信息论与编码大作业

贾淞淋　516021910673　林宇涛　516021910106
吴怡琳　516021910334　赵彧然　516021910267

一、编码算法基本介绍

本次作业中，我们使用了 Huffman 编码和 LZ 编码分别对所给文件进行压缩处理，下面分别介绍两种编码算法。

1.1 LZ 编码算法介绍

之前很多信源编码方式都需要精确已知信源的概率分布，一旦信源的实际分布与假设的分布有差异，编码性能就会急剧下降。但是在实际应用中，确切地获知信源统计特性有时是非常困难的，有时候信源统计特性还会随时间发生变化，因此需要一种与信源统计特性无关的信源编码方法。LZ 编码即属于这类编码。

LZ 编码基本原理

假设信源符号集 $A = \{a_1, a_2, \cdots, a_k\}$ 共 K 个符号，输入信源符号为 $(u_1, u_2, \cdots, u_L)$，LZ 编码将此序列分成不同的段进行处理。

分解是迭代进行的：在第 i 步，编码器从 s_{i-1} 短语之后的第一个符号开始向后搜索，寻找字典中之前从未出现过的最短短语作为 s_i，并将它添入字典第 i 段。由于 s_i 是此时字典中最短的新短语，所以 s_i 去掉最后一个符号 x 后的字符串一定已经在字典中出现过，假设它是在第 j（< i) 步被写入字典的，则对 s_i 进行编码时就可以利用 j 和 s_i 最后一位字符 x 来共同表示，即为码字（j, x）。对于段号 j，最多需要 $\lceil logi \rceil$ bit 来表示，而符号 x 则只需要 $\lceil logK \rceil$ bit。

LZ 编码的译码

译码无需字典，可以一边译码一边建立字典。记录字典当前要添加的行号 i，读入序列中 $\lceil logi \rceil$ bit 个符号，对应的值即为前缀所在的段号；再读入 $\lceil logK \rceil$ bit 符号，此即 s_i 最后的一个字符。在字典中搜索前缀段号对应的短语作为前缀，拼接上字符 x 即可组合出第 s_i 个短语，并将它写入字典的第 i 行。

1.2 Huffman 编码算法介绍

Huffman 编码基本原理

Huffman 编码是分组编码，完全依据各字符出现的概率来构造码字。其基本原理是基于二叉树的编码思想，所有可能的输入符号在 Huffman 树上对应为一个叶子结点，结

点的位置即为该符号的 Huffman 编码。因此 Huffman 编码是唯一可译码，不会出现前缀码。

以二进制编码为例，具体编码方法如下：

(1) 将信源消息符号按其出现的概率大小依次排列：

$$p_1 \geqslant p_2 \geqslant \cdots \geqslant p_n$$

(2) 取两个概率最小的字母分别配以 0 和 1 两个码元，并将这两个概率相加作为一个新字母的概率，与未分配二进制符号的字母一起重新排队。

(3) 对重排后的两个概率最小的符号重复步骤（2）的过程。

(4) 不断重复上述过程，直到最后两个符号概率之和为 1 为止，对其分别配以 0 和 1。

(5) 从最后一级开始，向前返回得到各个信源符号所对应的码元序列，即相应的码字。

Huffman 编码的译码

在对 Huffman 编码进行译码时，只需要对应编码时生成的码表将码字一一还原为其对应的字符即可。

二、编码算法实现思路

2.1 LZ 编码算法

LZ 算法实现的具体思路如图1所示。

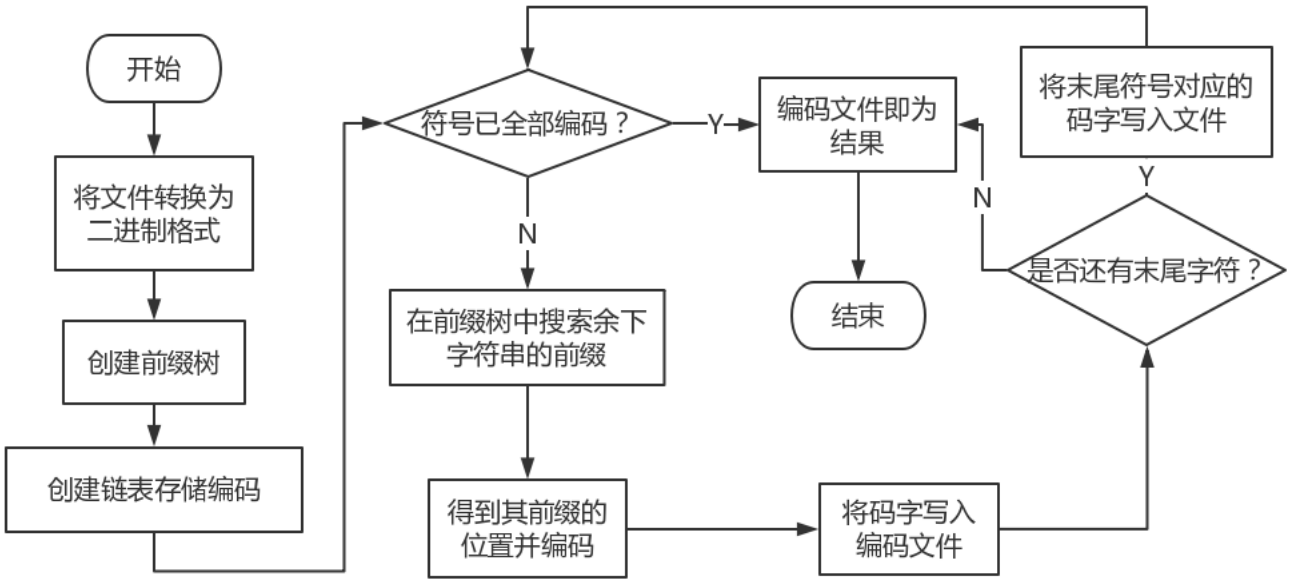


图 **1** **LZ** 编码算法

由于 LZ 编码需要不断搜索当前字符串的前缀，自然而然地，我们想到使用前缀树来方便查找。除此以外，我们先将文件转换为二进制，这样可以使用不同长度来考察编码效率，且末尾符号只有 ‘0’ 和 ‘1’ 两种符号，方便编码。

