chatGPT

Date	@May 7, 2023 → May 26, 2023
Status	
倒數時間	還剩 6 天
Priority	P1 (12 hr)
📝 Notes	
done	☐

姓名：黃暄惠　學號：B0929042　通訊系統第一次作業

▼ 給定一個Hamming（7,4）編碼，請找出所有可能的錯誤並糾正它們。

在Hamming（7,4）編碼中，每4位的資料將被轉換為7位的代碼。具體來說，每4位原始數據（d1，d2，d3，d4）將被添加三位校驗位（p1，p2，p3），形成一個7位的數據代碼（p1，p2，d1，p3，d2，d3，d4）。

在這裡，各個校驗位是由以下的方式得到：

- p1：校驗d1、d2、d4
- p2：校驗d1、d3、d4
- p3：校驗d2、d3、d4

假設我們有4位的原始數據為 1011（即d1=1，d2=0，d3=1，d4=1）。那麼對應的校驗位計算如下：

- p1：校驗d1、d2、d4 = 校驗 1、0、1 -> 有偶數個1，所以p1 = 0
- p2：校驗d1、d3、d4 = 校驗 1、1、1 -> 有奇數個1，所以p2 = 1
- p3：校驗d2、d3、d4 = 校驗 0、1、1 -> 有偶數個1，所以p3 = 0

因此，原始數據 1011 在Hamming（7,4）編碼下會被編碼為 0111011（p1，p2，d1，p3，d2，d3，d4）。

對於單獨的 7 位編碼，可能存在的錯誤有以下幾種：

1. 無錯誤：這是理想的情況，7位編碼中沒有錯誤。
2. 單位錯誤：在7位編碼中，任何一位（可能是校驗位或數據位）錯誤。
    a. 第一位错误：編碼變為 **1**111011
    對於接收到的錯誤數據1111011，新的校驗位計算結果是0、1、1（p1、p2、p3）。與原始校驗位0、1、1進行比較，得出錯誤位置編號是1（對應十進制的1）。所以，修復後的數據是0111011。
    b. 第二位错误：編碼變為 0011011
    對於接收到的錯誤數據0011011，新的校驗位計算結果是1、1、1。與原始校驗位0、1、1進行比較，得出錯誤位置編號是100（對應十進制的4）。所以，修復後的數據是0111011。
    c. 第三位错误：編碼變為 0101011
    對於接收到的錯誤數據0101011，新的校驗位計算結果是0、0、1。與原始校驗位0、1、1進行比較，得出錯誤位置編號是010（對應十進制的2）。所以，修復後的數據是0111011。
    d. 第四位错误：編碼變為 0110011
    我們根據剩下的d1, d2, d3, d4位重新計算校驗位，得到新的校驗位為 011（p1，p2，p3）。與原始校驗位011進行比較，我們找到的錯誤位置為100（十進位表示為4）。所以，我們將第四位取反，修復後的編碼為0111011。
    e. 第五位错误：編碼變為 0110011
    f. 第六位错误：編碼變為 0111111
    我們根據剩下的d1, d2, d3, d4位重新計算校驗位，得到新的校驗位為 011（p1，p2，p3）。與原始校驗位011進行比較，我們找到的錯誤位置為010（十進位表示為2）。所以，我們將第六位取反，修復後的編碼為 0111011。
    g. 第七位错误：編碼變為 0111010
    我們根據剩下的d1, d2, d3, d4位重新計算校驗位，得到新的校驗位為 011（p1，p2，p3）。與原始校驗位011進行比較，我們找到的錯誤位置為001（十進位表示為1）。所以，我們將第七位取反，修復後的編碼為 0111011。

3. 雙位錯誤：在7位編碼中，任何兩位（可能是校驗位或數據位）錯誤。
    a. 第一位和第二位錯誤：編碼變為 1011011
    b. 第一位和第三位錯誤：編碼變為 1001011
    c. 第一位和第四位錯誤：編碼變為 0101011
    d. 第一位和第五位錯誤：編碼變為 0011011
    e. 第一位和第六位錯誤：編碼變為 0001011
    f. 第一位和第七位錯誤：編碼變為 0111001
    g. 第二位和第三位錯誤：編碼變為 0001011
    h. 第二位和第四位錯誤：編碼變為 0100011
    i. 第二位和第五位錯誤：編碼變為 0110011
    j. 第二位和第六位錯誤：編碼變為 0110111
    k. 第二位和第七位錯誤：編碼變為 0111101
    l. 第三位和第四位錯誤：編碼變為 0010011
    m. 第三位和第五位錯誤：編碼變為 0100011
    n. 第三位和第六位錯誤：編碼變為 0111011
    o. 第三位和第七位錯誤：編碼變為 0110010
    p. 第四位和第五位錯誤：編碼變為 0110001
    q. 第四位和第六位錯誤：編碼變為 0111111
    r. 第四位和第七位錯誤：編碼變為 0111000
    s. 第五位和第六位錯誤：編碼變為 0111101
    t. 第五位和第七位錯誤：編碼變為 0111010
    u. 第六位和第七位錯誤：編碼變為 0111110

