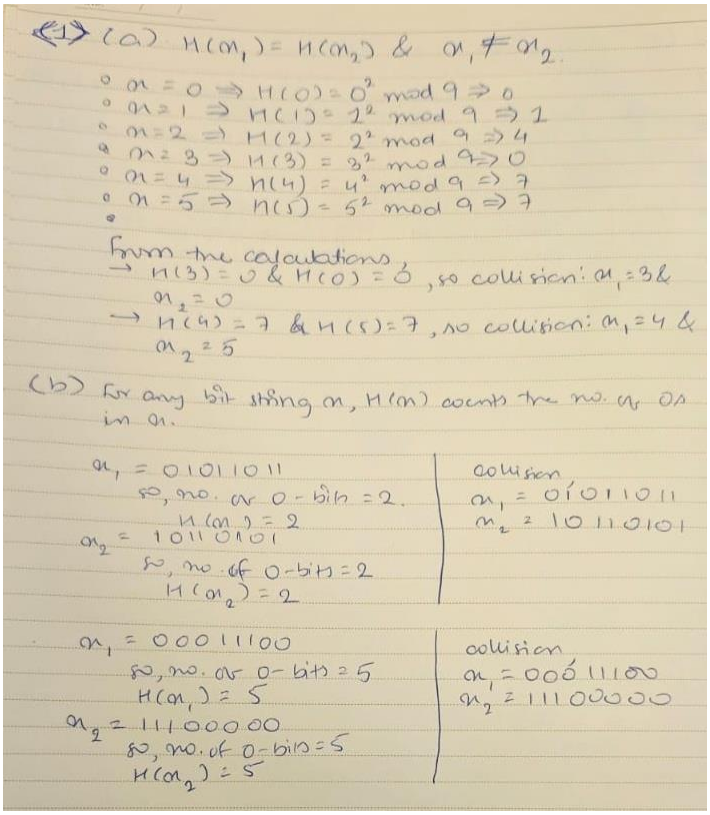
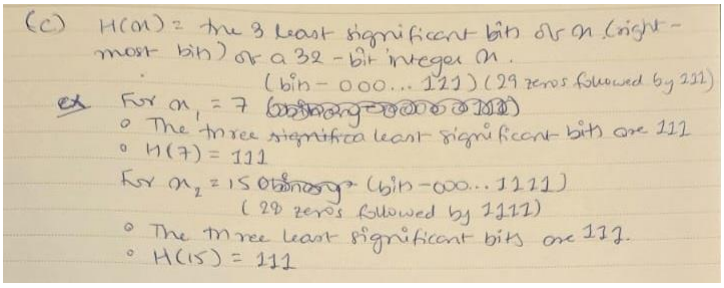
Homework 1

Submit your answers in a simple text file or as a link to a google doc with public access.

1. **Find a collision in each of the hash functions below:**
   a. **H(x) = x^2 mod 9, where x can be any integer**
   b. **H(x) = number of 0-bits in x, where x can be any bit string**
      i. **Note: a "bit string" is simply a sequence of 0s and 1s**
      ii. **For example, 01011011 is a bit-string**
   c. **H(x) = the three least significant bits of x, where x is a 32-bit integer.**
      i. **Note: "least significant bits" is the same as "least significant digits" but binary instead of decimal.**
      ii. **For example, for decimal number 384, what are the two least significant digits? 84.**
      iii. **Assume the least significant bits are on the right hand side, which means the number uses "big-endian" encoding, rather than "little endian" encoding. Read more about endianness here.**
         1. **The number 384 is the big-endian representation of three hundred, eighty-four.**
         2. **The number 483 is the little-endian representation of three hundred, eighty-hour.**





2. **Implement a program to find an x such that H (x ◦ id) ∈Y where**
   a. **H = SHA-256**
   b. **id = 0xED00AF5F774E4135E7746419FEB65DE8AE17D6950C95CEC3891070FBB5B03C78**
   c. **Y is the set of all 256 bit values that have some byte with the value 0x2F.**
   d. **Assume SHA-256 is puzzle-friendly. Your answer for x must be in hexadecimal.**
   e. **Here's a useful link for understanding binary encoding, decimal encoding, and hex encoding: https://www.rapidtables.com/convert/number/hex-to-decimal.html**
   f. **You must use a systems language like Java, Rust, or Golang.**
      i. **If you use Rust, use the following Rust crates:**
         1. **https://crates.io/crates/hex**
         2. **https://crates.io/crates/sha2**
         3. **https://crates.io/crates/rand**
   g. **Caution:**

i. The notation “x ◦ id” means the byte array x concatenated with the byte array id. For example, 11110000 ◦ 10101010 is the byte array 1111000010101010.

ii. The following two code segments are not equivalent:

INCORRECT	CORRECT

`let id_hex = "1D253A2F";`
 	
`if id_hex.contains("1D") {`
`  return;`
`}`
	`let id_hex = "1D253A2F";`
	
	`let decoded = hex::decode(id_hex).expect("Decoding failed");`
	`let u = u8::from(29); //29 in decimal is 0x1d in hex`
	`if decoded.contains(&u) {`
	`  return;`
	`}`