2.2 Huffman 编码算法

Huffman 算法实现的具体思路如图2所示。

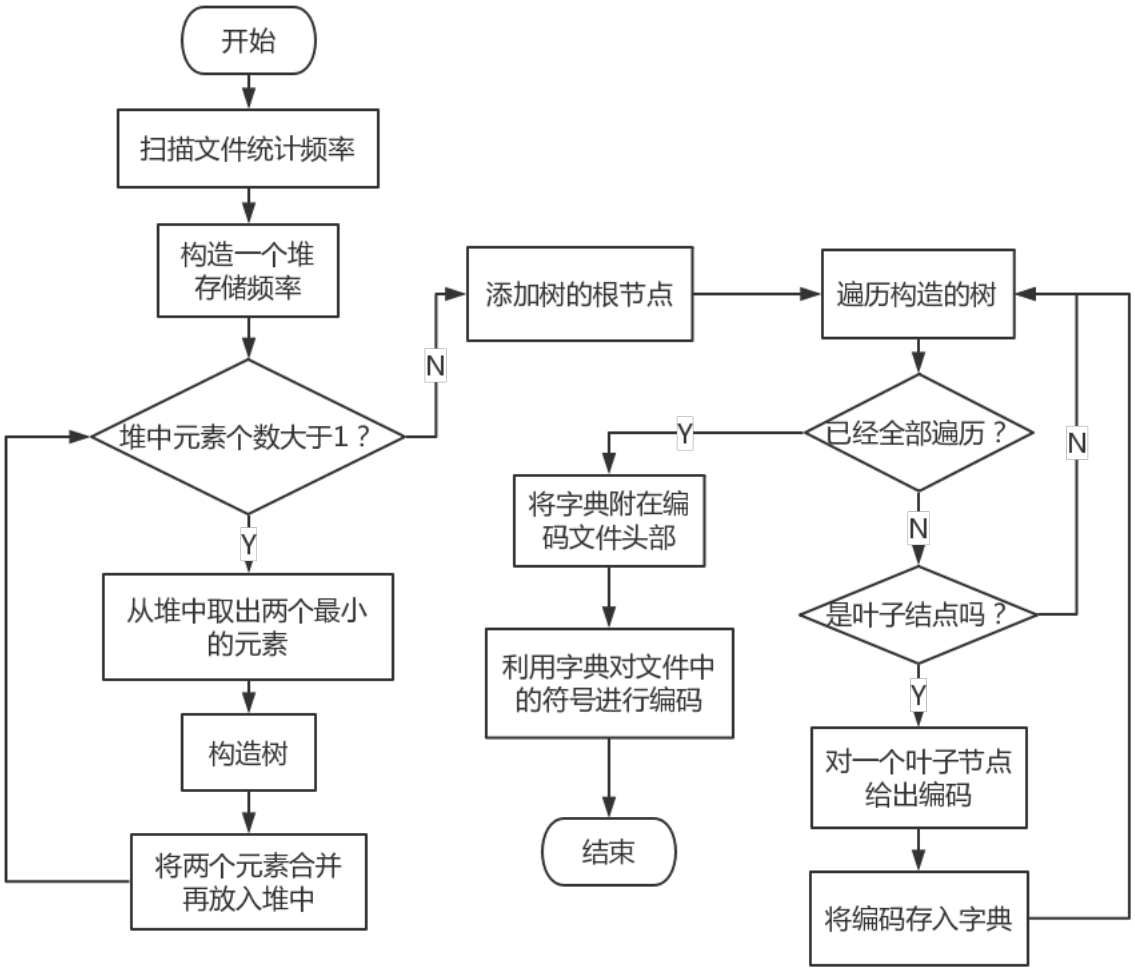


图 **2** **Huffman** 编码算法

在实现细节上：算法中地堆使用的是优先队列的结构。由于 Huffman 编码过后可能不是一个完整的字节，因而在末尾可能需要用 ‘0’ 补齐，这样会再添加一个字节数据用来表示填充的位数。

对 Huffman 编码的译码只需要依照编码字典，将码字还原为字符即可，但要注意的是在译码之前需要将文件中的字典先去除，具体步骤在此不再赘述。

三、编码结果

3.1 LZ 编码

使用 LZ 编码时，我们对转换得到的二进制文件分别尝试以 1bit、2bit、4bit 和 8bit 为单位进行编码，以对比不同编码单位下的结果。

在这里我们只展示以 1bit 为单位进行编码的结果，其余的编码结果可以在附录中查看。

txt 文件

以 1bit 作为单位进行压缩得到的编解码结果如图3所示，从上到下依次为：源文件、编码文件、解码文件。

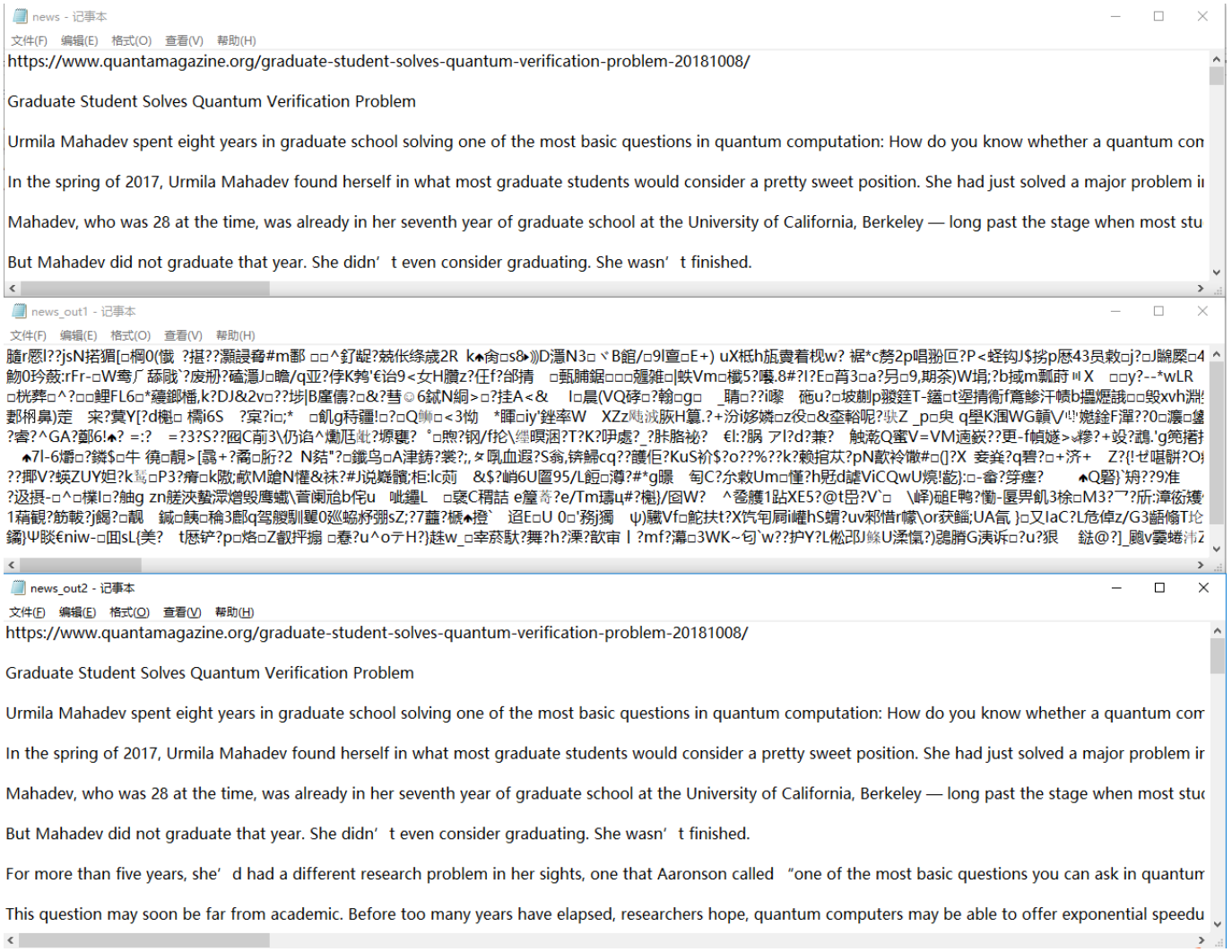


图 3　LZ 编解码（1bit）压缩 txt 文件结果

而其编码得到的 ‘0’、‘1’ 字符串则如图4所示。

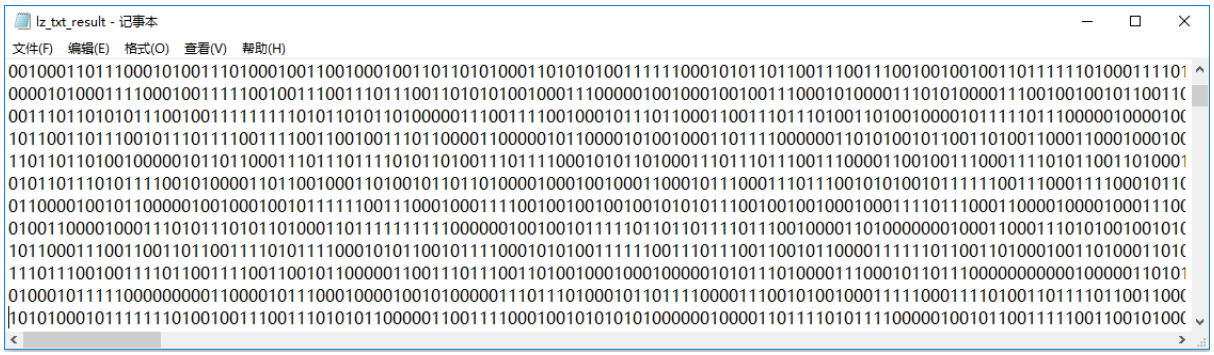


图 **4** **LZ** 编码（**1bit**）对 **txt** 文件处理结果

docx 文件

以 1bit 作为单位进行压缩得到的编码结果如图5所示，从上到下依次为：源文件、编码文件、解码文件。这里由于以编码形式写入的 docx 文件后不可读，为展示方便，将它写入了一个 txt 文件中。

