System Design for Mono-alphabetic Substitution Cipher:

We got Project 0 to apply mono-alphabetic substitution cipher : The mono substitution cipher is a cipher in which pair of letters of the plain text are replaced by a corresponding pair from a fixed key.

The user has **2 choices** whether they want to **input their own plain text** or the user wants to **generate a random string**.

```
if choice == "1":
    print("random string to be generated, enter length of the string:")
    length_s = int(input().strip())
    if length_s % 2 == 0:
        user_text = generate_random_string(length_s)
    else:
        print("length should be even, restart......")
        quit()
elif choice == "2":
    user_text = get_plain_text()
else:
    print("Invalid choice")
    quit()
```

Note the strings defined in both the cases should be **even in length** and should only have characters defined in the **character set**. This is done by functions def check(p_text, character_set) and get_plain_text().

```
# function to check whether the entered string is valid or not.
def check(p_text, character_set):
    # only takes in strings with even length
    if len(p_text) % 2 != 0: return False
    # string should only have characters defined in the character set
    return all(c in character_set for c in p_text)
```

```
#function to get plain_text as an user input.
def get_plain_text():
    while True :
        print("Enter the plain text\nPlain text should only contain {A,B,C} and
        character_set = "ABC"
        p_text = input().strip()
        if check(p_text, character_set):
            return p_text
        else :
            continue
```

After the user_text is generated, it is hashed using the hash function.
Note: **SHA256 is used to hash** the given string, after hashing a hex string is generated where each hexadecimal value is **mapped to the character set**. So the **final hash** generated is always 64 in length.

```
def hash(user_text, toPrint):
    hash_256 = hashlib.sha256(user_text.encode('utf-8')).hexdigest()
    hashstring = ""

    # MATCHING A = 0 - 5, B = 6 - 10, C = 11 - 16

    for b in hash_256:
        dec = int(b, 16)
        if dec <= 5 : hashstring += 'A'
        elif dec <= 10 : hashstring += 'B'
        else : hashstring += 'C'
    if toPrint:
        printt("hash: ", hashstring)
    #printt("length: ", len(hashstring))
    return hashstring
```

After getting the hash of the user_generated_text, we **concatenate user_text and its hash to form the plain_text**. Any plain_text generated would always have their last 64 characters as the hash of the orignal_text. After getting the plain_text, we need a key to encrypt it. So we generate a key **using the get_key() function**. The get_key() function returns a dictionary of key value pairs, which are used for **substitution** in plain_text during encryption.

```
# function returns a random key
def get_key():
    p = ['AA', 'AB', 'AC', 'BB', 'BA', 'BC', 'CC', 'CA', 'CB']
    e_p =  ['AA', 'AB', 'AC', 'BB', 'BA', 'BC', 'CC', 'CA', 'CB']
    # none of the key value should match check
    random.shuffle(e_p)
    # distribute
    key = {}
    for idx, p_ in enumerate(p):
        key[p_] = e_p[idx: idx+1][0]
    return key
```

After getting the key and the plain_text, we call the **encrypt function** which returns the **cipher text**, substituting the plain_text on the basis of the generated key.

```python
def encrypt(plain_text, key):
    c_text = ""
    for i in range(0, len(plain_text), 2):
        c_text += key[plain_text[i: i+2]]

    return c_text
```

As this is a **symmetric-key cryptographic system** the same key is used to decrypt the cipher_text. We use the **decrypt function** which gives us the decrypted_plain_text.

```python
def decrypt(cipher_text, key):
    key = dict((values,key) for key,values in key.items())
    decrypted_text = ""
    for i in range(0, len(cipher_text), 2):
        decrypted_text += key[cipher_text[i: i+2]]
    return decrypted_text
```

In all cases the plaintext that we work with should be "recognizable". To make the text recognizable, the plaintext p should satisfy some property, π. Here the last 64 characters of the plain_text will be the hash of the orignal_text. So here pi property is such that for any plain_text, we take the last 64 characters which represents the hash of the orignal_string let this be hash_orignal and rest of the plain_text represents the orignal_text. Now we hash(orignal_text) and check if it matches the hash_orignal. This is implemented in the checkMatch() function.

```python
# to check if the hash of decrypted_user_text matches the hash.
def checkMatch(decrypted_text):
    n = len(plain_text)
    decrypted_user_text = decrypted_text[0:n-64]
    decrypted_hash = decrypted_text[-64:]
    print("hashing the decrypted_user_text to check if it matches the decrypted_hash")
    if decrypted_hash == hash(decrypted_user_text, 0): printt("Its a match, after hashing the user_text: ", hash(decrypted_user_text, 0))
    else : print("Wrong Hash")
```

BruteForce:

In the brute_force function we generate a list of dictionaries which has all the permutations possible for the key. Now for each key in dics, we use them to decrypt the given ciphertext, the plain_text generated by the decrypt function is now checked for pi property, if it satisfies the property the brute force attack worked, and we get the desired key which was originally used.

```python
# function to perform brute force to find key
def brute_force(ciphertext):
    p = ['AA', 'AB', 'AC', 'BB', 'BA', 'BC', 'CC', 'CA', 'CB']
    e_p =  ['AA', 'AB', 'AC', 'BB', 'BA', 'BC', 'CC', 'CA', 'CB']
    dics = []
    for perm in itertools.permutations(p, len(e_p)):
        dics.append(dict(zip(perm, e_p)))
    print("total possible keys: ", len(dics))
    keysChecked = 0
    for d in dics:
        plain_text = decrypt(ciphertext, d)
        if checkMatchBruteForce(plain_text):
            print("plaintext statisfies the pi property. key found")
            print("Key Found: ", d)
            printt("total keys checked: " , keysChecked)
            break
        keysChecked += 1
    return "Not Found"
```