathbb{Z}_p \). As an ungraded but interesting math exercise, prove that any \( m \) rows of such a Vandermonde-type matrix are linearly independent.

### 3 Class Specification

In the specification, we will use the following vocabulary and variable names: \( \text{message} \) — the message to be encoded; \( \text{fragment} \) — one of the outputs of the encoding process; \( \text{fragmentID} \) — the label assigned to the fragment by the encoding process; \( n \) — the number of output fragments in encoding; \( m \) — the number of fragments needed for message reconstruction; \( p \) — the prime designating our modulus.
class IDA {

public IDA(int n, int m, int p)

    n -- the number of fragments to output
    m -- the number of fragments required for reconstruction
    p -- the prime designating the modulus to be used in calculations

    The IDA() function initializes your IDA object.

public int[][] encode(int[] message)

    message -- an array of integers, each guaranteed to be between 0 and p-1, that signify the message to be encoded.

    The encode() function encodes the given message and returns a two dimensional array, where the ith row of the returned array contains the ith output fragment. The length of message need not be a multiple of m. If it is not, the message should be padded with an appropriate number of zeros before proceeding with the encoding.

public void setLabels(int[] fragmentID)

    fragmentID -- an array of m integers, each between 0 and n-1 designating fragmentIDs of the m fragments that will be provided to a subsequent call of reconstructMessage().

    The setLabels() function provides a way for the user to notify your class which fragments will be provided to a subsequent call to decode(). The ith element of the fragmentID array designates the label of the ith fragment in the input to decode().

public int[] decode(int[][] fragment)

    fragment -- a table providing the fragment data to be used in decoding. Each fragment’s data is contained in a row of the table. The fragmentID of each row will have been provided in a preceding call to setLabels(). (You need not check the data for validity. For example, you may assume that each of the integers in the table is between 0 and p-1.)

    The decode() function decodes the original message using the provided fragments. The function returns the original message, so the output should only contain integers between 0 and p-1.

}

4 Turning in your project

You must develop your programs on the CS Linux machines. Your programs will be graded on the CS Linux machines. Any “portability” issue excuses will not be accepted.

For this project, you will turn in one file, IDA.java (your code). That file will have only one public class (the
IDA class specified above). It may also have a number of other non-public classes, depending on how you choose to
do your implementation.

Your code must compile successfully using only the commands:

javac IDA.java

If your program does not satisfy the above condition, you will receive no credit for the correctness portion of your
project’s evaluation.

The correctness of your program will be automatically evaluated based on the output of the encode() and decode()
methods of your IDA class.

Please follow the Project Protocol handout to turn in files. Use the Linux turn-in software to submit your project.
The command will look like: “turnin -submit ned Project_1 IDA.java”. To list the files you have
turned in, use the command “turnin -list ned Project_1”. All students will be turning in this project, but
not every project, to ned.

5 Questions and Project Clarifications

You may use the project newsgroup utexas.class.cs337 to ask your fellow classmates and the instructors for
clarifications on the project.

Instructor clarifications regarding the project will be posted to the project web page. You are responsible for any
modifications found there.