Ciphers

**Definition**: A cipher is a method used to encrypt and decipher encrypted information.

There are many ways to design a cipher.

**Mode 1: Scrambling**: Given an alphabet $A$, strings over $A$ may be encrypted as follows:

1. Select an invertible function $f(x) = px + q \mod |A|$
2. Replace a string $a_0a_1...a_n$ over $A$ with the string over $A$ $a_{f(0)}a_{f(1)}...a_{f(n)}$
Example

Given $A = \{a, b, c, d\}$ and $f(x) = 3x + 2 \mod 4$. The table of $f(x)$ is

<table>
<thead>
<tr>
<th>$x$</th>
<th>$f(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Consider the string “daba”

So, $d = a_0$, $a = a_1$, $b = a_2$, $a = a_3$

Since $a_{f(0)} = a_2 = b$, $a_{f(1)} = a_1 = a$; $a_{f(2)} = a_0 = d$; and $a_{f(3)} = a_3 = a$;

the encrypted (scrambled) string is “bada”
Another mode

Scrambling is independent of the alphabet. The next cipher, instead, starts by setting a one to one relation with the symbols in the given alphabet

**Mode 2:** Let $A$ be an alphabet. Then,

1. Select an invertible function $g: A \rightarrow \{0, 1, \ldots, |A| - 1\}$
2. Chose an invertible function $f(x) = px + q \mod |A|$
3. Replace a string $a_0a_1 \ldots a_n$ over $A$ with the string over $A$

$$g^{-1} f(g(a_0))g^{-1} f(g(a_1)) \ldots g^{-1} f(g(a_n))$$
Example

Given $A = \{a, b, c, d\}$ and $f(x) = 3x + 2 \mod 4$ and $g$ defined by the table

<table>
<thead>
<tr>
<th>$x$</th>
<th>$g(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>3</td>
</tr>
</tbody>
</table>

Consider the string “daba”

Since $g(d) = 3$, $g(a) = 0$, and $g(b) = 1$; $g(d)g(a)g(b)g(a) = 3010$. Since $f(3) = 3$, $f(1) = 1$ and $f(0) = 2$, we get

$f(g(d))f(g(a))f(g(b))f(g(a)) = 3212$. Finally, since $g^{-1}(3) = d$, $g^{-1}(2) = c$, and $g^{-1}(1) = b$, the encrypted string is “dcbc”
Deciphering

Decrypting messages requires the calculation of the inverse mappings of all mappings used in the encoding of the string. Since all these mod functions, the inversion technique discussed in the previous lecture becomes handy now. Applying this technique we get that the inverse of \( f(x) = 3x + 2 \mod 4 \) is \( f^{-1}(y) = 3y - 2 \mod 4 \). The table of this mapping is:

<table>
<thead>
<tr>
<th>( y )</th>
<th>( f(y) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Deciphering

For the scrambled message: “bada” we make $a_0 = b$, $a_1 = a$, $a_2 = d$, $a_3 = a$; and apply the inverse function to the indices to get $a_2a_1a_0a_3 = daba$

For the message encoded with the second method: the recipient must know $g$ and use the inverse of $f$ to apply it the received string “dcbc”, as follows:

$$g(f^{-1}(g^{-1}(d))) g(f^{-1}(g^{-1}(c))) g(f^{-1}(g^{-1}(b))) g(f^{-1}(g^{-1}(c))) =$$

$$g(f^{-1}(3)) g(f^{-1}(2)) g(f^{-1}(1)) g(f^{-1}(2)) =$$

$$g(3)g(0)g(1)g(0) =$$

$daba$
Definition: a fixed point in a mapping \( f \) is a value \( x \) such that \( f(x) = x \).

Example: Notice that \( f(x) = 3x + 2 \mod 4 \) has two fixed points: \( x = 1 \) and \( x = 3 \)

<table>
<thead>
<tr>
<th>( X )</th>
<th>( f(X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Avoiding fixed points

In general, a cipher with too many fixed points is easier to brake than one with few or no fixed points. Question is: Is there a cipher with no fixed points?