The second code segment above is the correct way to check whether 0x1D is a byte in 0x1D253A2F. Remember that hex format is only a way to represent a byte sequence in a human readable format. You should never perform operations directly on hex-string representations. Instead, you should first convert hex-strings into byte arrays, then perform operations on the byte arrays directly, and then convert the final byte array into a hex format when giving your answer. Performing operations directly on the hex strings is incorrect.

```
use sha2::{Digest, Sha256};
use rand::Rng;
use hex;

fn main() {
  // Step 1: Define the `id` as a byte array
  let id = hex::decode("ED00AF5F774E4135E7746419FEB65DE8AE17D6950C95CEC3891070FBB5B03C78")
    .expect("Decoding failed");

  // Step 2: Define a random number generator
  let mut rng = rand::thread_rng();

  // Step 3: Loop until we find a valid x
  loop {
    // Generate a random x (as a byte array of length 16)
    let x: Vec<u8> = (0..16).map(|_| rng.gen()).collect();

    // Concatenate x with id
    let mut data = x.clone();
    data.extend(&id);

    // Compute SHA-256 of the concatenated byte array
    let hash = Sha256::digest(&data);

    // Check if the hash contains a byte with value 0x2F
    if hash.iter().any(|&byte| byte == 0x2F) {
      // Convert the byte array x to a hexadecimal string and print it
      println!("Found x: {}", hex::encode(x));
      break;
```

```
    }
  }
}
```

3. **Alice and Bob want to play the game called "rocks-paper-scissors" over SMS text. Their game play is asynchronous in the sense that they can't expect the other person to be available at a certain time or within a certain time window. Design a protocol that enables Alice and Bob to play the game fairly and prevents the possibility of cheating. Provide a detailed explanation of the mechanism and why it works. An answer with insufficient detail will not receive credit. You should only need to use cryptographic hash functions to solve this problem. Keep the solution simple.**

To allow Alice and Bob to play Rock-Paper-Scissors asynchronously over SMS, we need a method that ensures fairness and prevents cheating. This can be achieved using cryptographic hash functions. The process has two phases: the Commitment Phase and the Reveal Phase.

1. Commitment Phase

1. Each player chooses their move:
   Both Alice and Bob independently select their moves (Rock, Paper, or Scissors). For example, Alice picks "Rock" and Bob picks "Scissors."
2. Add a random secret (salt):
   To protect their moves from being guessed, each player generates a random secret number (called "salt"). This salt is unique to each game.
   Example:
   - Alice chooses a secret like 1234.
   - Bob chooses a secret like 5678.
3. Compute a hash of the move and secret:
   Each player creates a hash by combining their move with their secret and applying a cryptographic hash function (e.g., SHA-256).
   - Alice computes H("Rock"||1234)H(\text{"Rock"} || 1234)H("Rock"||1234).
   - Bob computes H("Scissors"||5678)H(\text{"Scissors"} || 5678)H("Scissors"||5678).
4. Exchange hashes:
   Alice and Bob exchange their hashes via SMS. These hashes hide their actual moves but prove they've committed to a choice.

2. Reveal Phase

1. Players reveal their moves and secrets:
   Once both hashes have been exchanged, Alice and Bob reveal their actual moves and the corresponding secrets.
   - Alice sends "Rock" and 1234 to Bob.
   - Bob sends "Scissors" and 5678 to Alice.
2. Verify the hashes:
   Each player verifies that the other's revealed move and secret match the hash they received earlier. For example:
   - Alice checks if H("Scissors"||5678)H(\text{"Scissors"} || 5678)H("Scissors"||5678) matches Bob's hash.
   - Bob checks if H("Rock"||1234)H(\text{"Rock"} || 1234)H("Rock"||1234) matches Alice's hash.

3. Determine the Winner

After verifying the hashes, Alice and Bob apply the rules of Rock-Paper-Scissors to decide the winner:

- Rock beats Scissors.
- Scissors beats Paper.
- Paper beats Rock.
  For example, since Alice played "Rock" and Bob played "Scissors," Alice wins.

Why This Protocol Works

- Fairness: Neither Alice nor Bob can change their move after sending the hash because the hash is a one-way function.
- Prevents Cheating: The secret (salt) ensures that the move cannot be guessed from the hash.
- Asynchronous Play: The protocol allows Alice and Bob to play at their convenience without needing to be available at the same time.

Example Walkthrough

1. Commitment Phase:
    - Alice picks "Rock" and generates 1234. She computes: H("Rock"||1234)H(\text{"Rock"} || 1234)H("Rock"||1234) and sends the hash to Bob.
    - Bob picks "Scissors" and generates 5678. He computes: H("Scissors"||5678)H(\text{"Scissors"} || 5678)H("Scissors"||5678) and sends the hash to Alice.
2. Reveal Phase:
    - Alice reveals "Rock" and 1234 to Bob.
    - Bob reveals "Scissors" and 5678 to Alice.
3. Verification:
    - Alice checks if H("Scissors"||5678)H(\text{"Scissors"} || 5678)H("Scissors"||5678) matches the hash Bob sent earlier.
    - Bob checks if H("Rock"||1234)H(\text{"Rock"} || 1234)H("Rock"||1234) matches the hash Alice sent earlier.
4. Determine the Winner:
   Rock beats Scissors, so Alice wins.

This method ensures fairness and eliminates the possibility of cheating while allowing Alice and Bob to play asynchronously. It’s simple and secure, relying only on basic cryptographic principles.