Q1:

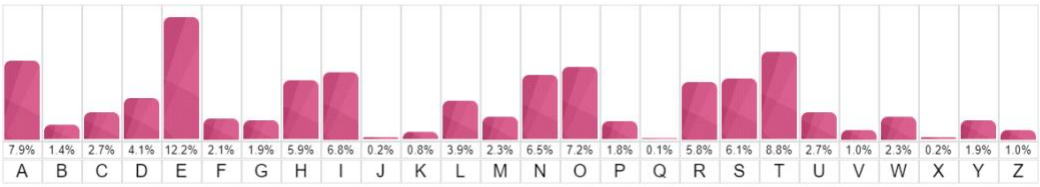
1.1: Ciphertext: “lhto iael erurcsoaelpd nilr ratknhgoseredfe socicoredtze seniBc nyv cadgoctan”

1.2: Plaintext: “The greatest glory in living lies not in never falling but in rising every time we fall Nelson Mandela”

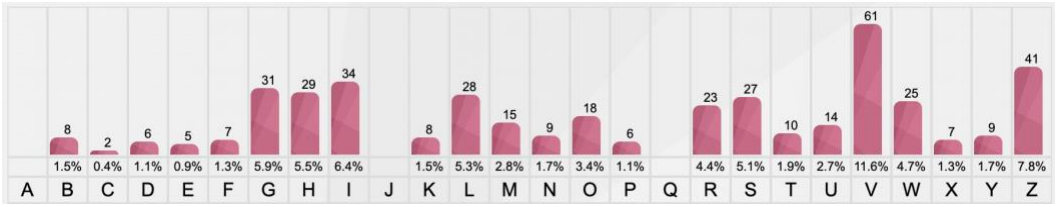
(p.s.: For the plaintext, blanks are added for better understanding the context.)

Q2:

According to the following chart, which shows the frequency of each letter of the alphabet for the English language:



Here is the frequency analysis outcome of the given message:



Thus, according to the frequencies, try:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	A	I	H	G	F	E	D	C	B	J

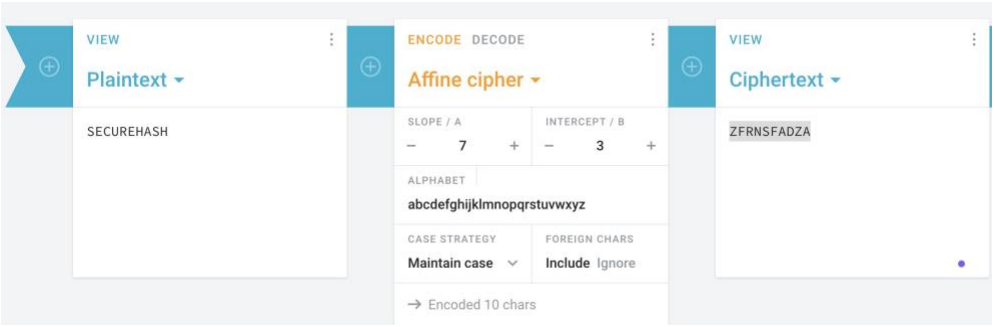
As in the given message, “V” got the highest frequency, so we assume it is allocated to “E”. And for “Z” in the message, we assume it is “A” given that “A” is the second frequent letter in English. The rest characters are deducted through the same methods, with a little bit of brute force using common digraphs, since there are some of the characters are not showed in the message.

Thus, the original plain text is:

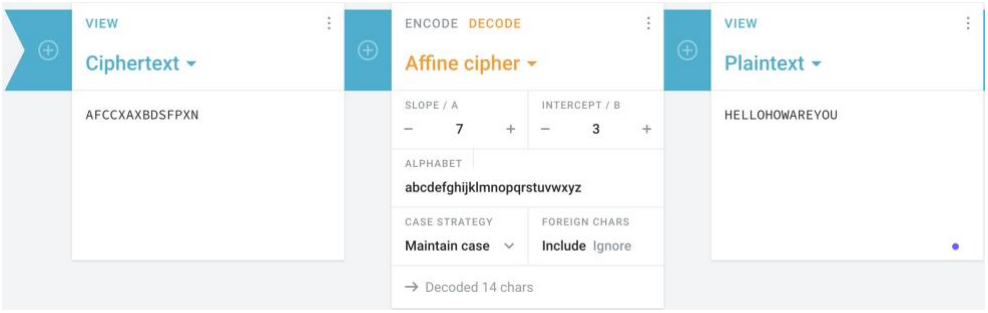
A long time ago, in a galaxy far, far away... It is a dark time for the Rebellion. Although the Death Star has been destroyed, Imperial troops have driven the Rebel forces from their hidden base and pursued them across the galaxy. Evading the dreaded Imperial Starfleet, a group of freedom fighters led by Luke Skywalker has established a new secret base on the remote ice world of Hoth. The evil lord Darth Vader, obsessed with finding young Skywalker, has dispatched thousands of remote probes into the far reaches of space…

Q3:

3.1: ZFRNSFADZA



3.2: HELLOHOWAREYOU



3.3: Pseudocode of Affine Cipher

Encryption:

```
E (x) = (a x + b) mod m

modulus m: size of the alphabet

a and b: key of the cipher.

a must be chosen such that a and m are coprime.
```

Decryption:

```
D (x) = a^-1 (x - b) mod m

a^-1: modular multiplicative inverse of a modulo m. i.e., it satisfies the equation

1 = a a^-1 mod m.
```

Q4:

4.1: This hash function is not collision resistant.