– If we replace \( f(x) = 3x + 2 \mod 4 \) with \( f(x) = 3x + 1 \mod 4 \) we get the following table:

<table>
<thead>
<tr>
<th>x</th>
<th>f(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

NO FIXED POINTS!!
Avoiding fixed points (cont.)

**Theorem:** Let \( n > 1 \) and let \( f(x) = ax + b \mod n \). Then, \( f \) has no fixed points if and only if the \( \gcd(a - 1, n) \) does not divide \( b \).

**Illustrations:**

- In the case of \( f(x) = 3x + 2 \mod 4 \) we have that \( \gcd(a - 1, n) = \gcd(2, 4) = 2 \). Since 2 divides \( b = 2 \), \( f \) has fixed points.

- In the case of \( f(x) = 3x + 1 \mod 4 \) we have that \( \gcd(a - 1, n) = \gcd(2, 4) = 2 \). Since 2 does not divide \( b = 1 \), \( f \) has no fixed points.
Hashing

The concepts:

• A table with n rows (n tuples), each holding some information, called **hash table**

• A key to access to, or store information in the rows of the table. This key is basically a mapping, called **hash mapping**

\[ f: S \rightarrow \{0, 1, \ldots, n-1\} \]

where S is the so-called **set of keys**
Why hashing?

Consider a simple problem: “A company wants to keep a consolidated and easy-to-access record of its employees. The company hires many temporary workers, so the personnel turn-around is significant. Also, the number of employees vary with the seasons. In summer time, the total amount of workers often doubles that of the winter season.”

How do you store such a large, ever-changing amount of information in a way such that employees’ data is readily available?
Employees record

An idea: What about storing the employees’ information in an array? Each row in this array will correspond to the employee’s social security number.

Sounds good in the sense that using an array guarantees constant time access to the information. Also, the social security number identifies the employee more accurately (no ambiguities) than his/her names.
However, there are some problems in this solution:

- Arrays are very limited in size
- Resizing (changing the size of) an array is done by copying the array’s information into a new array. This will be a drawback if resizing is needed with each new hiring (or firing)
- In order to have constant time access to the information it is necessary that the row addresses in the array coincide with the social security numbers. *This is highly unlikely***
Employees record (cont.)

Improved idea: Use the SSN “just to point” to an entry

Hash Table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Juan</td>
</tr>
<tr>
<td>1</td>
<td>Maria</td>
</tr>
<tr>
<td>2</td>
<td>Jose</td>
</tr>
<tr>
<td>3</td>
<td>Maria Jose</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>n - 1</td>
<td>Juan A.</td>
</tr>
</tbody>
</table>

Hash function(SSN of Jose) = 2 (this is, Jose’s entry in the table)
Pros and cons

The previous scheme solves some of the problems of the “array only” solution originally proposed

- Now, there is no need for aligning the SSN with the address in the array
- Provided that n is properly selected, there will be little or no need to resize the array each time a new employee is hired

But it introduces some problems of its own:

- The constant time access may be compromised: instead of direct addressing what we have is indirect addressing
- We have to design a hash function
Designing a hash function

A natural idea for the SSN is to use a mod function. Assume that we have 50 rows for the hash table. Then,

– Set \( n = 50 \)

– Define a hash function \( f: S \rightarrow \{0, 1, \ldots, 49\} \)

Now, for the design of the hash function we may choose (this is just one option):

\[
f(\text{SSN}) = \text{sum of the digits in SSN} \mod 50
\]
Using the hash function

Assume now that: Juan’s SSN is 788-12-1229, Jose’s is 992-93-5116, Maria’s is 842-78-8899, Maria Jose’s is 884-97-9369 and Juan A.’s 411-16-6171. Then, by using the previous hash function we get:

<table>
<thead>
<tr>
<th>SSN</th>
<th>Sum</th>
<th>f(SSN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>788-12-1229</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>992-93-5116</td>
<td>61</td>
<td>11</td>
</tr>
<tr>
<td>842-78-8898</td>
<td>62</td>
<td>12</td>
</tr>
<tr>
<td>884-97-9369</td>
<td>61</td>
<td>11</td>
</tr>
</tbody>
</table>

Collision!
Collisions

**Definition:** Whenever two different keys are map by the hash function into the same row in the hash table, we say that a *collision* (or data collision) has occurred.