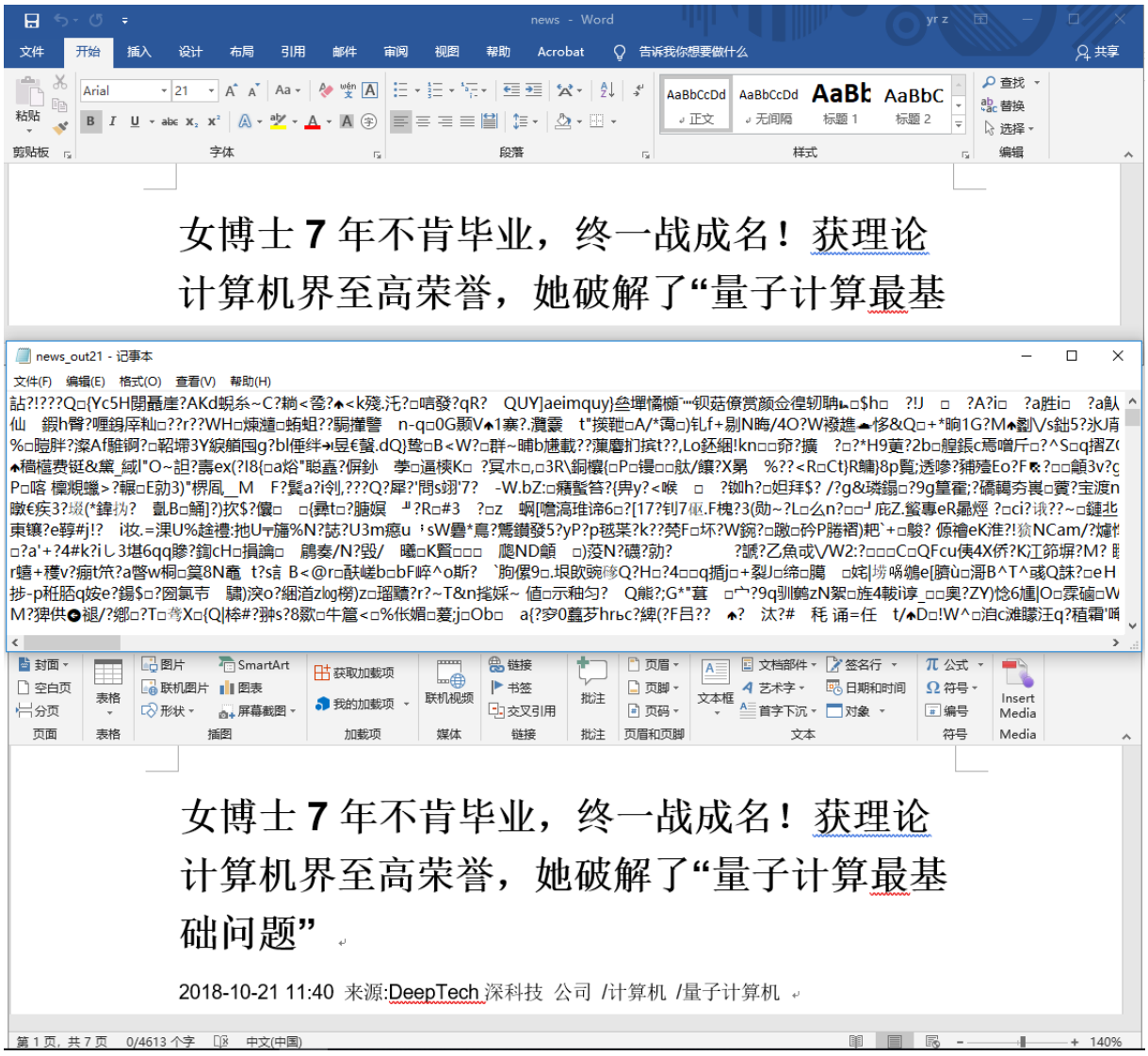


图 **5**　**LZ** 编解码（**1bit**）压缩 **docx** 文件结果

而其编码得到的 ‘0’、‘1’ 字符串则如图6所示。

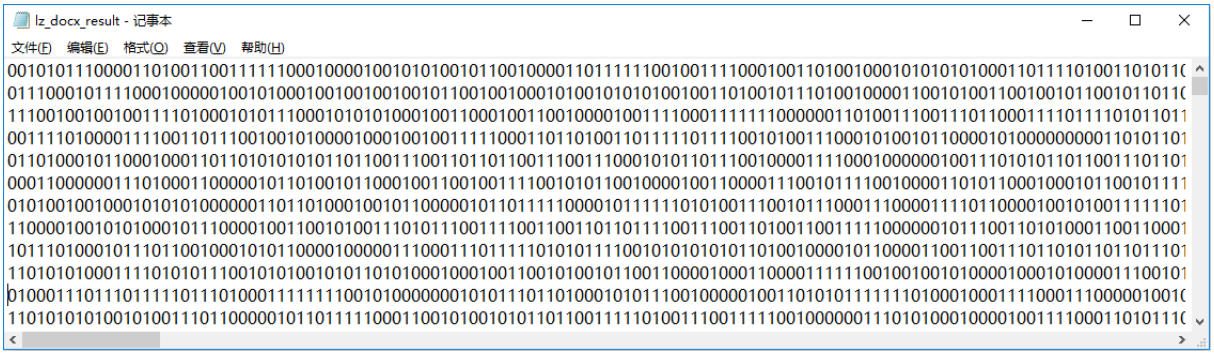


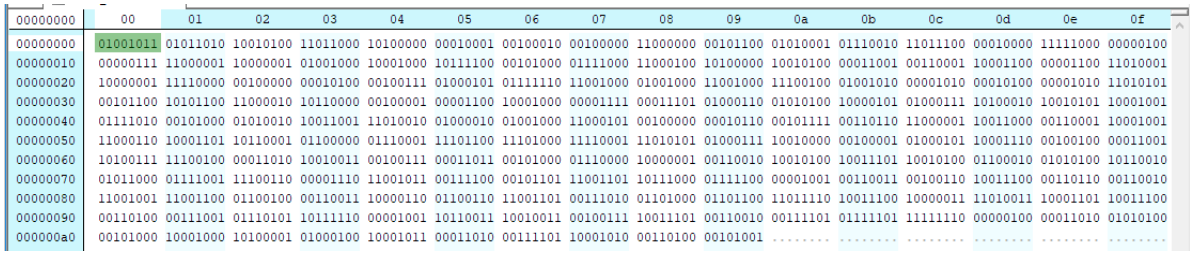
图 **6**　**LZ** 编码（**1bit**）对 **docx** 文件处理结果

3.2 Huffman 编码

Huffman 编码时，对二进制读入的文件以 8 bit 为单位进行频率统计，进而编码。Huffman 编码中我们更关心字典，因此主要列出字典的结果。

txt 文件

使用 Huffman 编码得到的字典为：



00000000	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00000000	01001011	01011010	10010100	11011000	10100000	00010001	00100010	00100000	11000000	00101100	01010001	01110010	11011100	00010000	11111000	00000100
00000010	00000111	11000001	10000001	01001000	10001000	10111100	00101000	01111000	11000100	10100000	10010100	00011001	00110001	10001100	00001100	11010001
00000020	10000001	11110000	00100000	00010100	00100111	01000101	01111110	11001000	01001000	11001000	11100100	01001010	00001010	00010100	00001010	11010101
00000030	00101100	10101100	11000010	10110000	00100001	00001100	10001000	00001111	00011101	01000110	01010100	10000101	01000111	10100010	10010101	10001001
00000040	01111010	00101000	01010010	10011001	11010010	01000010	01001000	11000101	00100000	00010110	00101111	00110110	11000001	10011000	00110001	10001001
00000050	11000110	10001101	10110001	01100000	01110001	11101100	11101000	11110001	11010101	01000111	10010000	00100001	01000101	10001110	00100100	00011001
00000060	10100111	11100100	00011010	10010011	00100111	00011011	00101000	01110000	10000001	00110010	10010100	10011101	10010100	01100010	01010100	10110010
00000070	01011000	01111001	11100110	00001110	11001011	00111100	00101101	11001101	10111000	01111100	00001001	00110011	00100110	10011100	00110110	00110010
00000080	11001001	11001100	01100100	00110011	10000110	01100110	11001101	00111010	01101000	01101100	11011110	10011100	10000011	11010011	10001101	10011100
00000090	00110100	00111001	01110101	10111110	00001001	10110011	10010011	00100111	10011101	00110010	00111101	01111101	11111110	00000100	00011010	01010100
000000a0	00101000	10001000	10100001	01000100	10001011	00011010	00111101	10001010	00110100	00101001	....	....	....	....	....	....

图 7　**Huffman** 编解码压缩 **txt** 文件字典

进行压缩得到的编解码结果如图所示，上面为源文件，下面的则是解码文件。

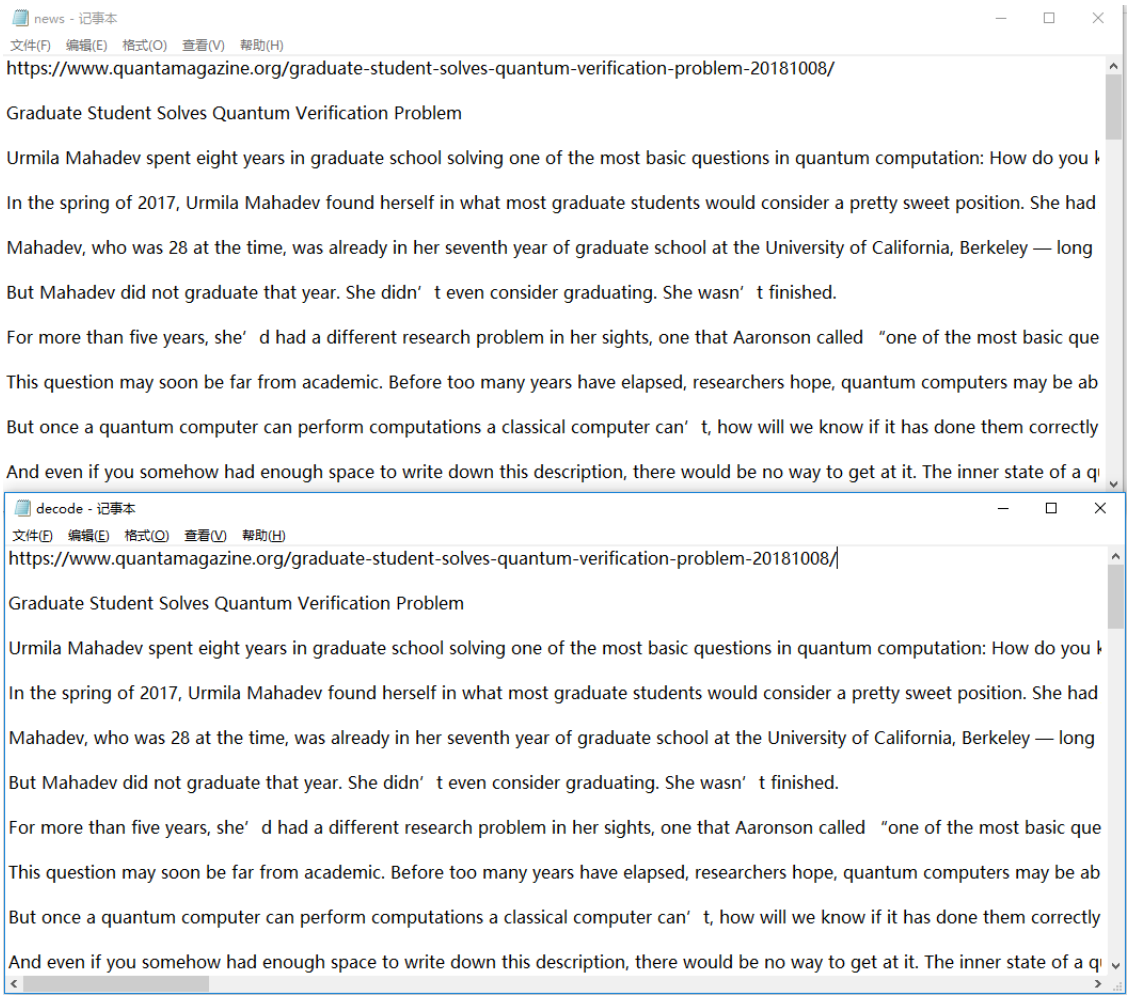
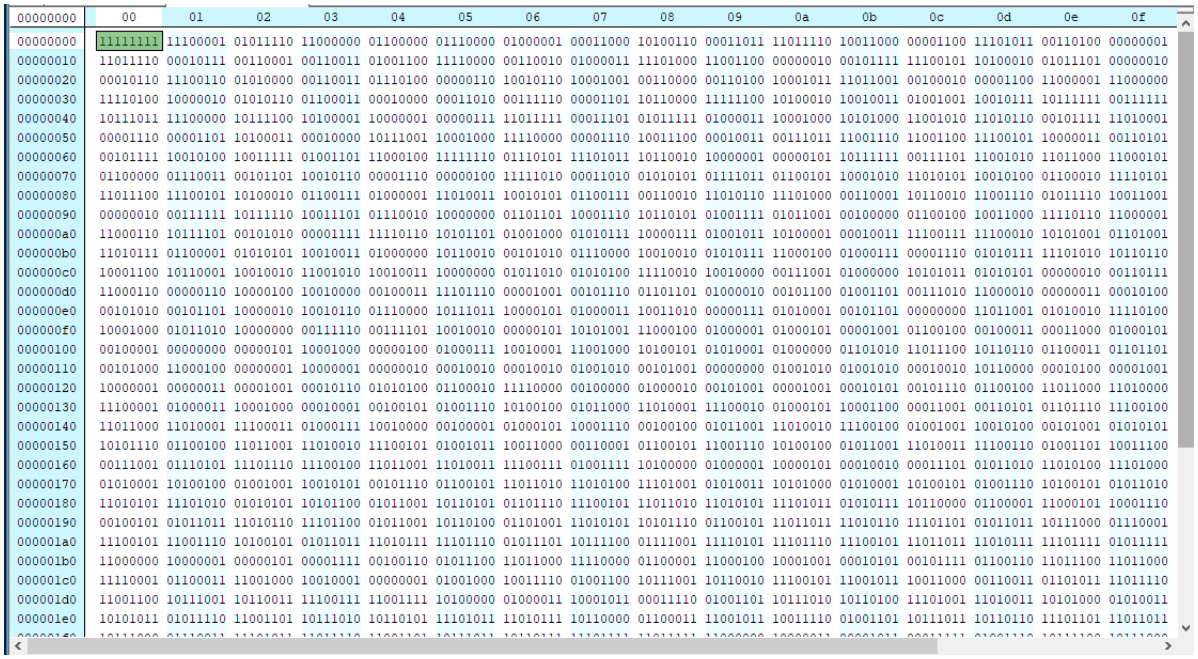


图 8　**Huffman** 编解码 **txt** 文件结果

docx 文件

使用 Huffman 编码得到的字典为：



00000000	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00000000	11111111	11100001	01011110	11000000	01100000	01110000	01000001	00011000	10100110	00011011	11011110	10011000	00001100	11101011	00110100	00000001
00000010	11011110	00010111	00110001	00110011	01001100	11110000	00110010	01000011	11101000	11001100	00000010	00101111	11100101	10100010	01011101	00000010
00000020	00010110	11100110	01010000	00110011	01110100	00000110	10010110	10001001	00110000	00110100	10001011	11011001	00100010	00001100	11000001	11000000
00000030	11110100	10000010	01010110	01100011	00010000	00011010	00111110	00001101	10110000	11111100	10100010	10010011	01001001	10010111	10111111	00111111
00000040	10111011	11100000	10111100	10100001	10000001	00000111	11011111	00011101	01011111	01000011	10001000	10101000	11001010	11010110	00101111	11010001
00000050	00001110	00001101	10100011	00010000	10111001	10001000	11110000	00001110	10011100	00010011	00111011	11001110	11001100	11100101	10000011	00110101
00000060	00101111	10010100	10011111	01001101	11000100	11111110	01110101	11101011	10110010	10000001	00000101	10111111	00111101	11001010	11011000	11000101
00000070	01100000	01110011	00101101	10010110	00001110	00000100	11111010	00011010	01010101	01111011	01100101	10001010	11010101	10010100	01100010	11110101
00000080	11011100	11100101	10100010	01100111	01000001	11010011	10010101	01100111	00110010	11010110	11101000	00110001	10110010	11001110	01011110	10011001
00000090	00000010	00111111	10111110	10011101	01110010	10000000	01101101	10001110	10110101	01001111	01011001	00100000	01100100	10011000	11110110	11000001
000000a0	11000110	10111101	00101010	00001111	11110110	10101101	01001000	01010111	10000111	01001011	10100001	00010011	11100111	11100010	10101001	01101001
000000b0	11010111	01100001	01010101	10010011	01000000	10110010	00101010	01110000	10010010	01010111	11000100	01000111	00001110	01010111	11101010	10110110
000000c0	10001100	10110001	10010010	11001010	10010011	10000000	01011010	01010100	11110010	10010000	00111001	01000000	10101011	01010101	00000010	00110111
000000d0	11000110	00000110	10000100	10010000	00100011	11101110	00001001	00101110	01101101	01000010	00101100	01001101	00111010	11000010	00000011	00010100
000000e0	00101010	00101101	10000010	10010110	01110000	10111011	10000101	01000011	10011010	00000111	01010001	00101101	00000000	11011001	01010010	11110100
000000f0	10001000	01011010	10000000	00111110	00111101	10010010	00000101	10101001	11000100	01000001	01000101	00001001	01100100	00100011	00011000	01000101
00000100	00100001	00000000	00000101	10001000	00000100	01000111	10010001	11001000	10100101	01010001	01000000	01101010	11011100	10110110	01100011	01101101
00000110	00101000	11000100	00000001	10000001	00000010	00010010	00010010	01001010	00101001	00000000	01001010	01001010	00010010	10110000	00010100	00001001
00000120	10000001	00000011	00001001	00010110	01010100	01100010	11110000	00100000	01000010	00101001	00001001	00010101	00101110	01100100	11011000	11010000
00000130	11100001	01000011	10001000	00010001	00100101	01001110	10100100	01011000	11010001	11100010	01000101	10001100	00011001	00110101	01101110	11100100
00000140	11011000	11010001	11100011	01000111	10010000	00100001	01000101	10001110	00100100	01011001	11010010	11100100	01001001	10010100	00101001	01010101
00000150	10101110	01100100	11011001	11010010	11100101	01001011	10011000	00110001	01100101	11001110	10100100	01011001	11010011	11100110	01001101	10011100
00000160	00111001	01110101	11101110	11100100	11011001	11010011	11100111	01001111	10100000	01000001	10000101	00010010	00011101	01011010	11010100	11101000
00000170	01010001	10100100	01001001	10010101	00101110	01100101	11011010	11010100	11101001	01010011	10101000	01010001	10100101	01001110	10100101	01011010
00000180	11010101	11101010	01010101	10101100	01011001	10110101	01101110	11100101	11011010	11010101	11101011	01010111	10110000	01100001	11000101	10001110
00000190	00100101	01011011	11010110	11101100	01011001	10110100	01101001	11010101	10101110	01100101	11011011	11010110	11101101	01011011	10111000	01110001
000001a0	11100101	11001110	10100101	01011011	11010111	11101110	01011101	10111100	01111001	11110101	11101110	11100101	11011011	11010111	11101111	01011111
000001b0	11000000	10000001	00000101	00001111	00100110	01011100	11011000	11110000	01100001	11000100	10001001	00010101	00101111	01100110	11011100	11011000
000001c0	11110001	01100011	11001000	10010001	00000001	01001000	10011110	01001100	10111001	10110010	11100101	11001011	10011000	00110011	01101011	11011110
000001d0	11001100	10111001	10110011	11100111	11001111	10100000	01000011	10001011	00011110	01001101	10111010	10110100	11101001	11010011	10101000	01010011
000001e0	10101011	01011110	11001101	10111010	10110101	11101011	11010111	10110000	01100011	11001011	10011110	01001101	10111011	10110110	11101101	11011011

图 **9**　**Huffman** 编解码压缩 **docx** 文件字典

使用 Huffman 编码进行压缩得到的编解码结果如图所示，上面为源文件，下面的则是解码文件。

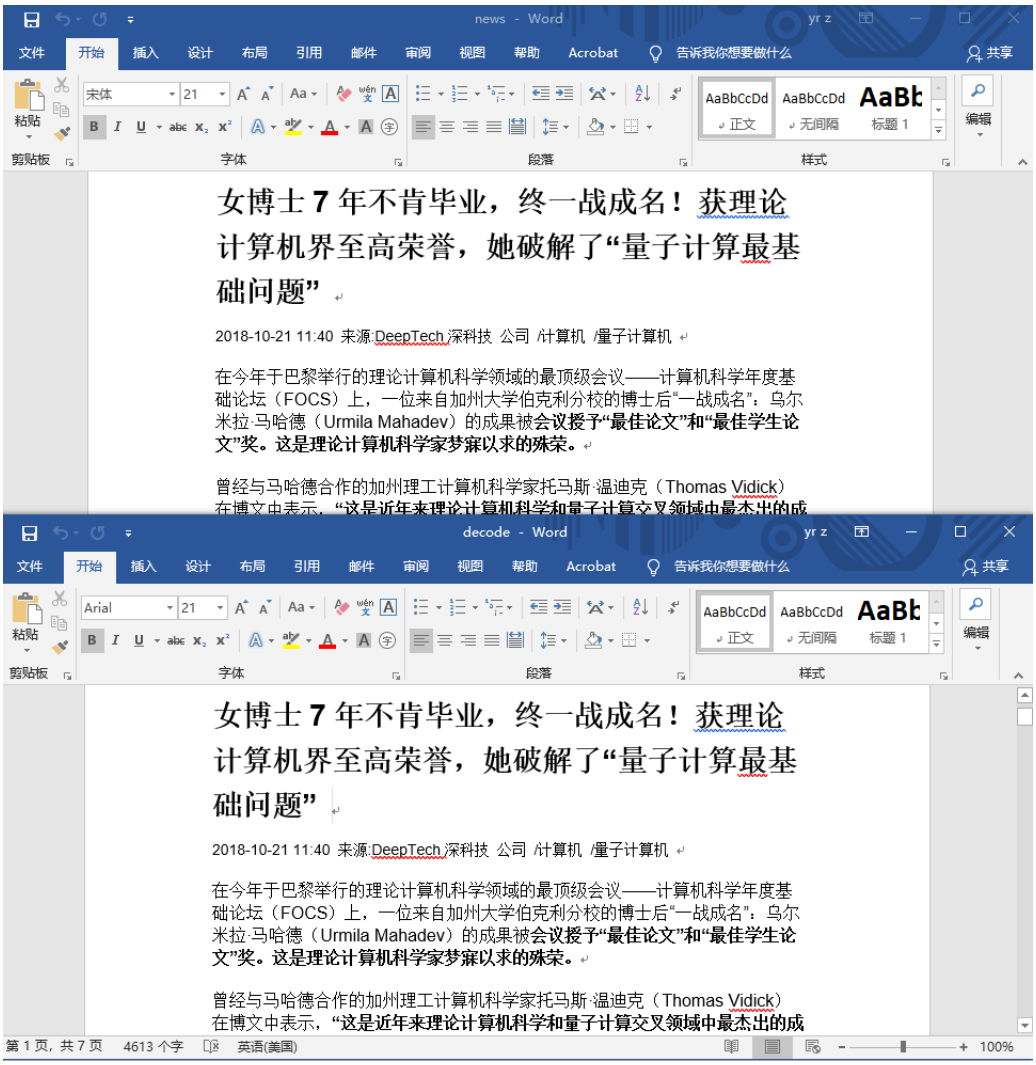


图 **10** **Huffman** 编解码 **docx** 文件结果

四、性能分析

性能考察主要从以下几方面着手：

- 编解码正确性：通过对比编解码后得到的符号串与源文件的符号串，得到编解码的正确率。这里中我们使用 Linux 中的 diff 命令来对比编解码一次后的输出文件与源文件，详细代码如下：

```
#!/bin/bash
TEMPFN1=temp2333.txt
TEMPFN2=temp4666.txt
base64 "$1" >$TEMPFN1
base64 "$2" >$TEMPFN2
diff $TEMPFN1 $TEMPFN2 && echo $1'' and $2'' are the
```

```
    same.
rm $TEMPFN1 $TEMPFN2
```

- 压缩比：计算源文件大小与压缩后的文件大小之比，当压缩比大于 1 时，说明文件得到了压缩；小于 1 则说明文件压缩失败。
- 编解码速度：利用多次（实验中每种编码算法的编解码次数均取 10）运行编解码程序计算平均值得到编解码所需时间来代表编解码速度的快慢，所耗费时间越短，编解码速度越快。

每种算法的具体结果如下所示。

4.1 LZ 编码

LZ 算法编解码的正确性检验如图11所示。

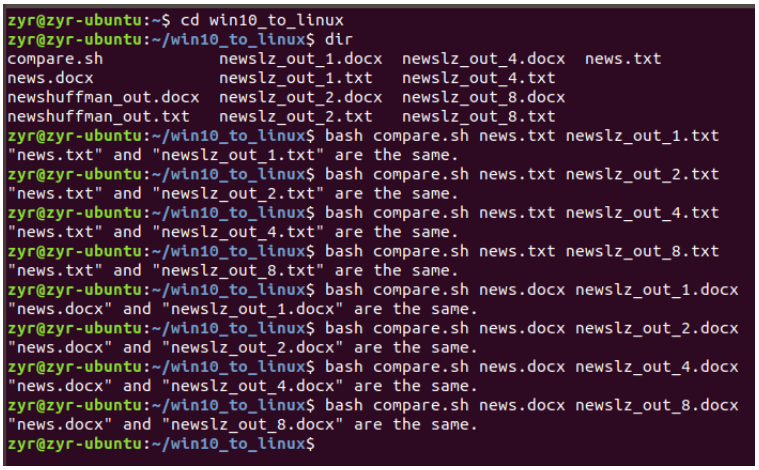


图 **11** **LZ** 编码正确性检验

LZ 算法的具体分析结果如下表所示：

表 **1** **LZ** 编码 **(1 bit)** 性能分析

	正确率	压缩比	编码速度（s/次）	解码速度（s/次）
.txt 文件 (17350 byte)	100%	1.09367	0.0168	0.269
.docx 文件 542915 byte)	100%	0.91164	0.6713	0.7543

表 **2** **LZ** 编码 **(2 bit)** 性能分析

	正确率	压缩比	编码速度（s/次）	解码速度（s/次）
.txt 文件 (17350 byte)	100%	1.31181	0.0112	0.0202
.docx 文件 542915 byte)	100%	0.930401	0.457	0.7196

表 **3** **LZ** 编码 **(4 bit)** 性能分析

	正确率	压缩比	编码速度（s/次）	解码速度（s/次）
.txt 文件 (17350 byte)	100%	1.52756	0.0091	0.0173
.docx 文件 542915 byte)	100%	0.920451	0.3102	0.6482

表 **4** **LZ** 编码 **(8 bit)** 性能分析

	正确率	压缩比	编码速度（s/次）	解码速度（s/次）
.txt 文件 (17350 byte)	100%	1.6568	0.0086	0.012
.docx 文件 542915 byte)	100%	0.886334	0.3676	0.5869

通过以上四个表格的对比，我们发现：

- 即使编码单位长度发生变化，LZ 编码总可以保证编解码的正确性。
- 随着编码单位的加长，txt 文件的编解码时间都有了显著的下降，且压缩比也不断增大。这是合理的，因为需要进行处理的数量在减少。
- 随着编码单位的加长，docx 文件的编码速度先降再升，在 4 bit 时有最大的编码速度；解码速度则不断加快。与此同时，文件的压缩比也先增大后减小，在 2 bit 时有最大的压缩比，但仍然无法对文件进行正向压缩（即压缩比大于 1）。
- 在每一组表格中，docx 文件的编解码速度都明显小于 txt 文件的编解码速度。这是因为首先 docx 文件远远大于 txt 文件，实验中 news.docx 文件的大小为 542915 字节，而 news.txt 文件仅有 17350 字节，因此在对 docx 文件进行操作时花费更长的时间是自然的。

随后我们对 docx 文件编码时使用的各前缀长度的数目进行了统计，如图12所示。

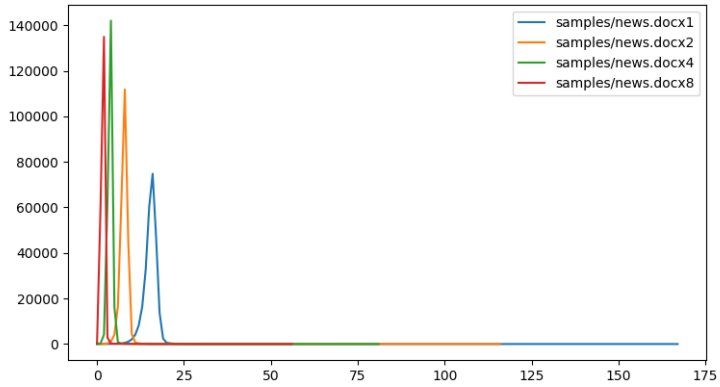


图 **12**　**LZ** 编码压缩 **docx** 文件时前缀利用情况

可以发现，随着编码单位长度的增加，那些长度更短的前缀被利用的次数不断增多，而且更加集中于利用更短的前缀。这种现象导致了 LZ 编码的优点没有得到充分利用，因此压缩比并不可观。

4.2　Huffman 编码

Huffman 算法编解码的正确性检验如图13所示。

```
zyr@zyr-ubuntu:~$ cd win10_to_linux
zyr@zyr-ubuntu:~/win10_to_linux$ dir
compare.sh            newslz_out_1.docx  newslz_out_4.docx  news.txt
news.docx             newslz_out_1.txt   newslz_out_4.txt
newshuffman_out.docx  newslz_out_2.docx  newslz_out_8.docx
newshuffman_out.txt   newslz_out_2.txt   newslz_out_8.txt
zyr@zyr-ubuntu:~/win10_to_linux$ bash compare.sh news.docx newshuffman_out.docx
"news.docx" and "newshuffman_out.docx" are the same.
zyr@zyr-ubuntu:~/win10_to_linux$ bash compare.sh news.txt newshuffman_out.txt
"news.txt" and "newshuffman_out.txt" are the same.
zyr@zyr-ubuntu:~/win10_to_linux$
```

图 **13**　**Huffman** 编码正确性检验

表 **5**　**Huffman** 编码性能分析

	正确率	压缩比	编码速度（s/次）	解码速度（s/次）
.txt 文件 (17350 byte)	100%	1.72637	0.0039	0.0238
.docx 文件 542915 byte)	100%	1.00031	0.0536	1.2666

由表格可知，我们的 Huffman 编码也可以保证编解码不发生错误。

除此以外，通过与 LZ 编码得到的结果进行对比，可以发现：

- Huffman 的编码速度十分优异，但是解码速度略逊于 LZ 编码。这一点应与程序的复杂度有关，不再对它进行更为详细的讨论。

- Huffman 编码的压缩比较为可观，且对 docx 文件实现了正向压缩，即压缩比大于 1。首先，Huffman 编码以 8 bit 为单位进行处理，因此压缩比相对前三种 LZ 编码自然更优；其次，实验所用到的两个文件本身比较大，Huffman 编码需要传输的码表的特点对编码效率的影响得到了降低，因而获得了更好的压缩比。
- 两种编码对于 txt 文件的压缩效果都要远好于 docx 文件。这是因为 txt 文本中主要是英文符号，种类较少，二进制文件中某些符号的出现概率可能更大；而 docx 文件中主要是中文符号，种类更多，因此其二进制文件中各种符号出现的概率更为平均。

五、总结与体会

通过这次对于 Huffman 编码和 LZ 编码的实现，我们对于课堂上所讲授的编码方法有了更深的理解，将课堂所学运用于实际之中，对于不同编码算法的实现及其性能都有了进一步的体会。在编码过程中，以二进制对文件进行读写操作、更多地利用位运算来加快运算速度等在平时不太注意的细节都得到了练习，使用 C++ 对文件进行操作虽然细碎繁琐，但同时也更加深了我们对于整个编码过程的理解。

在完成代码时也遇到了一些问题，比如：

- 读写文件时由于是小序端，因此需要将读入的序列先取为反向，写入前也要进行反向。
- 从比特串中提取子串时对序号的操作。

这些问题在和小组成员以及其他同学的讨论、请教中都最终得以解决，进一步地提高了自己的能力。除此之外，小组成员在完成作业的的过程中也取长补短，学习他人代码中的闪光点，最终受益匪浅。

六、C++ 源代码

作业中各编码算法的 C++ 源代码如下所示。

6.1 LZ 编码

trie.h

```
#ifndef _TRIE_H_
#define _TRIE_H_
#include <cstdint>
#include "binstr.h"

struct TrieNode {
```

```
    const size_t size_chars;
    uint32_t num;
    TrieNode* *next;
    TrieNode(uint32_t num, size_t sc);
    ~TrieNode();
};

class Trie {
    const size_t size_unit, size_chars;
    TrieNode *root;
    uint32_t cnt;
public:
    Trie(size_t su);
    ~Trie();
    TrieNode* find_path(const BinStr& bs, size_t& pos);
    void add_node(TrieNode *tn, uint32_t val);
};

#endif
```

binstr.h

```
#ifndef _BINSTR_H_
#define _BINSTR_H_
#include <cstdint>

class BinStr {
    size_t arr_size, bit_size;
    uint8_t *buf;
public:
    BinStr();
    BinStr(uint8_t *buf, size_t size);
    BinStr(const BinStr& bs);
    ~BinStr();
    size_t get_bit_size() const;
    uint32_t substr(size_t startpos, size_t len) const;
```

```
    void push_back(uint32_t val, size_t len);
    void push_back(const BinStr& bs);
    void fill();
    void unfill();
    char* copy_to_buffer(size_t & cnt_bytes);
};

#endif
```

tri.cpp

```
#include <cstring>
#include "trie.h"


TrieNode::TrieNode(uint32_t n, size_t sc): num(n),
   size_chars(sc) {
    next = new TrieNode*[size_chars];
    memset(next, 0, sizeof(TrieNode*) * size_chars);
}


TrieNode::~TrieNode() {
    for (size_t i = 0; i < size_chars; i++) {
        if (next[i]) {
            delete next[i];
        }
    }
    delete[] next;
}


Trie::Trie(size_t su): size_unit(su), size_chars(1 << su
   ) {
    cnt = 0;
    root = new TrieNode(cnt, size_chars);
```

```
}


Trie::~Trie() {
    delete root;
}


TrieNode* Trie::find_path(const BinStr& bs, size_t& pos)
    {
    TrieNode *cur = root;
    while (pos < bs.get_bit_size()) {
        uint32_t val = bs.substr(pos, size_unit);
        if (cur->next[val]) {
            cur = cur->next[val];
            pos += size_unit;
        } else {
            break;
        }
    }
    return cur;
}


void Trie::add_node(TrieNode *tn, uint32_t val) {
    tn->next[val] = new TrieNode(++cnt, size_chars);
}
```

binstr.cpp

```
#include <cstring>
#include "binstr.h"


BinStr::BinStr() {
    arr_size = 10;
```

```
    bit_size = 0;
    buf = new uint8_t[arr_size];
}


BinStr::BinStr(uint8_t *src, size_t src_bytes) {
    arr_size = src_bytes + 10;
    bit_size = (src_bytes << 3);
    buf = new uint8_t[arr_size];
    memcpy(buf, src, src_bytes);
}


BinStr::BinStr(const BinStr& bs) {
    arr_size = bs.arr_size;
    bit_size = bs.bit_size;
    buf = new uint8_t[arr_size];
    memcpy(buf, bs.buf, ((bit_size + 7) >> 3));
}


BinStr::~BinStr() {
    delete[] buf;
}


size_t BinStr::get_bit_size() const {
    return bit_size;
}


uint32_t BinStr::substr(size_t startpos, size_t len)
   const {
    size_t seg = startpos >> 3, off = startpos & 7, tail
        = (off + len + 7) >> 3;
    uint64_t data = 0, mask = ((1ull << len) - 1);
```

```
    for (size_t i = 0; i < tail; i++) {
        data |= buf[seg + i] << (i << 3);
    }
    data >>= off;
    return data & mask;
}


void BinStr::push_back(uint32_t val, size_t len) {
    size_t seg = bit_size >> 3, off = bit_size & 7, tail
        = (off + len + 7) >> 3;
    if (seg + tail > arr_size) {
        size_t new_size = (seg + tail) << 1;
        uint8_t *new_buf = new uint8_t[new_size];
        memcpy(new_buf, buf, arr_size);
        delete[] buf;
        arr_size = new_size;
        buf = new_buf;
    }
    uint64_t data = val, mask = ((1ull << off) - 1);
    data <<= off;
    data |= buf[seg] & mask;
    for (size_t i = 0; i < tail; i++) {
        buf[seg + i] = data >> (i << 3);
    }
    bit_size += len;
}


void BinStr::push_back(const BinStr& bs) {
    size_t len = bs.bit_size;
    uint32_t *ptr = (uint32_t*)bs.buf;
    while (len) {
        size_t seglen = len < 32 ? len : 32;
        push_back(*ptr, seglen);
        len -= seglen;
```

```
            ptr++;
        }
    }


    void BinStr::fill() {
        size_t fill = bit_size & 7;
        if (fill) {
            fill = 8 - fill;
            push_back(0, fill);
        }
        push_back(fill, 8);
    }


    void BinStr::unfill() {
        // assert input.get_bit_size() & 7 == 0
        bit_size -= (buf[(bit_size >> 3) - 1] + 8);
    }


    char* BinStr::copy_to_buffer(size_t & cnt_bytes) {
        cnt_bytes = (bit_size + 7) >> 3;
        char* ret = new char[cnt_bytes + 1];
        memcpy(ret, buf, cnt_bytes);
        ret[cnt_bytes] = 0;
        return ret;
    }
```

lzcompress.cpp

```
    #include <cstdint>
    #include <cstring>
    #include <vector>
    #include <fstream>
    #include <iostream>
```

```
#include "binstr.h"
#include "trie.h"
using namespace std;
const uint8_t BITS_IN_UNIT = 1;


struct TagIterHelper {
    size_t len_of_tag, tag_max_num, num_tag;
    TagIterHelper(): len_of_tag(1), tag_max_num(2),
       num_tag(1) {}
    void next() {
        if (num_tag == tag_max_num) {
            len_of_tag++;
            tag_max_num <<= 1;
        }
        num_tag++;
    }
};


void compress(const BinStr& input, BinStr& output) {
    Trie trie(BITS_IN_UNIT);
    size_t pos = 0;
    for (TagIterHelper it; pos < input.get_bit_size();
       it.next()) {
        TrieNode* last_node = trie.find_path(input, pos)
           ;
        output.push_back(last_node->num, it.len_of_tag);
        if (pos < input.get_bit_size()) {
            uint32_t ch = input.substr(pos, BITS_IN_UNIT
               );
            trie.add_node(last_node, ch);
            output.push_back(ch, BITS_IN_UNIT);
            pos += BITS_IN_UNIT;
        }
    }
```

```
    output.fill();
}


void decompress(BinStr& input, BinStr& output) {
    // input will finally be unchanged
    // assert input.get_bit_size() & 7 == 0
    input.unfill();
    vector<BinStr*> list;
    list.push_back(new BinStr());
    size_t pos = 0;
    for (TagIterHelper it; pos < input.get_bit_size();
       it.next()) {
        uint32_t val = input.substr(pos, it.len_of_tag);
        BinStr* new_bs = new BinStr(*list[val]);
        pos += it.len_of_tag;
        if (pos < input.get_bit_size()) {
            uint32_t ch = input.substr(pos, BITS_IN_UNIT
               );
            new_bs->push_back(ch, BITS_IN_UNIT);
            pos += BITS_IN_UNIT;
        }
        output.push_back(*new_bs);
        list.push_back(new_bs);
    }
    for (size_t i = 0; i < list.size(); i++) {
        delete list[i];
    }
    input.fill();
}


int main() {
    ifstream is("samples/news.docx", ifstream::binary);
    size_t buffer_size = 600000, file_size;
    char *buffer = new char[buffer_size];
```

```
    is.read(buffer, buffer_size);
    file_size = is.gcount();
    BinStr source((uint8_t*)buffer, file_size), dest,
       dest2;
    cout << source.get_bit_size() << endl;
    compress(source, dest);
    cout << dest.get_bit_size() << endl;
    decompress(dest, dest2);
    cout << dest2.get_bit_size() << endl;
    delete[] buffer;
    return 0;
}
```