The hash function $H(x) = (x \bmod p) + (x \bmod q)$, where p and q are large prime numbers, does not seem to be collision resistant. Consider two different inputs x_1 and x_2, and we need to check if $H(x_1) = H(x_2)$. Specifically, we are looking for a case where:

$$(x_1 \bmod p) + (x_1 \bmod q) = (x_2 \bmod p) + (x_2 \bmod q)$$

A simple approach to find a collision is to consider numbers that are multiples of both p and q. For example, suppose $p = 5, q = 7,$ *and* let $x_1 = 0$, $x_2 = 35$ then we have:

$$H(x_1) = H(0) = (0 \bmod 5) + (0 \bmod 7) = 0 + 0 = 0$$

$$H(x_2) = H(35) = (35 \bmod 5) + (35 \bmod 7) = 0 + 0 = 0$$

In this case, we have a collision where $H(0) = H(35)$, even though $0 \neq 35$. Therefore, this hash function is not collision-resistant because we can find different inputs x_1 and x_2, such that $H(x_1) = H(x_2)$.

4.2: In blockchain technology, hash functions are used to ensure the integrity of data. Each block contains a hash that uniquely represents its contents. If a collision occurs, it means two different sets of data could produce the same hash. This can be exploited by an attacker.

Example scenario of malicious use:

Let's say the blockchain stores transaction records in blocks, and the hash of each block is calculated using the function $H(x)$. Each block includes:

- The previous block's hash (to ensure the chain's integrity)
- A set of transactions

- A nonce

If an attacker finds two different inputs, x_1 and x_2, such that $H(x_1) = H(x_2)$, they can potentially create two blocks with the same hash, but with different transaction data.

Here's how the attacker could exploit this:

1. The attacker creates two versions of a block:
   a) Block A with legitimate transactions x_1
   b) Block B with altered or fraudulent transactions x_2
2. Since $H(x_1) = H(x_2)$, the hash of both blocks will be the same.
3. The attacker can present Block A to the blockchain network for validation, and once it's confirmed, they can later replace Block A with Block B, as both have the same hash. This allows the attacker to alter the contents of the blockchain without breaking the hash chain, effectively rewriting transaction history.

For example:

- Imagine a transaction where Alice sends 10 coins to Bob (in Block A).
- The attacker creates a collision (Block B), where the transaction is altered to Alice sending 10 coins to herself.
- After the block with the legitimate transaction is added, the attacker can replace it with the fraudulent block.

This type of attack undermines the trust in the blockchain, as the hash no longer guarantees the uniqueness or integrity of the data.

Q5:

```
// Smart Contract for Rental Payments
contract RentalAgreement {
    address tenant;                 // Alice
    address payable landlord; // Bob, must be payable to receive funds
    uint rentAmount;
    uint dueDate;                   // Monthly due date in Unix timestamp
    uint penaltyAmount;
    bool isPaid;
    // Constructor to initialize the contract
    constructor(address _tenant, address payable _landlord, uint _rentAmount,
                    uint _dueDate, uint _penaltyAmount) {
        tenant = _tenant;
        landlord = _landlord;
        rentAmount = _rentAmount;
        dueDate = _dueDate;
        penaltyAmount = _penaltyAmount;
        isPaid = false;
    }
    // Function to make rent payment
    function payRent() public payable {
        require(msg.sender == tenant, "Only the tenant can pay the rent.");
        // 5.1: Ensure rent is paid only once per month
```

```
        require(!isPaid, "Rent for this month has already been paid.");
        // 5.2: Check if payment is late
        if (block.timestamp > dueDate) {
            // Apply penalty for late payment
            uint totalAmount = rentAmount + penaltyAmount;
            require(msg.value == totalAmount, "Incorrect payment amount with penalty.");
            landlord.transfer(totalAmount); // 5.4
        } else {
            // Transfer rent to landlord without penalty
            require(msg.value == rentAmount, "Incorrect payment amount.");
            landlord.transfer(rentAmount);
        }
        isPaid = true;
    }
    // 5.3: Reset payment status at the start of each month
    function resetPaymentStatus() public {
        require(block.timestamp > dueDate && isPaid,
                "Can only reset at the start of a new month after payment is made.");
        isPaid = false;
        // Update due date to the next month
        dueDate += 30 days;
    }
    // Function to check if rent is paid
    function checkPaymentStatus() public view returns (bool) {
        return isPaid;
    }
}
```

5.4: Instead of modifying “rentAmount” directly, use a temporary variable or calculate the total payment dynamically when a late payment occurs. This ensures that the penalty applies only for the current month, and the “rentAmount” remains unchanged for future payments.