密碼工程 Quiz4

學號：113550021 姓名：陳孟楷

Problem 1

Question 1-1.

I wrote four yaml files based on the templates given in the GitHub. Each yaml file also generates multiple pem files.

- root_cert.yml generates root.cert.pem and root.key.pem.
- intermediate.cert.yml generates intermediate.cert.pem, intermediate.csr.pem, and intermediate.key.pem
- server.cert.yml generates server.cert.pem, server.csr.pem, and server.key.pem
- client.cert.yml generates client.cert.pem, client.csr.pem, and client.key.pem

Question 1-2.

1. In root.cert.yml, I entered "**cert-go create cert -y ./root/root.cert.yml -t root**".
2. In intermediate.cert.yml, I entered "**cert-go create cert -y ./intermediate/intermediate.cert.yml -t intermediate**".
3. In server.cert.yml, I entered "**cert-go create cert -y ./server/server.cert.yml -t server**".
4. In client.cert.yml, I entered "**cert-go create cert -y ./client/client.cert.yml -t client**".

Question 1-3.

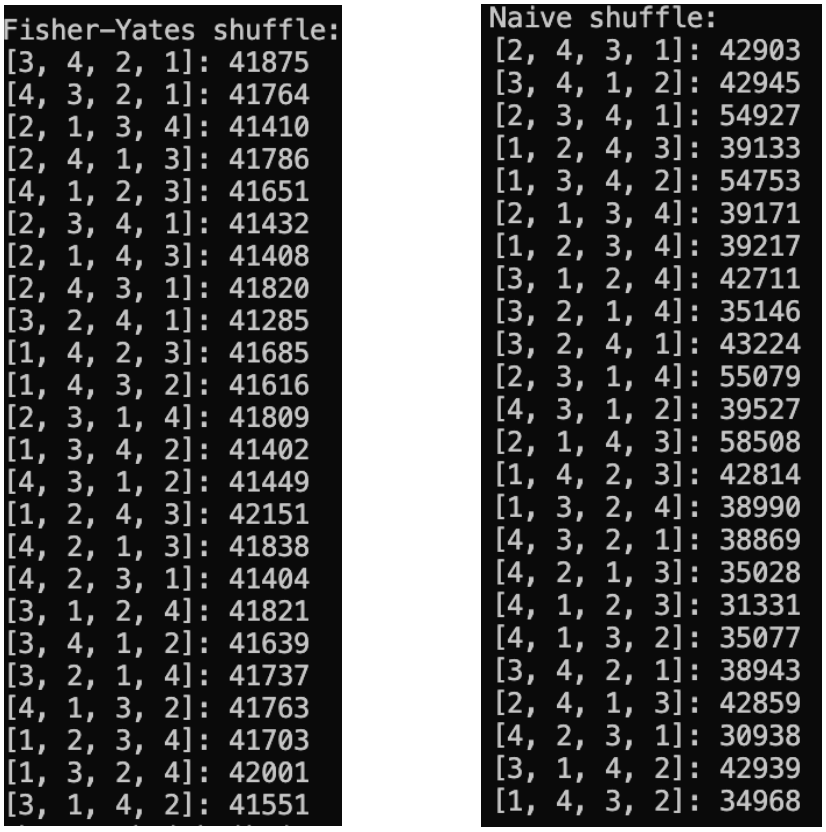
先用 root 的自簽憑證作為 root certificate，再簽發 intermediate certificate, 最後藉由 intermediate certificate 發行 server certificate 和 client certificate.

Problem 2

Question 2-1.

```python
import random
from collections import Counter
from itertools import permutations

cards = [1, 2, 3, 4]
naive_count = Counter()
fy_count = Counter()

def naive_shuffle(cards):
    arr = cards[:]
    for i in range(len(arr)):
        n = random.randint(0, len(arr)-1)
        arr[i], arr[n] = arr[n], arr[i]
    return tuple(arr)

def fisher_yates(cards):
    arr = cards[:]
    for i in range(len(arr)-1, 0, -1):
        n = random.randint(0, i)
        arr[i], arr[n] = arr[n], arr[i]
    return tuple(arr)

for _ in range(1000000):
    naive_count[naive_shuffle(cards)] += 1
    fy_count[fisher_yates(cards)] += 1

print("Naive shuffle:")

for k, v in naive_count.items():
    print(f"{list(k)}: {v}")

print("\nFisher-Yates shuffle:")
for k, v in fy_count.items():
    print(f"{list(k)}: {v}")
```

Python code



Fisher-Yates shuffle and Naïve shuffle

Question 2-2.

Fisher-Yates 洗牌演算法明顯較佳，原因如下：

- 它保證每一種排列的機率完全相同，是數學上「公平」的洗牌方式。
- 每次交換的目標是從尚未排定的元素中挑選，有效避免重複與偏差。
- 使用 O(n)O(n) 時間與 O(1)O(1) 額外空間即可完成。

相對來說，Naive 演算法可能會多次將元素交換回去原本的位置，導致某些排列機率較高、不均勻。

Question 2-3.

Naive 洗牌演算法的缺點如下：

- 每次交換時都從完整陣列中挑選目標，導致已排序的元素可能被「重新破壞」。

- 對於小樣本（如 4 張牌）尚能接受，但對於大樣本或安全用途，其偏差會明顯累積。
- 它無法保證每一種排列出現機率相同，因此會造成偏斜分布，不適合用於密碼學或機率模擬。

Question 2-4.

- 改進 KSA 隨機性：將 KSA 進行多輪混合（例如多輪 key 擴展與反覆打亂），或引入非線性轉換以打破偏差。
- 丟棄前段 keystream（RC4-drop）：由於前幾個輸出有統計偏差，建議跳過前 256～3072 bytes 的密鑰流。
- 加鹽機制（salt）：在 key 輸入前加入隨機 salt，可防止相同 key 導致相同輸出。
- 整體替代 RC4：RC4 屬於較舊的流加密演算法，建議使用更安全的演算法（如 ChaCha20、AES-CTR）來取代。

Problem 3

Question 3-1.

當我們將 Miller-Rabin 測試用在 RSA 的模數$n = pq$（其中p,q為兩個大質數）時，它有很高的機率會誤判n是質數，即使它實際上是合數。這是因為：

- Miller-Rabin 是一種 機率性質數測試，可以快速檢查一個數是否有「非質數的特徵」。
- 然而像 RSA 模數這樣的數，只由兩個非常大的質數相乘構成，其組成結構讓它不像一般的合數那樣容易被發現異常。
- 因此，在多次選擇隨機底數進行測試的情況下，RSA 模數仍可能通過測試，被誤判為質數。

Question 3-2.

不能，因為：

- RSA 的安全性是建立在「無法有效分解一個非常大的合數$n = pq$」這個數學難題上。
- 而 Miller-Rabin 測試的功能是「判斷一個數是不是質數」，它無法幫助我們找出 n 的質因數 p、q。

- 即使 Miller-Rabin 誤以為 n 是質數，這個資訊也不會幫助我們破解 RSA，因為知道一個數是不是質數，跟能不能把它分解，是兩回事。