1. Write a program for SHA-3 option with a block size of 1024 bits and assume that each of the lanes in the first message block (P0) has at least one nonzero bit. To start, all of the lanes in the internal state matrix that correspond to the capacity portion of the initial state are all zeros. Show how long it will take before all of these lanes have at least one nonzero bit. Note: Ignore the permutation. That is, keep track of the original zero lanes even after they have changed position in the matrix

Ans:
Code:

```
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#define LANE_COUNT 25
#define CAPACITY_LANES 12

uint64_t state[LANE_COUNT] = {0};
bool nonzero[LANE_COUNT] = {false};

void updateState(uint64_t input) {
    for (int i = 0; i < LANE_COUNT; i++) {
        if (input & (1ULL << i)) {
            state[i] ^= input;
            nonzero[i] = true;
        }
    }
}

int main() {
    uint64_t input = 0xFFFFFFFFFFFFFFFF;
    int rounds = 0;

    while (true) {
        updateState(input);
        rounds++;
        bool allNonzero = true;
        for (int i = 0; i < CAPACITY_LANES; i++) {
            if (!nonzero[i]) {
                allNonzero = false;
                break;
            }
        }

        if (allNonzero) {
            break;
        }
    }
```

```
    printf("All capacity lanes are nonzero after %d rounds.\n", rounds);
    return 0;
}
```

Out put:

```
Output

All capacity lanes are nonzero after 1 rounds.


=== Code Execution Successful ===
```

**2. Write a program for Diffie-Hellman protocol, each participant selects a secret number x
and sends the other participant ax mod q for some public number a. What would happen
if the participants sent each other xa for some public number a instead? Give at least one
method Alice and Bob could use to agree on a key. Can Eve break your system without
finding the secret numbers? Can Eve find the secret numbers?**

Ans:
Code:

```
#include <stdio.h>
#include <math.h>

long long power(long long base, long long exp, long long mod) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}

int main() {
    long long p = 23;
    long long g = 5;

    long long a_secret = 6;
```

```
    long long b_secret = 15;

    long long a_public = power(g, a_secret, p);
    long long b_public = power(g, b_secret, p);

    long long shared_key_a = power(b_public, a_secret, p);
    long long shared_key_b = power(a_public, b_secret, p);

    printf("Alice's Public Value: %lld\n", a_public);
    printf("Bob's Public Value: %lld\n", b_public);
    printf("Shared Key (Alice): %lld\n", shared_key_a);
    printf("Shared Key (Bob): %lld\n", shared_key_b);

    return 0;
}
```

Output:

3. Write a program for subkey generation in CMAC, it states that the block cipher is applied to the block that consists entirely of 0 bits. The first subkey is derived from the resulting string by a left shift of one bit and, conditionally, by XORing a constant that depends on the block size. The second subkey is derived in the same manner from the first subkey. a. What constants are needed for block sizes of 64 and 128 bits? b. How the left shift and XOR accomplishes the desired result.

Ans:
code:

```
#include <stdio.h>
#include <stdint.h>
```

```
#define BLOCK_SIZE 64
#define CONSTANT_64 0x1B

void left_shift(uint8_t *input, uint8_t *output) {
    uint8_t carry = 0;
    for (int i = BLOCK_SIZE / 8 - 1; i >= 0; i--) {
        output[i] = (input[i] << 1) | carry;
        carry = (input[i] & 0x80) ? 1 : 0;
    }
    output[BLOCK_SIZE / 8 - 1] &= 0xFE;
}

void generate_subkeys(uint8_t *key, uint8_t *k1, uint8_t *k2) {
    uint8_t temp[BLOCK_SIZE / 8];
    left_shift(key, k1);

    if (k1[0] & 0x80) {
        k1[BLOCK_SIZE / 8 - 1] ^= CONSTANT_64;
    }

    left_shift(k1, k2);

    if (k2[0] & 0x80) {
        k2[BLOCK_SIZE / 8 - 1] ^= CONSTANT_64;
    }
}

int main() {
    uint8_t key[BLOCK_SIZE / 8] = {0};
    uint8_t k1[BLOCK_SIZE / 8], k2[BLOCK_SIZE / 8];

    generate_subkeys(key, k1, k2);

    printf("K1: ");
    for (int i = 5; i < BLOCK_SIZE / 8; i++) {
        printf("%02X ", k1[i]);
    }
    printf("\nK2: ");
    for (int i = 7; i < BLOCK_SIZE / 8; i++) {
        printf("%02X ", k2[i]);
    }
    return 0;
}
```

Output:

