
信息论与编码——信道编码实验指导

1 概要

本次实验的目的是让同学们理解并体会信道编码的作用以及信道编码的性能优化。我们集中在汉明码这种较为简单且有效的编码。

你可以使用你喜欢的语言编写程序，选择的语言不会影响你的分数。对核心功能以外的部分，你可以自由使用不同的库/包。

你可以参考不同来源的各种实现，但请务必确保代码由你自己编写。

我们会编译并运行你的代码，请确保你测试运行了你的代码。

2 计分

由于信道编码的数学性比较强，因此这次实验将更多地要求你理解并体会工作过程，而不是代码本身。

在理解了编码的工作过程后，用程序实现是非常简单的，因此，请仔细阅读指导书，并对有疑惑的部分积极提问。

指导书中所有 C 和 OP 开头的项目为你需要完成的目标。

其中，C 为必做(Critical)目标，你应该完成所有的必做目标。

OP 为选做(Optional)目标，你可以按你的喜好完成选做目标。

完成所有必做目标，你将获得 9~13 分的分数。（视完成的正确性）

每个选做目标都有他的分值，选做目标的分数将根据完成度、正确性给出。（即使没有完全完成也有机会拿到部分分数！请积极尝试感兴趣的选做目标。）

3 实验内容

3.1 汉明码——线性代数观点

在课上，你已经学习了如何使用矩阵运算实现[7, 4]汉明码，我们给出标准的[7, 4]汉明码的工作过程；请仔细阅读并理解这个过程。

以下的所有代数运算都在 \mathbb{F}_2 进行。

首先，对一个矩阵 H ，他的每一列从左到右分别对应二进制表示的 1, 2, 3, 4, 5, 6, 7。

$$H = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

我们约定矩阵上端代表最低有效位，这与课上所讲的不一样，这是为了在编码时可以直接按 $d_0d_1d_2d_3$ 的顺序构造欲编码的向量，且能与标准的汉明码——即在码字按 $(p_0, p_1, d_0, p_2, d_1, d_2, d_3)$ 排列数据校验位——保持一致。

如果使用矩阵下端代表最低有效位，也能正常工作，但需要重排编码后的码字以与标准汉明码保持一致。

为了方便测试你的程序，我们决定采用标准的汉明码。当然，即使以矩阵下端代表最低有效位，汉明码本身也能正常工作。

之后，我们需要寻找它的生成矩阵 G ，满足：

$$HG^T = \mathbf{0}$$

其中， G 是一个秩为 4 的矩阵。

如果你熟悉线性代数，你会发现我们实际在求 H 的零空间的一组基组成的矩阵，如果你不熟悉线性代数，也可以使用下面的方法：

我们交换 H 的列，将 H 变换到标准型 $H_s = (A \mid I_3)$ 。（为了与标准的汉明码保持一致， A 中每列按从小到大的顺序排列（矩阵上端为最低有效位））。

$$H_s = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

此时， H_s 的生成矩阵由 $G_s = (I_{7-3} \mid -A^T)$ 给出，同时注意到在 F_2 ， $-A = A$ ，我们给出标准形的生成矩阵：

$$G_s = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

之后，交换 H_s 的列到 H ，对 G_s 做完全同样的列交换，得到 G ：

$$G = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

欲对要发送的比特串 $d = (d_0, d_1, d_2, d_3)$ 编码，计算：

$$r = dG$$

欲对要接收的比特串 $r = (p_0, p_1, d_0, p_2, d_1, d_2, d_3)$ 译码，首先计算：

$$z = Hr^T$$

若结果为 0，则认为未出现错误，提取出 r 中的 (d_0, d_1, d_2, d_3) 作为结果。

否则，译码出现了错误，首先将 z 指示的位置的比特翻转（ z 的最左端为最低位），

之后，提取出 r 中的 (d_0, d_1, d_2, d_3) 作为结果。

例如，欲译码 $r = 00111111$ ：

$$z = Hr^T = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

因此，我们翻转位于第 $(011)_2 = 3$ 个位置的比特，即 $r' = 00011111$ 。

之后，我们提取出数据位，译码结果为 0111。

C1: 仿照这个工作过程，在实验报告给出 $[15, 11]$ 汉明码两个矩阵的推导过程，并且以矩阵乘法的形式，编程实现 $[15, 11]$ 汉明码。我们推荐你使用 `matlab` 或者一些矩阵运算库/包完成运算。过程的矩阵可以稀疏表示，即 0 的位置可用空白代替，只保留 1。

为了让你专注于算法的实现本身，在所有任务中你都可以认为比特串以包含一系列 0 和 1 的字符串组成（而不是将一个字节当成 8 个比特）

例如，比特串 $(00011111)_2$ 会是字符串 “00011111”，而非 `0x1f`。

为了验证你的实现，在 `lab2_data/hamming15_11.txt` 有标准的 $[15, 11]$ 汉明码的码表，其格式为每行一个（原始数据，编码结果），例如 000000000000，0000000000000000。

为了拿到 C1 的分数，你需要：

1. 验证：码表中的每个原始数据在使用你的程序编码后与编码结果相同。

（可以想办法把标准码表读进一个哈希表、字典等结构方便判断，也可

以构造相同格式的文件利用 diff 等命令比较)。

2. 验证：每个编码结果在使用你的程序解码后与原始数据相同。
3. 验证：修改每个编码结果中的一位，并使用你的程序解码，解码后的结果与原始数据相同。

C2:设想你要在一台资源（存储、运算）非常紧张的嵌入式设备实现[255, 247]汉明码，如果使用矩阵乘法，会出现什么问题？分别从时间和所需的存储空间考虑，在实验报告中给出你能想到的问题。

3.2 汉明码——奇偶校验观点

另一种理解汉明码的标准是从奇偶校验的角度进行的，我们给出标准的(7, 4)汉明码的工作过程；请仔细阅读并理解这个过程。

1. 将 1 到 15 写成二进制的形式：1, 10, ..., 1111。
2. 对于位置序号分别为 1, 2, ..., 7 的 7 个位置，我们将所有位置序号为 2^k 形式的位置用作奇偶校验位 p_k ，剩下的位置作为数据位 d_k 。
因此，奇偶校验位为第 1, 2, 4 位，数据位为第 3, 5, 6, 7 位。
最终传输的比特串为： $p_0 p_1 d_0 p_2 d_1 d_2 d_3$ 。
3. 编码时，先将数据依次填入数据位。

之后，对每个校验位 p_k ，我们通过如下方式选择它的值：从最低有效位算起，我们选出所有位置序号写成二进制后第 $k+1$ 位是 1 的位置，将这些位置的比特加起来，若为 1，则取 $p_k = 1$ ，反之则是 0。（注意到实际上就是使选出的位置 and 对应校验位恰有偶数个 1）

例如，对 p_0 ，对应的数据位是那些位置下标二进制表示下第 1 个位置是 1 的：1, 3, 5, 7（分别为 001, 011, 101, 111），所以要考虑的位置是第三个位置(d_0)，第五个位置(d_1)，第七个位置(d_3)，若这些位置上有偶数个 1，置 $p_0 = 0$ ，否则，置 $p_0 = 1$ 。

OP1 (1 分):对[7, 4]汉明码，证明：用上述方法编码比特串与线性代数观点中 $r = dG$ 的构造方法等价。（提示：对每个位，写出他的表达式）

这个编码过程可以进一步优化：

-
1. 将 1 到 15 写成等长的二进制的形式：0001, 0010, ..., 1111。
 2. 对于位置下标分别为 $1, 2, \dots, 7$ 的一个比特串，所有位置下标形如 2^k , $k \geq 0$ 的位作为奇偶校验位 p_k ，其他的位作为数据位 d_k 。

因此，奇偶校验位为第 1, 2, 4 位，数据位为第 3, 5, 6, 7 位。

最终传输的比特串为： $p_0 p_1 d_0 p_2 d_1 d_2 d_3$ 。

将数据依次填入数据位，先将奇偶校验位都填入 0。

3. 记录下所有值为 1 的比特的位置，对这些位置的位置下标进行异或，得到一个异或结果： $b_2 b_1 b_0$ 。置 $p_0 = b_0$, $p_1 = b_1$, $p_2 = b_2$ 。