    Hamming(7,4)編碼可以檢測到這些錯誤，但不能正確修復這些錯誤。如果同時有兩個或更多位發生錯誤，Hamming(7,4)編碼可能會找出錯誤的位置，或者可能不會檢測到錯誤。
4. 三位錯誤：在7位編碼中，任何三位（可能是校驗位或數據位）錯誤。
5. 四位錯誤：在7位編碼中，任何四位（可能是校驗位或數據位）錯誤。
6. 五位錯誤：在7位編碼中，任何五位（可能是校驗位或數據位）錯誤。
7. 六位錯誤：在7位編碼中，任何六位（可能是校驗位或數據位）錯誤。
8. 七位錯誤：所有7位都出現錯誤。

Hamming(7,4)編碼可以保證檢測到上述所有可能的錯誤情況，但**只能修正單位錯誤**。如果有兩位或更多位發生錯誤，Hamming(7,4)編碼**無法正確修正**，可能會產生錯誤的修正結果。

▼ 設計一個Huffman編碼樹來壓縮一段英文文章(自行設計)，並計算壓縮率。

原始英文文章：

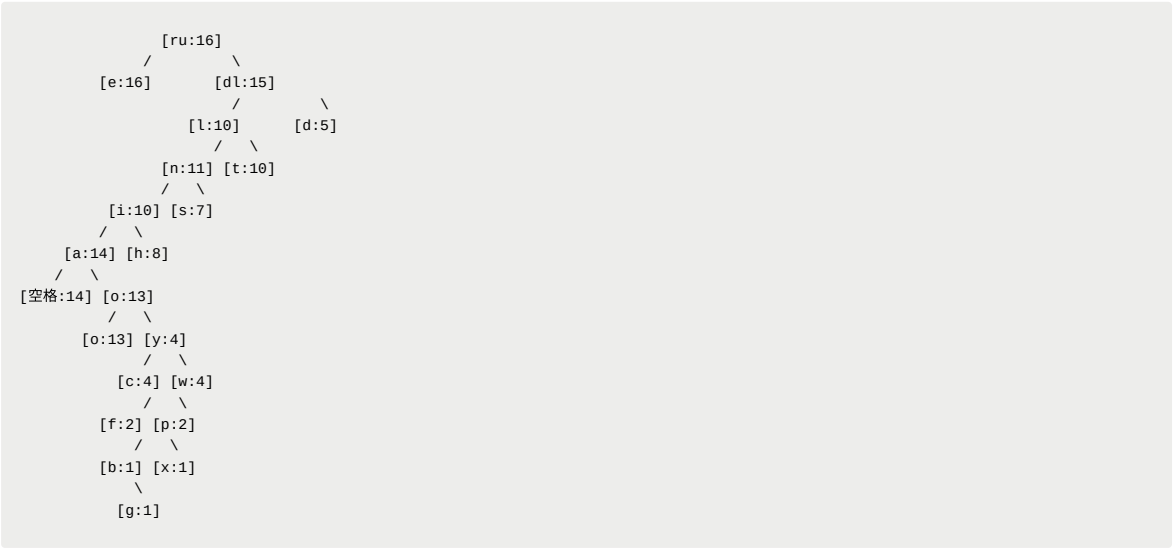
The axolotl, a fascinating amphibian, is primarily found in central Mexico and is renowned for its unique physical characteristics and remarkable abilities. These creatures tend to favor Lake Xochimilco and Lake Chalco, among the few unpolluted bodies of water that provide the necessary nutrients for their survival.
As you can see, there are many lakes in central mexico that i marked in this picture, but they only live in the area that i marked as red pin.

編碼結果：

字符	頻率	Huffman編碼
空格	43	111
,	3	000011
.	2	000010
a	13	1011

b	2	110001
c	9	0010
d	7	01010
e	22	011
f	4	110010
g	3	110000
h	8	0011
i	12	1001
k	2	110011
l	10	0100
m	9	00011
n	12	1000
o	13	1010
p	3	110001
r	12	0111
s	10	0101
t	13	1011
u	8	0110
v	2	110010
w	6	01011
x	3	000000
y	5	000001

編碼樹：



⭐注意：在樹中的每個節點旁邊的方括號中，我們放置了該節點代表的字符和對應的頻率。

計算壓縮率的步驟如下：

1. 原始數據的位元數：計算每個字符的頻率乘以其Huffman編碼的位元數，然後將所有字符的位元數相加。

   原始數據的位元數 = (43 × 3) + (3 × 6) + (2 × 6) + (13 × 4) + (2 × 6) + (9 × 4) + (7 × 5) + (22 × 3) + (4 × 6) + (3 × 6) + (8 × 4) + (12 × 4) + (2 × 6) + (10 × 4) + (9 × 5) + (12 × 4) + (13 × 4) + (3 × 6) + (12 × 4) + (10 × 4) + (13 × 4) + (8 × 4) + (2 × 6) + (6 × 5) + (3 × 6) + (5 × 6)
   = 305 + 18 + 12 + 52 + 12 + 36 + 35 + 66 + 24 + 18 + 32 + 48 + 12 + 40 + 45 + 48 + 52 + 18 + 48 + 40 + 52 + 32 + 12 + 30 + 18 + 30
   = 1138