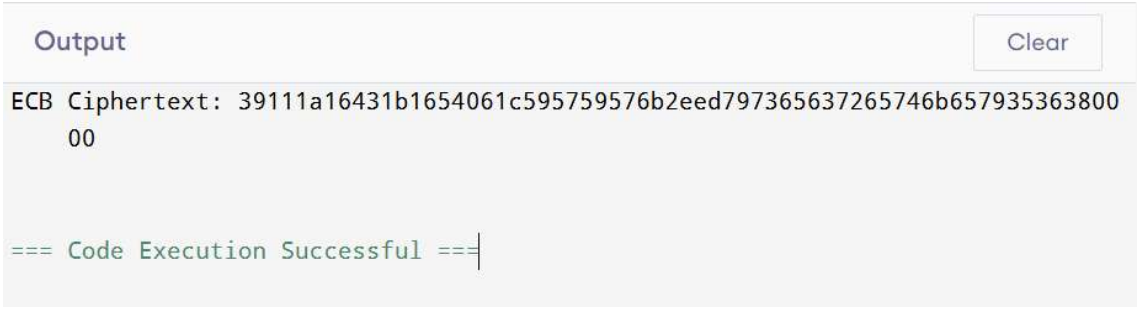
4. Write a program for ECB, CBC, and CFB modes, the plaintext must be a sequence of one
or more complete data blocks (or, for CFB mode, data segments). In other words, for these three modes, the total number of bits in the plaintext must be a positive multiple of
the block (or segment) size. One common method of padding, if needed, consists of a 1
bit followed by as few zero bits, possibly none, as are necessary to complete the final block. It is considered good practice for the sender to pad every message, including messages in which the final message block is already complete. What is the motivation
for including a padding block when padding is not needed

Ans:
Code:

```
#include <stdio.h>
#include <string.h>

#define BLOCK_SIZE 16

void ecb_encrypt(const unsigned char *plaintext, unsigned char *ciphertext, const unsigned
char *key) {

    for (int i = 0; i < strlen((const char *)plaintext); i += BLOCK_SIZE) {
        for (int j = 0; j < BLOCK_SIZE; j++) {
            ciphertext[i + j] = plaintext[i + j] ^ key[j];
        }
    }
}

void pad_plaintext(unsigned char *plaintext, int *length) {
    int padding_length = BLOCK_SIZE - (*length % BLOCK_SIZE);
    plaintext[*length] = 0x80;
    memset(plaintext + *length + 1, 0, padding_length - 1);
```

```
    *length += padding_length;
}

int main() {
    unsigned char key[BLOCK_SIZE] = "mysecretkey568";
    unsigned char plaintext[64] = "This is my book.";
    unsigned char ciphertext[64] = {0};
    int length = strlen((const char *)plaintext);

    pad_plaintext(plaintext, &length);
    ecb_encrypt(plaintext, ciphertext, key);

    printf("ECB Ciphertext: ");
    for (int i = 0; i < length; i++) {
        printf("%02x", ciphertext[i]);
    }
    printf("\n");

    return 0;
}
```

Output:



5. Write a program for Caesar cipher, known as the affine Caesar cipher, has the following
form: For each plaintext letter p, substitute the ciphertext letter C: C = E([a, b], p) = (ap +
b) mod 26 A basic requirement of any encryption algorithm is that it be one-to-one. That
is, if p q, then E(k, p) E(k, q). Otherwise, decryption is impossible, because more than one plaintext character maps into the same ciphertext character. The affine Caesar cipher

is not one-to-one for all values of a. For example, for a = 2 and b = 3, then E([a, b], 0) = E([a, b], 13) = 3.

Ans:
Code:

```
#include <stdio.h>
```

```
char encrypt(char p, int a, int b) {
    return (a * (p - 'A') + b) % 26 + 'A';
}

int main() {
    int a = 5;
    int b = 8;
    char plaintext[] = "this is my book";
    char ciphertext[sizeof(plaintext)];

    for (int i = 0; plaintext[i] != '\0'; i++) {
        ciphertext[i] = encrypt(plaintext[i], a, b);
    }
    ciphertext[sizeof(plaintext) - 1] = '\0';

    printf("Ciphertext: %s\n", ciphertext);
    return 0;
}
```

Output:

6. Write a program for CBC MAC of a one block message X, say T = MAC(K, X), the adversary immediately knows the CBC MAC for the two-block message X || (X ⊕ T) since this is once again.
Ans:
Code:

```
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#define BLOCK_SIZE 16
void xor_blocks(uint8_t *block1, const uint8_t *block2) {
    for (int i = 0; i < BLOCK_SIZE; i++) {
        block1[i] ^= block2[i];
    }
}
```

```
void encrypt_block(const uint8_t *key, const uint8_t *input, uint8_t *output) {

    memcpy(output, input, BLOCK_SIZE);
}

void cbc_mac(const uint8_t *key, const uint8_t *message, uint8_t *mac) {
    uint8_t block[BLOCK_SIZE] = {0};
    uint8_t encrypted[BLOCK_SIZE];

    xor_blocks(block, message);
    encrypt_block(key, block, encrypted);
    memcpy(mac, encrypted, BLOCK_SIZE);
}

int main() {
    uint8_t key[BLOCK_SIZE] = {0x16};
    uint8_t message[BLOCK_SIZE] = {0x69};
    uint8_t mac[BLOCK_SIZE];

    cbc_mac(key, message, mac);

    printf("CBC MAC: ");
    for (int i = 0; i < BLOCK_SIZE; i++) {
        printf("%02x", mac[i]);
    }
    printf("\n");

    return 0;
}
```

Output: