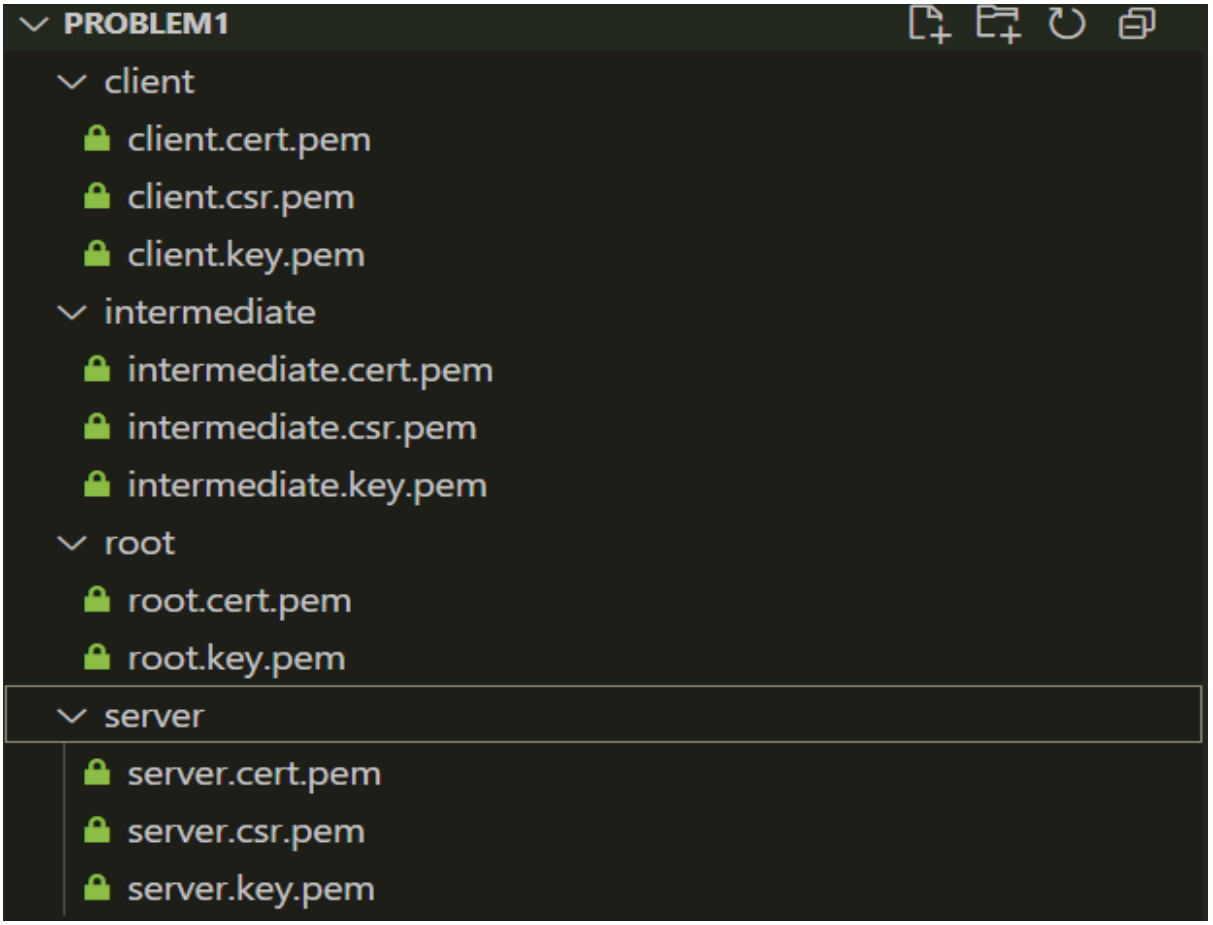
Cryptography Engineering Quiz 4 多工所 313553024 蘇柏叡

Problem 1

1.1

我們生成之後的檔案結構如下，符合本題要求。此外，關於如何生成的，我們下的command line將會在1.2呈現。



1.2

2. **Description of Commands:**
   - Describe the commands you used in the previous section and list them in the order they were executed, using numeric order.

Ans:

首先我們建立了名為problem1的資料夾，並且將下載完的執行檔案（cert-go）放置在problem1底下，接著我們建立了四個子資料夾 root、intermediate、server、client用以儲存後續輸出的私鑰（.key.pem）、簽署請求（.csr.pem）以及憑證（.cert.pem）檔案。接著，我們參考官方原始碼與預設設定檔 defaultCfg.yaml，編寫了自訂的config.yaml，並依照作業規範將憑證的organization欄位統一設為 "NYCU-CE"、調整Common Name，以及指定各項路徑與簽署鍵結構。

以下為執行並生成.pem檔案時，使用的command line(需要按照順序)，這些command line會依據config.yaml自動依序產生Root、Intermediate、Server與Client憑證與私鑰，並將所有.pem檔案依照路徑輸出至對應的資料夾中。

(1) 建立Root憑證：

./cert-go create cert --type root --yaml config.yaml

(2) 建立Intermediate憑證：

./cert-go create cert --type intermediate --yaml config.yaml

(3) 建立Server憑證：

./cert-go create cert --type server --yaml config.yaml

(4) 建立Client憑證：

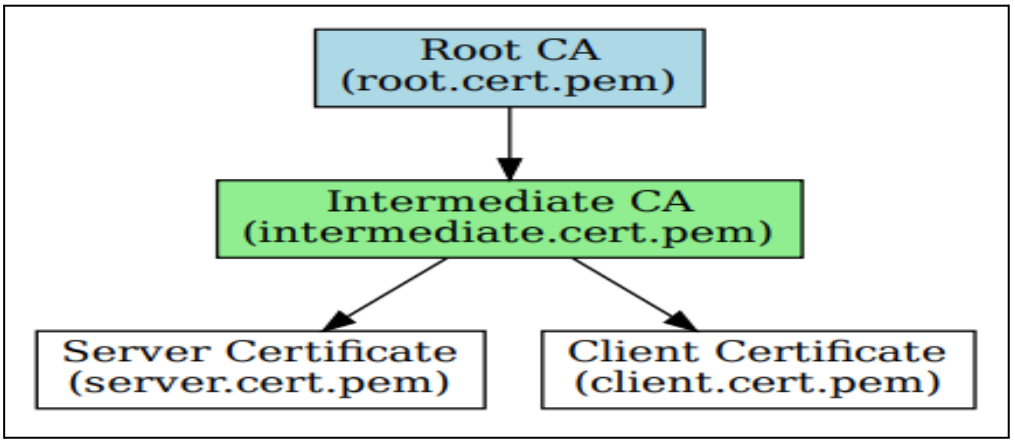
./cert-go create cert --type client --yaml config.yaml

1.3

3. **Certificate Chain Description:**
   - Describe your certificate chain structure.

Ans:

下圖為我們使用的Certifcate Chain Structure，在整個結構的最上層為Root CA，其為自簽憑證，亦即憑證的簽發者與主體為同一實體。而Root CA負責簽署下層的Intermediate CA，並作為整體信任來源的根據。Intermediate CA為由Root CA所簽發的中繼憑證，主要用途在於連接Root憑證與最終實體所使用的憑證，並作為實際簽發伺服器端與用戶端憑證的主體。透過此層級設計，可以在不直接暴露Root憑證的情況下完成所有實體的簽章程序，進一步提升系統的安全性與可控性。至於Server與Client所使用的憑證則是皆由Intermediate CA所簽發，並分別應用於伺服器端與用戶端的身分驗證流程中。



1.4

4. **(Extra) Repository Improvement:**
   - Check for any areas of improvement in the `cert-go` repository.
   - If you find any, try to create a pull request.

Ans:

我們想針對**--force**的部分，提出使用上曾出錯並且可以在之後進行改進的建議。**如下圖**，若是我們想要針對已存在的private key、csr、certifcate直接做覆蓋時，會無法使用。因此，我們建議作者可以針對此部分進行修復並且在下次版本更新時，進行調整。並且確保README上面提供的flags是都可以正確使用的。

```
vboxuser@aaa:~/Desktop/problem1$ ./cert-go create cert --type client --yaml config.yaml --force
Error: unknown flag: --force
Usage:
  cert-go create cert [flags]

Flags:
  -h, --help          help for cert
  -t, --type string   specify the type of the certificate: [root, intermediate, server, client]
  -y, --yaml string   specify the configuration yaml file path
```

Problem 2

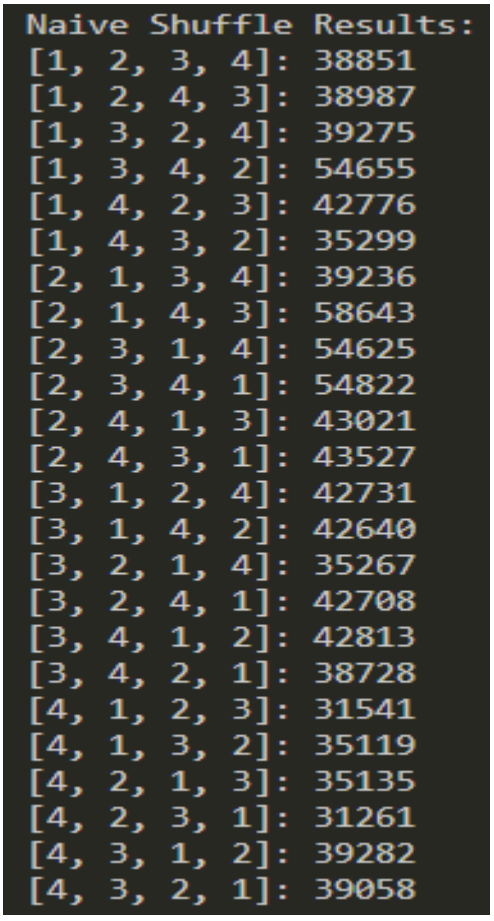
2.1

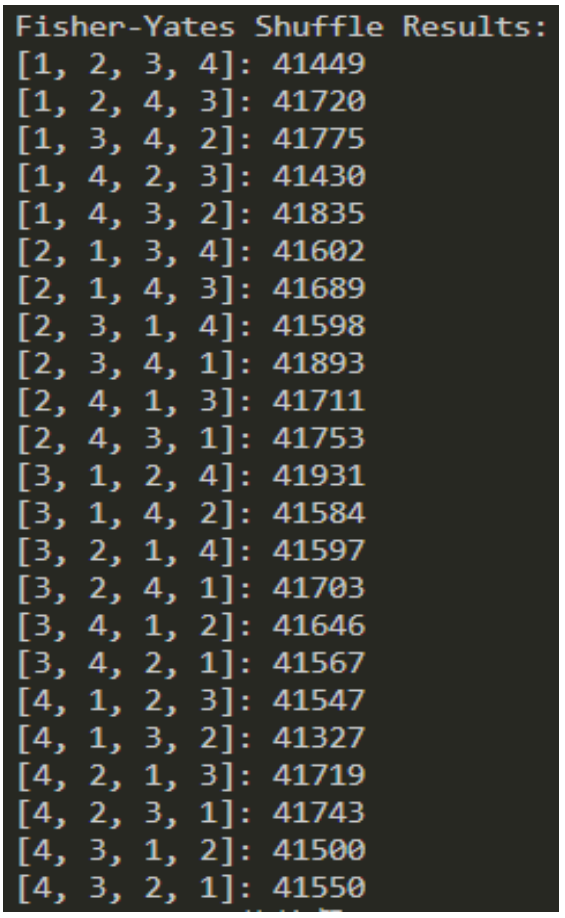
Write a Go/Python program to simulate two algorithms with a set of 4 cards, shuffling each a million times. Collect the count of all combinations and output. We focus on "distribution", so the card sets' order does not need to be in lexicographical order and is accepted as is.