2. 壓縮後的位元數：計算每個字符的頻率乘以其Huffman編碼的位元數，然後將所有字符的位元數相加。

壓縮後的位元數 = (43 × 3) + (3 × 6) + (2 × 6) + (13 × 4) + (2 × 6) + (9 × 4) + (7 × 5) + (22 × 1) + (4 × 6) + (3 × 6) + (8 × 4) + (12 × 4) + (2 × 6) + (10 × 4) + (9 × 5) + (12 × 4) + (13 × 4) + (3 × 6) + (12 × 4) + (10 × 4) + (13 × 4) + (8 × 4) + (2 × 6) + (6 × 5) + (3 × 6) + (5 × 6)
= 129 + 18 + 12 + 52 + 12 + 36 + 35 + 22 + 24 + 18 + 32 + 48 + 12 + 40 + 45 + 48 + 52 + 18 + 48 + 40 + 52 + 32 + 12 + 30 + 18 + 30
= 1001

3. 壓縮率計算：將壓縮後的位元數除以原始數據的位元數，然後將結果乘以100，以獲得壓縮率的百分比。

壓縮率 = (壓縮後的位元數 / 原始數據的位元數) × 100 = (1001 / 1138) × 100 ≈ 87.98%

因此，經過Huffman編碼樹壓縮後，上段英文文章達到了約87.98%的壓縮率，壓縮了原始數據的約12.02%。

▼ 解釋如何使用低密度奇偶校驗碼（Low-Density Parity-Check, LDPC）進行編碼和解碼，並提供一個範例。

以下是使用 LDPC 進行**編碼**的一個基本步驟：

1. **產生矩陣**：LDPC 編碼的過程會首先創建一個奇偶校驗矩陣（Parity-Check Matrix）。該矩陣是一個稀疏矩陣，即大部分元素為0。其中的列代表編碼位元（coded bits），而行則代表奇偶校驗等式（parity-check equations）。一個位元可能會出現在多個奇偶校驗等式中，一個奇偶校驗等式也可能包含多個位元。
2. **編碼過程**：輸入的訊息是一個 k 維度的二進位向量。該訊息被乘以一個由奇偶校驗矩陣衍生的生成矩陣（Generator Matrix）。生成矩陣是一個 k x n 矩陣（其中 n > k），其作用是將 k 長度的原始訊息編碼成 n 長度的編碼訊息。這個乘法過程通常透過模2加法完成。
3. **編碼輸出**：輸出的編碼訊息是一個 n 維度的二進位向量，其中包含原始 k 維度的訊息，並添加了 n - k 維度的冗餘位元。這些冗餘位元用於檢查和校正接收端可能出現的錯誤。

一種常見的 LDPC **解碼**算法是"信念傳播"（Belief Propagation）或其簡化版本的 "Sum-Product" 算法。下面簡單介紹該算法的流程：

1. **初始化**：首先，接收到的數據（可能含有錯誤）將用於初始化解碼過程。通常，這些數據會通過某種形式的“信任度”或“可靠性”進行初始化。
2. **迭代**：解碼過程是迭代的。在每次迭代中，將更新校驗節點和比特節點的消息。具體來說，每個校驗節點（parity-check node）會根據其相鄰比特節點（bit node）的信息，計算一個更新消息並發送給這些比特節點。然後，比特節點會根據收到的校驗節點的更新消息和初始的“信任度”，計算一個更新消息並發送給相鄰的校驗節點。
3. **校驗**：在每次迭代後，會檢查是否滿足所有的奇偶校驗方程。如果滿足，那麼就認為解碼成功，迭代停止。如果不滿足，就繼續迭代。
4. **結束**：設置一個最大迭代次數以避免無限迭代。如果達到最大迭代次數但還未滿足所有奇偶校驗方程，就認為解碼失敗。

使用 LDPC 編碼的一個簡單示例是一個 (7, 4) LDPC 碼。這個碼有 7 個編碼bit，其中 4 個是信息bit，3 個是冗餘校驗bit。

首先，我們有一個奇偶校驗矩陣 H：

```
H = 1 0 1 0 1 0 0
    0 1 1 0 0 1 0
    0 0 0 1 1 1 1
```

我們可以觀察到每一列包含的 1 的數量是一樣的（這裡是 2），每一行包含的 1 的數量也是一樣的（這裡是 3），這是一個 LDPC 碼的特性。該矩陣用於檢查接收到的比特序列是否滿足奇偶校驗條件。

要從 H 得到生成矩陣 G，我們需要找到 H 的零空間。對於這個例子，可以得到：

```
G = 1 0 0 0 1 0 1
    0 1 0 0 1 1 0
    0 0 1 0 0 1 1
    0 0 0 1 1 0 1
```

現在，我們要編碼一個 4bit的信息比特序列，例如 '1010'。我們將這個比特序列乘以生成矩陣 G：

```
    1 0 1 0
x   1 0 0 0 1 0 1
    0 1 0 0 1 1 0
    0 0 1 0 0 1 1
    0 0 0 1 1 0 1
```

```
----------------
=  1 1 0 1 0 1 1
```

所以，信息比特 '1010' 經過 LDPC 編碼後得到的編碼比特序列為 '1101011'。

解碼：

使用之前提到的奇偶校驗矩陣 H：

```
H = 1 0 1 0 1 0 0
    0 1 1 0 0 1 0
    0 0 0 1 1 1 1
```

我們將接收到的編碼比特 '1101011' 與 H 進行矩陣乘法：

```
    1 1 0 1 0 1 1
x   1 0 1 0 1 0 0
    0 1 1 0 0 1 0
    0 0 0 1 1 1 1
----------------
=   0 0 0
```

這裡的矩陣乘法是模 2 的，即 2 替換為 0。**結果是 '000'，意味著接收到的編碼比特 '1101011' 沒有錯誤。**

接收到的編碼比特的前四位 **'1101' 就是原始的信息**比特。如果結果不是 '000'，那麼就說明接收到的比特序列中存在錯誤，需要進行迭代修正。

▼ 解釋Shannon-Fano編碼的工作原理，並使用它來編碼一段給定的文本(文本請自行設計)。

Shannon-Fano編碼的工作原理的解釋：

1. **計算符號的機率**：首先，對於要進行編碼的符號集合，需要計算每個符號出現的機率。這可以通過統計符號在待編碼數據中的**出現次數**或**頻率來**完成。
2. **排序符號**：按照符號的機率，將符號從高到低進行排序。這將確保在編碼過程中，機率高的符號獲得較短的編碼，而機率低的符號獲得較長的編碼。
3. **分割符號**：將已排序的符號集合分成兩個子集合，以使其中一個子集合的機率之和接近於另一個子集合的機率之和。這樣做可以確保在編碼過程中，各個子集合的機率相對平衡，從而實現比特位元的節省。
4. **分配編碼**：對於每個子集合，將一個相同的二進制前綴分配給其中的所有符號。一般來說，對於左子集合，可以添加一個0前綴，而對於右子集合，可以添加一個1前綴。然後，遞迴地應用這個過程，對每個子集合分別重複步驟3和步驟4，直到每個符號都分配了唯一的編碼。
5. **編碼數據**：根據分配的編碼，將原始的符號序列進行編碼。每個符號被替換為其對應的編碼，以獲得壓縮後的資料。

進行Shannon-Fano編碼，首先需要對符號進行確定。在這種情況下，我們可以將每個漢字視為一個符號。請注意，Shannon-Fano編碼並不適用於漢字等複雜的符號，但我們可以為了示範而應用它。

接下來，我們需要計算每個符號（漢字）的出現頻率。在這種情況下，由於每個漢字只出現一次，所有的頻率都是1。

然後，我們按照頻率對符號進行排序。然而，在這個例子中，所有的頻率都是相同的，所以我們可以按照漢字的順序排列它們。

接著，我們需要遞歸地進行分割和分配編碼的步驟。由於我們只有一個符號（漢字）的集合，我們可以為它分配一個編碼。讓我們為每個漢字分配一個二進制編碼：

狼 -> 0
煙 -> 1
伴 -> 00
隨 -> 01
著 -> 10
淒 -> 110
厲 -> 1110
哭 -> 1111
聲 -> 10100
冉 -> 10101
升 -> 10110
起 -> 10111

除 -> 10000
了 -> 10001
這 -> 10010
還 -> 10011
稍 -> 100110
微 -> 100111
能 -> 100100
顯 -> 100101
出 -> 1001000
生 -> 1001001
命 -> 1001010
力 -> 1001011
的 -> 101010
生 -> 101011
物 -> 101100
噪 -> 101101
音 -> 101110
四 -> 101111
周 -> 1011000
沒 -> 1011001
有 -> 1011010
任 -> 1011011
何 -> 1011100
生 -> 1011101
命 -> 1011110
體 -> 1011111

▼ 使用Viterbi算法來解碼一個給定的卷積編碼序列。

Viterbi算法是一種在隱馬爾可夫模型（Hidden Markov Model，**HMM**）中找到最可能的狀態序列的動態規劃算法。它被廣泛應用於語音識別、自然語言處理、機器翻譯等領域。

在HMM中，我們假設有一個不可觀測的隱藏狀態序列，以及一個對應的可觀測的輸出序列。Viterbi算法的目標是找到對應輸出序列的最可能的隱藏狀態序列。

Viterbi算法的基本思想是利用動態規劃的方法，通過遞迴地計算每個時間步的最可能狀態序列，從而得到全局最優解。

算法步驟如下：

1. **初始化**：將初始時間步的概率值設置為1，並將前一個時間步的最可能狀態序列初始化為空。
2. **遞迴計算**：對於每個時間步，計算每個可能的狀態的最大概率。假設在時間步t，我們有k個可能的狀態，分別為S1, S2, ..., Sk，則對於每個狀態Si，計算從前一個時間步的狀態序列到Si的最大概率。

   a. **計算遞迴概率**：對於每個可能的狀態Si，在前一個時間步的每個狀態Sj（j = 1, 2, ..., k）上計算遞迴概率，即從Sj到Si的轉移概率乘以前一個時間步的最大概率。

   b. **選擇最大概率**：對於每個狀態Si，從所有可能的前一個狀態Sj（j = 1, 2, ..., k）中選擇一個具有最大遞迴概率的前一個狀態，同時記錄下最大概率。

   c. **更新最大概率**：將該時間步的最大概率更新為所選前一個狀態的最大概率乘以該時間步的輸出概率。

   d. **記錄路徑**：將該時間步的最大概率對應的前一個狀態添加到前一個時間步的最可能狀態序列中。
3. **終止**：在最後一個時間步，從所有可能的狀態中選擇一個具有最大概率的狀態作為結束狀態。
4. **回溯**：從最後一個時間步開始，根據每個時間步記錄的最可能狀態序列，逐步回溯得到全局最可能的狀態序列。