Only an injective hash function has no collisions.

– In the case of our example, an injective hash function will require an array with \( n = \) total number of SSN’s rows. But, if we do so, we’ll face an unsolvable “too-big array” problem!!!
If a collision occurs, the natural thing to do is to use an unoccupied row (if any) to store the information associated with the colliding key. But, if we do this, the hash mapping is not longer taking us to the right place, at least for one of the keys. There is no way around this problem except using a search procedure. **Linear probing** is one such procedure. Now, the key is to be stored along with the object

\[
\text{LinearProbingFinding}(q, \text{key}, n)
\]

For \( j = 1 \) to \( n \)

If \( \text{key} = \text{key of object stored in row } q + j \mod n \)

Return object

Return “object not found”
Linear Probing (cont.)

Linear probing for storing:
   Similarly, we may search for an unoccupied row for storing new information

LinearProbingStore( q, key, n)
For j = 1, n
   If row q + j mod n is empty
     Store (key, object)
Return “Not enough space”

Illustration: Storing Jose’s information with Linear Probing
q = 11 is occupied
For j = 1
Check row 11 + 1 mod 50 = row 12
Row 12 is occupied
For j = 2
Check row 11 + 2 mod 50 = row 13
Row 13 is available: Store
Probe sequences

The linear probe can be replaced with a “gapped search” which searches the rows of the hash tables in strides of a fixed size. Let $1 \leq g < n$. Then the probe method is:

\[
\text{ProbeSeqSearch}(q, \text{key}, n) \\
\text{For } j=1, n \\
\quad \text{If } \text{key} = \text{key of object stored in row } q + j \cdot g \mod n \\
\quad \quad \text{Return object} \\
\quad \text{Return “object not found”}
\]

ProbeSeqStore is similar.
Illustration

**Definition**: The sequence $q + j \cdot g \mod n$; $j = 1, \ldots, n$ is called **probe sequence**

**Example**: In the example we had $q = 11$. By choosing $g = 5$ we generate the probe sequence:

$16, 21, 26, 31, 36, 41, 46, 1, 6, 11$

Thus, in this case, only 10 out of the 50 rows will be checked.

**Theorem**: The probe sequence has $n$ elements if and only if $\gcd(g, n) = 1$
Summary

• Ciphers
• Ciphers constructed with mod functions
• Fixed points
• Hashing concepts
• Motivations
• Collisions
• Linear Probing
• Probe sequences
Exercises

1. Consider the alphabet \{0, 1, 2, 3, 4, 5\}.
   a) List all possible ciphers of the form \(ax+b \mod n\)
   b) Among them list those with no fixed points
   c) Find the inverses of each cipher
   d) Select a cipher and encode the strings 401325 and 33344.
   e) Decipher the encoded strings using the inverse of the selected cipher
Exercises

2. Consider the 26 letters of the alphabet together with the set of symbols \{_, . , , , ;, :\} to complete a 31-character alphabet. On this alphabet, we propose the following cipher: (a) Associate a→0, b→1, ... , z→25, _→26, .→27, ,→28, ;→29, and :→30. (b) Define the cipher map as follows:

Given a string \( c_0c_1 \cdots c_k \), we define the mapping

\[
f(c_j) = 2(\text{number assigned to } c_j) + 1 \mod 31
\]

a) How can you make sure that this mapping is a bijection (as any cipher must be)?

b) Encode the string

“this_is,_i_believe,_a_complex_cipher.”

c) Devise a method for deciphering a string encoded with this cipher. Test it with the encoded string.
Exercises

3. Problems 11, 12 and 13, page 114 in the textbook