例如，欲编码 0111，写出最终传输的比特串： $p_0 p_1 0 p_2 1 1 1$ 。

值为 1 的比特位置为：101, 110, 111 (第 5, 6, 7 位)，将他们异或得到 100，

因此 $p_0 = 0$, $p_1 = 0$, $p_2 = 1$ 。

传输的比特串为：0001111。

OP2 (1 分) :证明：这种构造方式与优化前的构造方式等价。(提示：异或运算相当于 \mathbb{F}_2 上的加法)

对这个过程稍做改动，便可以进行解码：

1. 将 1 到 15 写成等长的二进制的形式：0001, 0010, ..., 1111。
2. 对接收到的比特串： $p_0 p_1 d_0 p_2 d_1 d_2 d_3$ ：
3. 记录下所有值为 1 的比特的位置，对这些位置的下标进行异或，得到一个异或结果： $b = b_3 b_2 b_1$, b_1 为最低有效位。

如果 $b=0$ ，认为传输正确，提取出 $d_0 d_1 d_2 d_3$ 作为结果。

否则，位置为 b 的比特出错，将其翻转后，提取出 $d_0 d_1 d_2 d_3$ 作为结果。

例如，欲解码 0011111，值为 1 的比特位置为：011, 100, 101, 110, 111，对这些位置进行异或，得到一个异或结果 $b = (011)_2 = 3$ ，这说明第三个位置在传输过程中出现了错误。

因此将第三个位置的比特翻转，得到 0001111，提取出数据位，译码结果为 0111。

OP3 (2 分) :对[7, 4]汉明码，证明这种译码方式与线性代数观点中的译码

方式等价。

C3:使用上述的优化编码方法与译码方法，编程实现[15, 11]汉明码。

所有要求均与 C1 相同。

C4:同样地，设想你要在一台资源（存储、运算）非常紧张的嵌入式设备实现(255, 247)汉明码，使用上述的奇偶校验观点会有什么优势？分别从时间和所需的存储空间考虑，在实验报告中给出解释。

3.3 汉明码——直观体会

在这一部分，你可以直观理解到汉明码的实际应用。

C5: 修改你的程序，使其通过测试。

注意：本题应采取字符型输出，并且在 windows 下应该使用 cmd 运行而非 powershell!!!

选择你在 C2 或者 C3 实现的编码流程，修改它为如下的结构：

从标准输入读入只包含 0 和 1 的字符串，每读满 11 个字符，便将其编码，向标准输出写 15 个只包含 0 和 1 的编码字符串。持续进行，直到没有数据读入。测试程序保证你读入的字符串长度是 11 的倍数。

选择你在 C2 或者 C3 实现的译码流程，修改它为如下的结构：

从标准输入读入只包含 0 和 1 的字符串，每读满 15 个字符，便将其译码，向标准输出写 11 个只包含 0 和 1 的译码结果字符串。持续进行，直到没有数据读入。

之后，在终端或 cmd 键入下列命令运行测试，所有的测试都是确定的（包括虚拟的噪声信道），如果你的测试结果正确，下面所有命令的输出会是 Verify:ok。

此外，程序会生成两张图片，分别是信源数据的可视化、经过这个虚拟信道后的结果可视化。

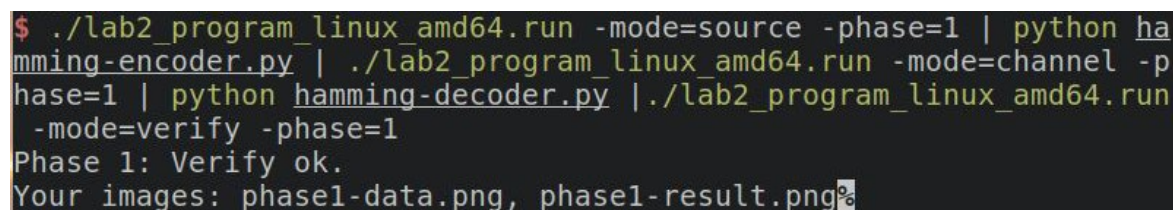
两张图片宽度均为 330 像素，高度为 30 像素的整数倍，从左到右，从上到下地，每个 30*30 的区域代表一个比特。图片中，黑色代表 1，白色代表 0。

你的实验报告应该包含所有的可视化图片和命令的运行结果。

C5.1 标准

请在一行内根据你的操作系统使用下列命令执行测试，将 `your_encode_program` 和 `your_decode_program` 替换为你的编码和解码程序；如果你只有 32 位系统/使用 BSD 系的发行版，请联系助教。

一个 Linux 下的运行示例如图：



```
$ ./lab2_program_linux_amd64.run -mode=source -phase=1 | python hamming-encoder.py | ./lab2_program_linux_amd64.run -mode=channel -phase=1 | python hamming-decoder.py | ./lab2_program_linux_amd64.run -mode=verify -phase=1
Phase 1: Verify ok.
Your images: phase1-data.png, phase1-result.png
```

Linux:

```
./lab2_program_linux_amd64.run -mode=source -phase=1 | your_encode_program
| ./lab2_program_linux_amd64.run -mode=channel -phase=1 |
your_decode_program | ./lab2_program_linux_amd64.run -mode=verify -phase=1
```

Mac(没有 mac OS 的设备，不知道交叉编译的能不能用，也不知道怎么交叉编译到 M1，所以如果没法执行请积极联系助教):

```
./lab2_program_mac_amd64 -mode=source -phase=1 | your_encode_program
| ./lab2_program_mac_amd64 -mode=channel -phase=1 | your_decode_program
| ./lab2_program_mac_amd64 -mode=verify -phase=1
```

Windows:

```
lab2_program_windows_amd64.exe -mode=source -phase=1 | your_encode_program
| lab2_program_windows_amd64.exe -mode=channel -phase=1 |
your_decode_program | lab2_program_windows_amd64.exe -mode=verify -phase=1
```

上面的命令在终端虚拟了一个噪声信道，`lab2_program` 在 `sourcemode` 下是一个虚拟信源，他产生一系列数据；之后，数据交由你的编码程序编码后发送到虚拟信道；`lab2_program` 在 `channelmode` 下是一个虚拟信道，他会为数据添加噪声；添加噪声的数据由你的程序解码后，解码结果被送至 `verifymode` 的 `lab2_program`，用于测试你的编码、解码程序实现的正确性，并给你一个直观的可视化结果。（如果有兴趣，可以试试将信道编码和解码程序去掉，即直接将虚拟信源的数据流经过虚拟信道传送到验证程序，这可以让你更直观体会到信道编码的作用）。

注意：

(1) 命令中的 `your_encode_program` 需要输入程序的执行命令，如 `python`

encode.py 或 java encode 等，而不是只上传程序。

(2) 注意程序的输出需要标准输出，并且只包含 0、1 字符串，不应包含任何空白字符。注意输出编码方式。

对 C5.1，噪声信道满足：每 15 个比特都会有 1 个比特出错。

在实验报告回答：信源数据和解码后的图片有（或者没有）什么差异？为什么会这样？

C5.2——2 比特错误

Linux:

```
./lab2_program_linux_amd64.run -mode=source -phase=2 | your_encode_program  
| ./lab2_program_linux_amd64.run -mode=channel -phase=2 |  
your_decode_program|. /lab2_program_linux_amd64.run -mode=verify -phase=2
```

Mac:

```
./lab2_program_mac_amd64 -mode=source -phase=2 | your_encode_program  
| ./lab2_program_mac_amd64 -mode=channel -phase=2 | your_decode_program  
| ./lab2_program_mac_amd64 -mode=verify -phase=2
```

Windows:

```
lab2_program_windows_amd64.exe -mode=source -phase=2 | your_encode_program |  
lab2_program_windows_amd64.exe -mode=channel -phase=2 | your_decode_program|  
lab2_program_windows_amd64.exe -mode=verify -phase=2
```

对 C5.2，噪声信道满足：每 30 个比特中，前 15 个比特有 1 个比特出错，后 15 个比特有 2 个比特出错。

信源满足：每 22 个比特中，前 11 个比特和后 11 个比特完全相同。

在实验报告回答：信源数据和解码后的图片有（或者没有）什么差异？为什么会这样？有哪些分组是解码正确的？

C5.3——突发错误

Linux:

```
./lab2_program_linux_amd64.run -mode=source -phase=3 | your_encode_program  
| ./lab2_program_linux_amd64.run -mode=channel -phase=3 |  
your_decode_program|. /lab2_program_linux_amd64.run -mode=verify -phase=3
```

Mac:

```
./lab2_program_mac_amd64 -mode=source -phase=3 | your_encode_program  
| ./lab2_program_mac_amd64 -mode=channel -phase=3 | your_decode_program  
| ./lab2_program_mac_amd64 -mode=verify -phase=3
```

Windows:

```
lab2_program_windows_amd64.exe -mode=source -phase=3 | your_encode_program |  
lab2_program_windows_amd64.exe -mode=channel -phase=3 | your_decode_program  
| lab2_program_windows_amd64.exe -mode=verify -phase=3
```

对对 C5.3，噪声信道满足：每 45 个比特中，前 15 个比特有连续的 3 比特出错，中间 15 个比特和后 15 个比特不会出错。

在实验报告回答：信源数据和解码后的图片有（或者没有）什么差异？为什么会这样？有哪些分组是解码正确的？

3.4 OP4——2 比特错误检测

上课已经提到，普通汉明码的最小距离为 3。

因此，如果编码后的比特串出现了两个错误，解码时，会错误地纠错为与出错的比特串距离为 1 的比特串，而解码程序完全不会意识到自己纠错后的结果是错误的。（推荐使用你编写的[15,11]汉明码做一些尝试）

在很多通信场合中，未发现错误比发现错误但无法改正更致命（前者会使数据出现问题，而后者能及时请求重传），因此，发现 2 比特错误（而非错误地纠错）十分重要。

方法有很多，包括使用数据完整性校验（例如 CRC）；这里要求你实现的是称为“扩展”汉明码的技术。（这项技术叫做 SEC-DED，被广泛地运用于 ECC 内存纠错，在要求高鲁棒性的服务器内存中极其常用）

扩展汉明码在一般汉明码的基础上加入一个总校验位 p ，以对(7,4)汉明码扩展为(8,4)汉明码为例：

1. 编码时，使用一般的汉明码构造流程，得到 $p_0p_1d_0p_2d_1d_2d_3$ 。

之后，在最后端插入一个总校验位，调整这个比特的值，使比特串中出现偶数个 1。例如，对汉明码 0001111，在最后添加一个 0，得到 00011110，即为扩展汉明码。

2. 解码时，首先忽视总校验位进行一般汉明码的检测，之后，分几种情况（以接收到扩展汉明码 00011110 为例）：
 - a) 一般汉明码检出了错误，总校验位为 1：认为仅有一个错误发生，按正常的汉明码纠错后提取数据输出。

- b) 一般汉明码检出了错误，总校验位为 0：认为有两个错误发生，输出错误（无法纠错）。
- c) 一般汉明码未检出错误，总校验位为 1：总校验位错误，直接提取数据输出。
- d) 一般汉明码未检出错误，总校验位为 0：认为没有错误发生。

OP4.1（2 分）：对[7,4]汉明码，证明上述的算法能正确纠正一个错误，检出两个错误，你可以认为错误的数量不会超过 2。（提示：尝试翻转一个比特或两个比特，分析几个校验位和总校验位会如何变化）

OP4.2（3 分）：仿照上述过程，在你的程序实现[16,11]扩展汉明码。输入一些数据测试，确保他能纠正一个错误，检出两个错误。

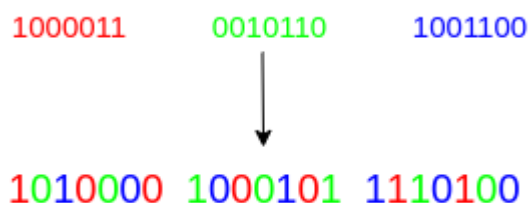
3.5 OP5——突发错误

信道的特性并非一成不变的，很多信道的噪声是突发性的。

具体来说，在大部分时候，信道都能当作可靠信道，不会发生翻转；但是，在少数时候，信道会受干扰，造成连续几个比特错误的发生。

尽管平均来看，突发错误信道的错误概率并没有更高，但因为信道的特点，汉明码无法直接用于突发错误信道。（你可以从 C5.3 看到这一点）

为了解决这个问题，一种方法是换用支持检查、纠正突发错误的编码（如 Reed-Solomon 编码，这种编码大量用于 CD 的纠错）；另一种方法是通过比特交织的技术。具体来说，我们将多个比特分组混杂起来发送，在接收端将比特分组恢复，使用这种策略，我们将突发错误分散到了多个比特分组，一个例子如图：



任何一个连续三比特的突发错误都会被分摊到三个分组处理，因而可以纠错连续三比特的突发错误。

OP5（4 分）：修改你的程序，实现进行三个分组的比特交织的 [15, 11] 汉明码，用它重新进行 C5.3 的测试，确保输出的两张图片相同（你会看到程序输

出 Verify: failed, 这是正常行为)。

或(6分): 实现 Reed-Solomon 编码, 选择一些带有突发错误的测试数据, 测试 Reed-Solomon 编码对突发错误的纠错性能。(Reed-Solomon 比起汉明码在软件领域的实际运用多很多, 并且是 MDS 码, 推荐有兴趣都试着做一做)。

3.6 OP6——噪声数目上限恒定的信道

我们考虑利用信道编码解决一类问题, 我们这里考虑一类噪声的数目不随信息总长增加而增加的信道, 这意味着, 在传输过程中错误出现的次数不超过某个常数限制 K 。

为了让同学们直观理解, 我们以 ulam 游戏为例子, 回答者选择 $[1, N]$ 中的一个整数 X , 提问者每次提问都可以选择一个整数集 S , 回答者回答 X 是否在 S 中, 回答者有 K 次撒谎的机会, 并且在过程中可以更改 X , 但必须保证最后的结果存在, 我们有两种模式, 分别是交互式(实时回答)和非交互式(问题一起传输, 结果一起传回)。我们想要知道提问者需要提问多少次才能保证知道 X 。

我们研究 $K = 0$ or 1 的情况。

$K = 0$ 时, 一个有效的询问方法是每次询问 X 的一个二进制位, 当回答者无法撒谎时, 我们在 $\lceil \log(N) \rceil$ 次询问后一定可以知道 X 。

延续之前思路, 我们去考虑 $K = 1$ 的非交互情况, 一个比较直接的方式是我们对同一个位置直接询问 3 次, 然后取回答次数多的答案, 但是这样显然会带来很多不必要的浪费。

op6(4+2 分): 在之前的思路, 去思考用信道的方式重新描述这个问题, 我们如何编码可以减少询问次数。(这一步可以只在实验报告中描述即可, 但给出你所提出方案的询问次数, 越小越好), 进一步可以使用之前实现的成果实现你的方案。

提示: 将撒谎理解为噪声/不要脱离实际背景, 一次询问实际得到的只有一位 0/1 表示是或否, 但是我们可以将连续的询问的回答看成一串需要编码的字符串。

Bonus: $K \geq 2$ 的情况我们能使用汉明码吗? 为什么。 /交互形式下 $K = 1$

的情况有没有更好的方案呢

4 实验提交

实验所需的文件位于 lab2_release 文件夹的 lab2_data 中：

<https://bhpan.buaa.edu.cn:443/link/8D7E064977663D36CA11E446BCAB2EA9>

你需要提交包含实验报告、程序源代码和过程文件（用于测试你的程序的编码后、解码后的文件）的压缩包。

请使用 **7z**、**rar** 等格式，或在 **utf-8locale** 下使用 **tar.gz**，不要使用 **zip**。

实验报告只需要包含运行源代码所需要的依赖；对必选模组代码的必要解释、运行测试截图；对可选模组代码的必要解释、运行测试截图；实践问题的运行截图和解答。

实验报告并无特定的格式，但需要转换成 **pdf**。只要实验报告包含所有必要的内容和解释，它便不会影响你的分数，字数多的实验报告不会给你带来任何加分。你需要将压缩包命名为 2023 信息论与编码-实验 2-学号-姓名，例如 2023 信息论与编码-实验 2-21371111-神秘人.7z，你可以运行 lab2_data/check.py 检查你的文件名是否合乎规范。

在截止日期前，你可以无限次地修改并重新上传压缩包（只要保持文件名一样，便可覆盖上传）。

请不要覆盖别人（例如你的室友）的作业。

在确认无误后，将你的压缩包上传至：

<https://bhpan.buaa.edu.cn:443/link/49278B2B670C477E0E803BA1ED9D059F>

此外，还需要在雨课堂提交对应的实验作业（不需要在雨课堂上传任何内容，仅仅为了方便评分）。

你应该在北京时间 **2023 年 5 月 31 日 23:59** 前完成上述的提交。