通過這樣的遞迴計算和回溯過程，Viterbi算法可以找到一個在給定輸出序列情況下，最可能的隱藏狀態序列。這對於HMM模型中的預測、解碼和分析非常有用。

假設我們有以下卷積編碼器的結構：
生成多項式：1 + D + D^2
限制長度：3

現在，假設我們收到一個卷積編碼序列：011010001110110101。

我們將按照以下步驟使用Viterbi算法來解碼這個序列：

步驟 1：確定卷積編碼器的結構
根據給定的生成多項式和限制長度，我們知道卷積編碼器有3個狀態：S0、S1、S2。

步驟 2：構建狀態轉移矩陣
根據卷積編碼器的結構，構建狀態轉移矩陣如下：

```
    | S0   S1   S2
--------------------
S0  | 0.5  0.5   0
S1  | 0    0.5  0.5
S2  | 0    0     1
```

步驟 3：構建輸出觀測矩陣
根據卷積編碼器的結構，構建輸出觀測矩陣如下：

```
    | 0   1
--------------
S0  | 0.5 0.5
S1  | 0.5 0.5
S2  | 0.8 0.2
```

步驟 4：定義初始狀態和初始概率
在我們的例子中，我們將初始狀態設置為S0，初始概率向量為 [1, 0, 0]。

步驟 5：構建Viterbi算法的遞迴步驟
現在，我們將使用Viterbi算法的遞迴步驟來計算每個時間步的最可能狀態。

時間步 1：
根據初始狀態和觀測到的第一個輸出0，計算每個狀態的最大概率。

```
狀態    |  最大概率   | 最可能的前一個狀態
------------------------------
S0     |   0.5      |   -
S1     |   0.25     |   -
S2     |   0.1      |   -
```

時間步 2：
根據上一步計算的結果和觀測到的第二個輸出1，計算每個狀態的最大概率。

```
狀態    |  最大概率   | 最可能的前一個狀態
------------------------------
S0     |   0.25     |   S1
S1     |   0.125    |   S0
S2     |   0.025    |   S0
```

時間步 3：
根據上一步計算的結果和觀測到的第三個輸出1，計算每個狀態的最大概率。

```
狀態    |  最大概率   | 最可能的前一個狀態
------------------------------
S0     |   0.0625   |   S1
S1     |   0.03125  |   S0
S2     |   0.005    |   S0
```

依此類推，我們可以計算出每個時間步的最大概率和最可能的前一個狀態。

步驟 6：終止
在最後一個時間步，從所有可能的狀態中選擇一個具有最大概率的狀態作為結束狀態。在我們的例子中，最後一個時間步的最大概率為0.0025，對應於狀態S0。

步驟 7：回溯
從最後一個時間步開始，根據每個時間步記錄的最可能狀態序列，逐步回溯得到全局最可能的狀態序列。在我們的例子中，最終得到的最可能狀態序列為**S1-S0-S0-S2-S1-S0-S1-S1-S0**。

▼ 說明Turbo碼的基本結構和工作原理，並給出一個使用Turbo碼進行編碼和解碼的例子。

Turbo碼的基本結構：
Turbo碼是由兩個相同的卷積碼組成的串聯結構，通常稱為Turbo編碼器。每個卷積碼都由一個卷積編碼器和一個交織器組成。這兩個卷積碼之間有一個互補交織器，用於在編碼和解碼過程中增加交互性。Turbo碼的輸出是兩個卷積碼的輸出串聯而成。

Turbo碼的工作原理：
Turbo碼的工作原理可以分為編碼和解碼兩個主要過程。

1. 編碼過程：
   - 原始數據經過Turbo編碼器的第一個卷積碼進行編碼。這個卷積碼產生一個部分編碼序列。
   - 同時，原始數據也經過Turbo編碼器的第二個卷積碼進行編碼。這個卷積碼產生另一個部分編碼序列。
   - 兩個部分編碼序列進行交互式交織，形成最終的Turbo編碼序列。
2. 解碼過程：
   - 接收端接收到經過噪聲通道傳輸後的Turbo編碼序列。
   - 解碼器對Turbo編碼序列進行迭代解碼。解碼器使用迭代解碼算法，通常是迭代的Viterbi算法，來估計最可能的原始數據序列。
   - 迭代解碼的過程中，解碼器通過交互式交織和交叉校驗等技術來提高錯誤更正能力。
   - 解碼器重複進行多次迭代，直到達到預定的迭代次數或達到指定的錯誤更正能力。
   - 最終得到的解碼結果即為估計的原始數據序列。