Ans:

程式碼撰寫如下:

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

def naive_shuffle(cards):
    """Naive Shuffle: 每次都從整個 range(0, len(cards)-1) 取隨機索引並交換。"""
    for i in range(len(cards)):
        n = random.randint(0, len(cards)-1)
        cards[i], cards[n] = cards[n], cards[i]

def fisher_yates_shuffle(cards):
    """Fisher-Yates Shuffle: 從最後一張卡往前，確保僅在尚未洗過的區間取亂數索引並交換。"""
    for i in range(len(cards)-1, 0, -1):
        n = random.randint(0, i)
        cards[i], cards[n] = cards[n], cards[i]

def simulate(shuffle_fn, trials=1000000):
    """重複執行洗牌並統計結果。"""
    results = defaultdict(int)
    for _ in range(trials):
        cards = [1, 2, 3, 4]
        shuffle_fn(cards)
        results[tuple(cards)] += 1
    return results

def print_all_permutations(title, results):
    """強制生成所有24種排列，確保未出現的排列顯示次數為 0。"""
    print(title)
    all_perms = permutations([1, 2, 3, 4])  # 生成所有可能排列
    for perm in sorted(all_perms):
        count = results.get(perm, 0)         # 若不存在則返回 0
        print(f"{list(perm)}: {count}")

if __name__ == "__main__":
    # 模擬兩種洗牌方法
    naive_counts = simulate(naive_shuffle, 1000000)
    fy_counts = simulate(fisher_yates_shuffle, 1000000)

    # 輸出完整排列結果
    print_all_permutations("Naive Shuffle Results:", naive_counts)
    print("\n" + "="*50 + "\n")
    print_all_permutations("Fisher-Yates Shuffle Results:", fy_counts)
    print("\n" + "="*50 + "\n")
```





輸出結果如上，左圖為Naïve Shuffle Algorithm，右圖為Fisher-Yates Shuffle Algorithm，可看出相較之下Naïve Shuffle Algorithm的偏誤較大，而Fisher-Yates Shuffle Algorithm輸出結果較為均勻。

2.2

2. **Algorithm Comparison:**

   - Based on your analysis, which one is better, and why?

Ans: 筆者認為**Fisher–Yates**的演算法較佳，原因如下:

因為以古典機率而言在**1,000,000**次模擬中，所有24種排列的出現次數的理論值應為41,666次。而對照模擬結果可發現，Fisher–Yates洗牌演算法顯示出較為均勻的隨機分布，且與理論值較接近。反觀Naïve洗牌法則因為每一步均從全牌組中選取index，導致其分布存在統計偏差，因而某些排列會明顯偏多、某些排列偏少。

至於為何Fisher–Yates洗牌法可以展現較均勻的隨機分布，則是因為其在交換過程能**確保每個元素僅在未處理區間中交換**，使得Fisher–Yates洗牌具備嚴謹且均勻的隨機性，故在隨機性要求較高或安全性要求嚴格的場景中必然優於Naïve洗牌法。相較之下Naïve洗牌演算法雖易於理解，但其隨機分布不均之缺陷，使其在有安全需求的應用中極為不適合。

2.3

3. **Drawback Analysis:**

- What are the drawbacks of the other algorithm?

Naïve洗牌演算法的主要缺點在於它未能對尚未處理的數列進行限定，每次交換時從會從全部的範圍進行選取，這會導致已交換過的元素可能會被重複選取，進而產生非均勻的排列分布，因而出現某些排列的出現機率明顯高於理論的均勻分布，或是明顯低於理論值(1/24)的情況。而這種分布偏差與偏誤會在大量模擬中呈現較為明顯的差異，從而降低隨機性。

在金鑰生成或隨機數產生中，這種統計偏差很容易被攻擊者利用，進一步導致攻擊成功的風險提高。此外，Naïve方法缺乏嚴謹的數學證明，其隨機性並不可靠，因此在需保證完全均勻分布的前提下，此演算法是無法滿足安全要求。

2.4

4. **RC4 Improvement:**

- After learning these two shuffling algorithms, please propose improvement methods for "RC4."
- (Note: This section does not cover the Naïve or Fisher-Yates (FY) algorithms mentioned above. Instead, it focuses on discussing "RC4," specifically the "KSA" and "PRGA".)

Ans: RC4的缺點在於其**Key Scheduling Algorithm (KSA)**階段存在混亂不足、初始化狀態偏差以及隨後PRGA輸出前幾位存在統計偏差。以下為改進策略：

1. 針對RC4的KSA初始化過程進行改善：

- **實現多輪混合**：即不僅進行單輪256次交換，而是增加2到3輪混合操作，以使S陣列的初始狀態更加隨機、均勻。
- **採用Fisher–Yates風格的交換方式**：即僅在尚未處理的區間內選取交換對象，這樣可避免重複交換已固定的位置，從而進一步降低初期偏差。

2. **丟棄初始輸出（RC4-drop）**： 在使用RC4生成密鑰流時，先丟棄前768或1024 Bytes的輸出(統計偏誤較高的輸出)，待後續輸出**達到足夠隨機性後**再進行應用。

3. **加入額外熵（Entropy Injection）**：加入動態的初始化向量、nonce或salt，使得每次KSA初始化有額外不可預測的變數，從而有效防止重複使用相同密鑰與初始化向量組合而引起的偏差。

3.1

1. **Miller-Rabin on RSA Moduli:**
   - What happens when we apply the Miller-Rabin test to numbers in the format pq, where p and q are large prime numbers?

Ans:

對於RSA模數n = p×q而言，n會由兩個大質數(p、q)相乘而成，因此必然為合數。故在應用Miller-Rabin測試來檢查此類 n的質性時，該測試透過隨機選取基數a並計算a^dmod n及其相關冪次運算來評估 n是否具有質數的屬性。理論上，對於任意合數，至少有75%的隨機選取基數將顯示出與質數應有性質不符的行為，從而使測試能夠迅速判定n為合數。儘管卡邁克爾數（Carmichael numbers）在某些情況下可能使部分基數無法揭示其合數性，但由於RSA模數中所選取的p與q為隨機生成的大質數，其乘積極少滿足卡邁克爾數所需的特殊條件，此外，在RSA金鑰生成過程中通常會對p與q進行多次Miller-Rabin測試以確保其確實為質數，因此能夠可靠地識別n為合數。需要注意的是，Miller-Rabin測試僅作為一種機率性質數檢定工具，旨在確定一個數是否為合數，並不提供由此分解 n得出其質因數p與q的方法。

3.2

2. **RSA Security Analysis:**
   - Can we break RSA with it? Please provide a reasonable explanation.

Ans:

RSA的安全性將依賴於把大數n分解為兩個質數p與q的困難度，而 Miller-Rabin測試僅能協助我們判定某個數是否可能是質數，卻無法分解該合數。此外，Miller-Rabin在偵測到n是合數之後，並不提供任何後續步驟來獲取其質因數，因此即使知道n不為質數，也無法藉此破解RSA。換言之，RSA面臨的真正挑戰在於如何有效率地將n因式分解，Miller-Rabin對此無能為力；要想攻破RSA，仍需能在合理時間內取得p與q，而非僅僅證明n是合數，因此使用Miller-Rabin進行合數判定並不足以破壞RSA的安全性。