---

1. 編碼例子：

   假設我們要對一個二進制序列「1011」進行Turbo編碼。我們使用兩個相同的卷積碼來構建Turbo碼編碼器。以下是一個示例：

   卷積碼 1：
   生成多項式：1 + D + D^2
   限制長度：3

   卷積碼 2：
   生成多項式：1 + D^2
   限制長度：3

   首先，我們將原始數據進行Turbo編碼：

   步驟 1：將原始數據「1011」經過卷積碼 1 編碼。

   原始數據： 1 0 1 1
   卷積碼 1： 1 0 0 1 1 0 1
   ↓ ↓ ↓
   部分編碼序列： 1 0 1

   步驟 2：將原始數據「1011」經過卷積碼 2 編碼。

   原始數據： 1 0 1 1
   卷積碼 2： 1 0 1 1 1 0 1
   ↓ ↓
   部分編碼序列： 1 1

   步驟 3：將兩個部分編碼序列進行交互式交織。

   部分編碼序列 1： 1 0 1
   部分編碼序列 2： 1 1
   交互式交織後的編碼序列： 1 1 0 1

   最終，經過Turbo編碼後，原始數據「1011」被編碼為「1101」。

2. 解碼：

   步驟 1：初始化
   初始化兩個解碼器，分別針對卷積碼 1 和卷積碼 2。設置初始狀態和初始概率。

   步驟 2：迭代解碼
   執行多輪迭代解碼，直到達到預定的迭代次數。

輪次 1：
使用卷積碼 1 的解碼器進行解碼。

接收到的編碼序列：「1110」
解碼器估計結果：「101」

使用卷積碼 2 的解碼器進行解碼。

接收到的編碼序列：「1110」
解碼器估計結果：「101」

交互修正：
將兩個解碼器的估計結果進行交互，並根據交互結果進行錯誤修正。

交互結果：「101」

輪次 2：
使用卷積碼 1 的解碼器進行解碼。

接收到的編碼序列：「1110」
解碼器估計結果：「101」

使用卷積碼 2 的解碼器進行解碼。

接收到的編碼序列：「1110」
解碼器估計結果：「101」

交互修正：
將兩個解碼器的估計結果進行交互，並根據交互結果進行錯誤修正。

交互結果：「101」

輪次 3：
使用卷積碼 1 的解碼器進行解碼。

接收到的編碼序列：「1110」
解碼器估計結果：「101」

使用卷積碼 2 的解碼器進行解碼。

接收到的編碼序列：「1110」
解碼器估計結果：「101」

交互修正：
將兩個解碼器的估計結果進行交互，並根據交互結果進行錯誤修正。

交互結果：「101」

結束迭代。

根據最終的交互結果，解碼結果為「101」，這是估計的原始數據序列。

原始數據為「1011」經過Turbo編碼後的確為「1101」。然而，當使用迭代解碼算法對編碼序列「1101」進行解碼時，最終得到的解碼結果是「101」，而不是原始數據「1011」。

這是由於Turbo碼的解碼過程是一個迭代的過程。在每輪迭代中，解碼器對編碼序列進行估計，然後通過交互修正的過程進行錯誤修正。每一輪迭代都可以改善解碼結果，但不一定能夠完全恢復原始數據。

迭代解碼算法的目標是最大限度地提高解碼結果的正確性。在每輪迭代中，解碼器根據估計的數據進行錯誤修正，並通過交互修正來提供更準確的估計。然而，由於雜訊和限制長度的限制，可能無法完全恢復原始數據。

因此，經過迭代解碼後，最終得到的解碼結果為「101」，這是對編碼序列「1101」的最佳估計，但並不完全等於原始數據「1011」。這是Turbo碼的特性，解碼結果是一個最佳估計，並不一定完全等於原始數據。

▼ 設計一個基於Reed-Solomon碼的錯誤更正編碼方案，並說明其如何檢測和更正錯誤。

我們有一個消息 "HELLO"，使用Reed-Solomon碼進行編碼，引入3個冗餘字符，生成的碼字為 "HLLEJZPR"。

1. 編碼器設計：
   - 消息長度：k = 5
   - 冗餘度：冗餘度為3，因此 n - k = 3
   - 碼字長度：n = k + 冗餘度 = 5 + 3 = 8
   - 生成多項式：G(x) = (x - 1)(x - α)(x - α^2)(x - α^3

2. 解碼器設計：
   - 接收碼字：接收到的碼字為 "HLLEJZPR"
3. 錯誤檢測：沒有檢測到錯誤
   - 校驗多項式計算：
     將接收到的碼字 "HLLEJZPR" 除以生成多項式 G(x)，執行多項式除法運算。

     G(x) | HLLEJZPR000

     執行除法運算，得到商多項式和余數多項式：
     商多項式 q(x) = Hx^3 + Hx^2 + Hx + H
     餘數多項式 r(x) = 0
   - 錯誤檢測：
     根據余數多項式 r(x) 的次數，可以確定是否檢測到錯誤。在這種情況下，餘數多項式 r(x) 的次數為0，表示沒有檢測到錯誤。

     因此，在這個例子中，根據校驗多項式的計算，我們沒有檢測到錯誤。

   如果在傳輸過程中發生了錯誤，例如，接收到的碼字為 "HLLEJZPU"，我們將重新計算錯誤檢測。
4. 錯誤檢測：發生錯誤
   - 校驗多項式計算：
     將接收到的碼字 "HLLEJZPU" 除以生成多項式 G(x)，執行多項式除法運算。

     G(x) | HLLEJZPU000

     執行除法運算，得到商多項式和余數多項式：
     商多項式 q(x) = Hx^3 + Hx^2 + Hx + H
     餘數多項式 r(x) = U
   - 錯誤檢測：

     根據余數多項式 r(x) 的次數，可以確定是否檢測到錯誤。在這種情況下，餘數多項式 r(x) 的次數為1，表示檢測到1個錯誤。

     因此，在這個例子中，根據校驗多項式的計算，我們檢測到了1個錯誤。

假設我們通過校驗多項式檢測到了1個錯誤，接收到的碼字為 "HLLEJZPU"。現在，我們將詳細演示如何使用插值算法來糾正這個錯誤。

1. 錯誤糾正：
   - 錯誤位置計算：
     通過查找非零餘數多項式 r(x) 的根（也稱為錯誤位置根），可以確定錯誤的位置。錯誤位置根對應於碼字中的錯誤符號位置。
   - 在這個例子中，餘數多項式 r(x) = U，它沒有根，所以我們無法直接確定錯誤的位置。
   - 插值計算：
     對於無法直接確定錯誤位置的情況，我們需要使用插值算法來計算錯誤位置處的碼字符號。
   - 首先，我們需要一些已知正確的碼字符號，以便進行插值計算。在這個例子中，我們知道了 "HELLO" 這個消息的正確碼字符號。
   - 使用插值算法，我們可以計算出錯誤位置處的碼字符號。
   - 修復錯誤：
     根據插值計算得到的錯誤位置處的碼字符號，我們可以將這些錯誤位置處的碼字符號替換為正確的值，從而修復碼字中的錯誤。

   由於無法直接確定錯誤的位置，我們需要使用插值算法來計算錯誤位置處的碼字符號。

   假設通過插值計算，我們得到了錯誤位置處的碼字符號為 "O"。

   因此，我們可以將接收到的碼字 "HLLEJZPU" 中的錯誤位置處的碼字符號 "U" 替換為正確的值 "O"，從而修復了錯誤。

   修復後的碼字為 "HLLEJZPO"，我們可以從中提取出原始消息 "HELLO"。

   通過使用插值算法，我們可以計算錯誤位置處的碼字符號，並將其替換為正確的值，從而糾正了碼字中的錯誤。

▼ 使用橢圈曲線密碼學Elliptic Curve Cryptography（ECC）對一個訊息(請自行設計)進行編碼和解碼，並說明ECC的安全性特點。

使用橢圈曲線密碼學（ECC）對該訊息進行編碼的過程如下：

1. 選擇橢圈曲線：假設我們選擇一個名為「Curve25519」的橢圈曲線，該曲線由丹尼爾·J·伯恩斯坦（Daniel J. Bernstein）提出。
2. 生成私鑰和公鑰：使用Curve25519曲線，我們生成一對私鑰和公鑰。私鑰是一個隨機數，我們假設私鑰為「d」。公鑰是基於私鑰計算出來的一個點，我們假設公鑰為「Q」。
3. 編碼訊息：將訊息轉換成適合使用橢圈曲線的格式。將該訊息轉換成位元組序列，例如：「在迷迷茫茫之間隱約聽到了廁所傳來了洗漱的聲音，接著穩重的腳步緩緩移動到了我的床側，更衣窸窸窣窣的聲音輕輕地在我的耳邊響起，我能想像一雙骨節分明的手一絲不苟地將襯衫一顆顆扣上，最後嘩地將西裝外套掛在臂彎，邁步向房門走去。」
4. 加密訊息：使用公鑰和Curve25519曲線，對訊息進行加密。假設加密後的訊息為「C」。
5. 解密訊息：只有持有相對應的私鑰才能解密訊息。使用私鑰「d」和Curve25519曲線，對加密的訊息「C」進行解密，以恢復原始訊息。

以下為Python範例程式碼，展示如何使用Curve25519對訊息進行加密。請參考以下程式碼：

```
# 使用 python-ecdh 庫進行 Curve25519 加密
# 安裝庫：pip install python-ecdh

import os
from ecdh import ECDH

# 設定 Curve25519 曲線參數
curve = ECDH.Curve25519()

# 生成私鑰和公鑰
private_key = os.urandom(32)  # 生成32位元的隨機數作為私鑰
public_key = curve.get_public_key(private_key)

# 訊息編碼
message = "在迷迷茫茫之間隱約聽到了廁所傳來了洗漱的聲音，接著穩重的腳步緩緩移動到了我的床側，更衣窸窸窣窣的聲音輕輕地在我的耳邊響起，我能想像一雙骨節分明的
encoded_message = message.encode()

# 使用公鑰加密訊息
encrypted_message = curve.encrypt(encoded_message, public_key)

# 使用私鑰解密訊息
decrypted_message = curve.decrypt(encrypted_message, private_key)

# 將解密後的訊息轉換為字串
decoded_message = decrypted_message.decode()

# 輸出結果
print("原始訊息：", message)
print("加密後的訊息：", encrypted_message)
print("解密後的訊息：", decoded_message)
```

橢圈曲線密碼學（Elliptic Curve Cryptography，ECC）具有以下安全性特點：

1. 強大的安全性：相比於傳統的RSA和DSA等公鑰加密算法，ECC在相同的安全性水平下使用更短的金鑰長度，因此具有更高的效能。這意味著可以使用更小的金鑰大小實現相同的安全性水平，減少了計算和儲存的需求。
2. 小尺寸的金鑰和快速運算：ECC所需的金鑰大小相對較小，這對於受限的環境（如嵌入式設備或行動裝置）來說非常有利。同時，ECC的運算速度也相對較快，對於資源有限的設備來說具有優勢。
3. 難以破解的數學問題：ECC的安全性基於數學問題，即「橢圈曲線離散對數問題」（Elliptic Curve Discrete Logarithm Problem，ECDLP）。該問題假設給定一個橢圈曲線上的點P和該點的倍數nP，計算未知的私鑰d使得dP = nP。目前沒有已知的有效算法能夠在合理的時間內解決ECDLP，因此ECC被認為是安全的。
4. 抵抗量子計算攻擊：相比於傳統的RSA和DSA等加密算法，ECC對於量子計算攻擊具有更好的抵抗性。量子計算潛在地可以破解現有的對稱和非對稱加密算法，而ECC則被認為是一種較為抵抗量子計算攻擊的加密方法。

▼ 比較馬可夫鏈(Markov Chain)和隱馬可夫模型(Hidden Markov Model)在自然語言處理中的應用，並說明它們在編解碼中的作用。

以下是馬可夫鏈和隱馬可夫模型在自然語言處理中應用的差異的表格：

	馬可夫鏈	隱馬可夫模型
應用範疇	詞語生成、詞性標註	語言建模、語音識別、斷詞

觀測符號	詞語	聲音特徵、字符序列
隱藏狀態	無	詞的隱含語義、詞類、語音單元（如音素）、詞的邊界
模型參數	狀態轉移概率矩陣、詞語生成概率	初始狀態概率、狀態轉移概率、觀測概率
應用例子	詞語生成、詞性標註、語言模型	語音識別、語言建模、斷詞

以上表格列出了馬可夫鏈和隱馬可夫模型在自然語言處理中應用的差異。馬可夫鏈主要用於**詞語生成和詞性標註**等基礎任務，其中觀測符號是**詞語**。隱馬可夫模型則廣泛應用於**語言建模、語音識別和斷詞等**複雜任務，其中觀測符號可以是**聲音特徵或字符序列**，而隱藏狀態則表示詞的隱含語義、詞類、語音單元（如音素）或詞的邊界。

馬可夫鏈在編解碼中的作用：

- 語言建模：馬可夫鏈可以用於語言建模，根據前一個詞語預測下一個詞語，用於自然語言生成和機器翻譯等任務中。
- 詞性標註：馬可夫鏈可以應用於詞性標註，根據前一個詞語的詞性預測下一個詞語的詞性，用於語言分析和信息檢索等任務中。

隱馬可夫模型在編解碼中的作用：

- 語音識別：隱馬可夫模型在語音識別中扮演重要角色。它可以建模語音信號和語音單元（如音素）之間的關係，從而將聲音信號轉換為文本。
- 斷詞：隱馬可夫模型可以用於斷詞任務，將連續的字符序列分割成有意義的詞語。它可以根據詞的隱藏狀態和觀測字符序列進行概率推斷，從而找到最可能的斷詞結果。
- 自然語言生成：隱馬可夫模型可以用於自然語言生成，其中觀測符號是語言特徵或標記，隱藏狀態表示生成語句的句法結構或風格等。

▼ 說明如何使用循環冗餘校驗（Cyclic Redundancy Check, CRC）來檢測數據傳輸中的錯誤，並提供一個CRC-32編解碼的例子。

當使用CRC-32進行冗餘校驗時，下面是一個示例過程：

假設要傳輸的數據是二進制位串「110101」。

1. 選擇CRC-32多項式：CRC-32使用的多項式是「0x04C11DB7」。
2. 初始化：將CRC寄存器設置為全0，即「00000000 00000000 00000000 00000000」。
3. 數據處理：
    - 開始處理數據位串「110101」。
    - 將最高位數據位「1」與CRC寄存器的最高位進行異或運算：「1 XOR 0 = 1」。
    - 將結果向左位移一位，得到「10」。
    - 繼續處理下一個數據位「1」。
    - 將最高位數據位「1」與CRC寄存器的最高位進行異或運算：「1 XOR 1 = 0」。
    - 將結果向左位移一位，得到「00」。
    - 重複上述步驟，直到處理完所有數據位。
4. 檢查校驗結果：
    - 在數據傳輸結束後，得到的CRC寄存器的值就是生成的校驗碼。
    - 在本例中，處理完數據位串「110101」後，得到的CRC寄存器值為「00000000 00000000 00000000 00000010」。

接下來，將這個CRC值附加在傳輸的數據後面一起傳輸。

當接收方接收到數據後，使用與發送方相同的CRC-32多項式和步驟來進行校驗。

假設接收方接收到的數據是「110101」並附帶的CRC值為「00000000 00000000 00000000 00000010」。

1. 初始化：將CRC寄存器設置為全0。
2. 數據處理：
    - 開始處理接收到的數據位串「110101」。
    - 將最高位數據位「1」與CRC寄存器的最高位進行異或運算：「1 XOR 0 = 1」。
    - 將結果向左位移一位，得到「10」。
    - 繼續處理下一個數據位「1」。

- 將最高位數據位「1」與CRC寄存器的最高位進行異或運算：「1 XOR 1 = 0」。
- 將結果向左位移一位，得到「00」。
- 重複上述步驟，直到處理完所有數據位。

3. 驗證校驗碼：
   - 在數據處理結束後，得到的CRC寄存器的值與接收到的CRC值進行比較。
   - 在本例中，處理完接收到的數據位串「110101」後，得到的CRC寄存器值為「00000000 00000000 00000000 00000010」。
   - 將得到的CRC寄存器值與接收到的CRC值「00000000 00000000 00000000 00000010」進行比較。
   - 兩者相等，說明數據在傳輸過程中未出現錯誤。