6.2 Huffman 编码

由于 Huffman 编码中使用的 binstr.h 与 binstr.cpp 与 LZ 编码略有不同，因此这里仍然将它们列出。

binstr.h

```
#ifndef _BINSTR_H_
#define _BINSTR_H_
#include <cstdint>

class BinStr {
    size_t arr_size, bit_size;
    uint8_t *buf;
public:
    BinStr();
    BinStr(uint8_t *buf, size_t size);
    BinStr(const BinStr& bs);
    ~BinStr();
    size_t get_bit_size() const;
    uint32_t substr(size_t startpos, size_t len) const;
    void push_back(uint32_t val, size_t len);
    void pop_bit(size_t cnt=1);
```

```
    void push_back(const BinStr& bs);
    void fill();
    void unfill();
    char* copy_to_buffer(size_t & cnt_bytes);
};

#endif
```

binstr.cpp

```
#include <cstring>
#include "binstr.h"

//create binstr
BinStr::BinStr() {
    arr_size = 10;
    bit_size = 0;
    buf = new uint8_t[arr_size];
}

//add a byte
BinStr::BinStr(uint8_t *src, size_t src_bytes) {
    arr_size = src_bytes + 10;
    bit_size = (src_bytes << 3);
    buf = new uint8_t[arr_size];
    memcpy(buf, src, src_bytes);
}

//copy the Binstr
BinStr::BinStr(const BinStr& bs) {
    arr_size = bs.arr_size;
    bit_size = bs.bit_size;
    buf = new uint8_t[arr_size];
    memcpy(buf, bs.buf, ((bit_size + 7) >> 3));
}
```

```
//delete the BinStr
BinStr::~BinStr() {
    delete[] buf;
}

//get the length of the string
size_t BinStr::get_bit_size() const {
    return bit_size;
}

//get a bit from the string
uint32_t BinStr::substr(size_t startpos, size_t len)
   const {
    size_t seg = startpos >> 3, off = startpos & 7, tail
        = (off + len + 7) >> 3;
    uint64_t data = 0, mask = ((1ull << len) - 1);
    for (size_t i = 0; i < tail; i++) {
        data |= buf[seg + i] << (i << 3);
    }
    data >>= off;
    return data & mask;
}

//append a byte
void BinStr::push_back(uint32_t val, size_t len) {
    size_t seg = bit_size >> 3, off = bit_size & 7, tail
        = (off + len + 7) >> 3;
    if (seg + tail > arr_size) {
        size_t new_size = (seg + tail) << 1;
        uint8_t *new_buf = new uint8_t[new_size];
        memcpy(new_buf, buf, arr_size);
        delete[] buf;
        arr_size = new_size;
        buf = new_buf;
    }
    uint64_t data = val, mask = ((1ull << off) - 1);
```

```
    data <<= off;
    data |= buf[seg] & mask;
    for (size_t i = 0; i < tail; i++) {
        buf[seg + i] = data >> (i << 3);
    }
    bit_size += len;
}

//append a Binstr
void BinStr::push_back(const BinStr& bs) {
    size_t len = bs.bit_size;
    uint32_t *ptr = (uint32_t*)bs.buf;
    while (len) {
        size_t seglen = len < 32 ? len : 32;
        push_back(*ptr, seglen);
        len -= seglen;
        ptr++;
    }
}

//reduce bit_size
void BinStr::pop_bit(size_t cnt) {
    bit_size -= cnt;
}

//add zero-padding
void BinStr::fill() {
    size_t fill = bit_size & 7;
    if (fill) {
        fill = 8 - fill;
        push_back(0, fill);
    }
    push_back(fill, 8);
}

//delete zero-padding
```

```
void BinStr::unfill() {
    // assert input.get_bit_size() & 7 == 0
    bit_size -= (buf[(bit_size >> 3) - 1] + 8);
}

//change the binstr to char
char* BinStr::copy_to_buffer(size_t & cnt_bytes) {
    cnt_bytes = (bit_size + 7) >> 3;
    char* ret = new char[cnt_bytes + 1];
    memcpy(ret, buf, cnt_bytes);
    ret[cnt_bytes] = 0;
    return ret;
}
```

main.cpp

```
#include <queue>
#include <vector>
#include <cstring>
#include <iostream>
#include <fstream>
#include "binstr.h"
using namespace std;


//node of the Huffman tree
struct TreeNode
{
    unsigned short num, children[2];
    unsigned freq;
};


//compare the weight of two node
bool operator<(const TreeNode &lhs, const TreeNode &rhs)
{
```

```
    return lhs.freq > rhs.freq || (lhs.freq == rhs.freq
       && lhs.num < rhs.num);
}


//add a new tree node to the Huffman tree
TreeNode add_leaf_node(std::vector<TreeNode> &the_tree,
   unsigned freq)
{
    TreeNode ret;
    ret.freq = freq;
    ret.num = the_tree.size();
    the_tree.push_back(ret);
    return ret;
}

//merge two node and add their frequency
TreeNode add_merged_nodes(std::vector<TreeNode> &
   the_tree,
                          const TreeNode &lhs, const
                             TreeNode &rhs)
{
    TreeNode ret;
    ret.freq = lhs.freq + rhs.freq;
    ret.children[0] = lhs.num;
    ret.children[1] = rhs.num;
    ret.num = the_tree.size();
    the_tree.push_back(ret);
    return ret;
}

//create convert table
void recur(size_t cur, const std::vector<TreeNode> &
   the_tree, BinStr *convert_table[], BinStr &tmp)
{
    if (cur < 256)
```

```
    {
        convert_table[cur] = new BinStr(tmp);
    }
    else
    {
        tmp.push_back(0, 1);
        recur(the_tree[cur].children[0], the_tree,
           convert_table, tmp);
        tmp.pop_bit();
        tmp.push_back(1, 1);
        recur(the_tree[cur].children[1], the_tree,
           convert_table, tmp);
        tmp.pop_bit();
    }
}

//compress the file
char *compress(const char *input, size_t insize, size_t
   &outsize)
{
    // build tree
    unsigned freq[256];
    memset(freq, 0, sizeof(unsigned) * 256);

    for (size_t i = 0; i < insize; i++)
    {
        freq[(unsigned char)input[i]]++;
    }
    std::vector<TreeNode> the_tree;
    std::priority_queue<TreeNode> the_heap;
    for (size_t i = 0; i < 256; i++)
    {
        TreeNode tmp = add_leaf_node(the_tree, freq[i]);
        if (freq[i])
        {
            the_heap.push(tmp);
```

```
    }
}
while (the_heap.size() > 1)
{
    TreeNode a = the_heap.top();
    the_heap.pop();
    TreeNode b = the_heap.top();
    the_heap.pop();
    the_heap.push(add_merged_nodes(the_tree, a, b));
}
// build convert table
BinStr *convert_table[256];
memset(convert_table, 0, sizeof(BinStr *) * 256);
BinStr tmp;
recur(the_tree.size() - 1, the_tree, convert_table,
   tmp);
// start dumping
BinStr output;
//add the tree node
output.push_back(the_tree.size() - 256, 8);
for (size_t i = 256; i < the_tree.size(); i++)
{
    output.push_back(the_tree[i].children[0], 9);
    output.push_back(the_tree[i].children[1], 9);
}
//add the real content
for (size_t i = 0; i < insize; i++)
{
    output.push_back(*convert_table[(unsigned char)
       input[i]]);
}
for (size_t i = 0; i < 256; i++)
{
    delete convert_table[i];
}
//use zero-padding to fill the whole byte
```

```
    output.fill();
    return output.copy_to_buffer(outsize);
}


char *decompress(const char *input, size_t inlen, size_t
    &outlen)
{

    //tmp for real content
    BinStr compressed((uint8_t *)input, inlen);
    //delete the padding
    compressed.unfill();
    //rebuild the tree
    size_t cnt_nonleaf = compressed.substr(0, 8);
    std::vector<TreeNode> the_tree;
    size_t index = 8;
    for (size_t i = 0; i < cnt_nonleaf; i++)
    {
        TreeNode tmp;
        tmp.children[0] = compressed.substr(index, 9);
        index += 9;
        tmp.children[1] = compressed.substr(index, 9);
        index += 9;
        the_tree.push_back(tmp);
    }
    //start decoding
    BinStr decompressed;
    size_t cur, root_node;
    cur = root_node = cnt_nonleaf + 255;
    while (index < compressed.get_bit_size())
    {
        int bit = compressed.substr(index, 1);
        cur = the_tree[cur - 256].children[bit];
        if (cur < 256)
        {
```

```
            decompressed.push_back(cur, 8);
            cur = root_node;
        }
        index++;
    }
    //turn the binstr into char
    return decompressed.copy_to_buffer(outlen);
}

int main()
{
    //read the file
    std::ifstream is("samples/news.docx", std::ios::
       binary);
    char *initial = new char[600000];
    is.read(initial, 600000);
    size_t inlen = is.gcount(), outlen;
    //compress
    char *buf = compress(initial, inlen, outlen);
    //the length of the original file and the compressed
        file
    std::cout << inlen << std::endl;
    std::cout << outlen << std::endl;
    //decompress the file
    char *debuf = decompress(buf, outlen, inlen);
    std::ofstream decode("samples/decode.docx", std::ios
       ::binary);
    decode.write(debuf, inlen);
    //delete the dynamic array
    delete[] debuf;
    delete[] buf;
    delete[] initial;
    return 0;
}
```

七、附录

以 2 bit 为单位对 txt 文件进行 LZ 编解码的结果：

图 **14** **LZ** 编解码（**2bit**）压缩 **txt** 文件结果

以 2 bit 为单位对 docx 文件进行 LZ 编解码的结果：

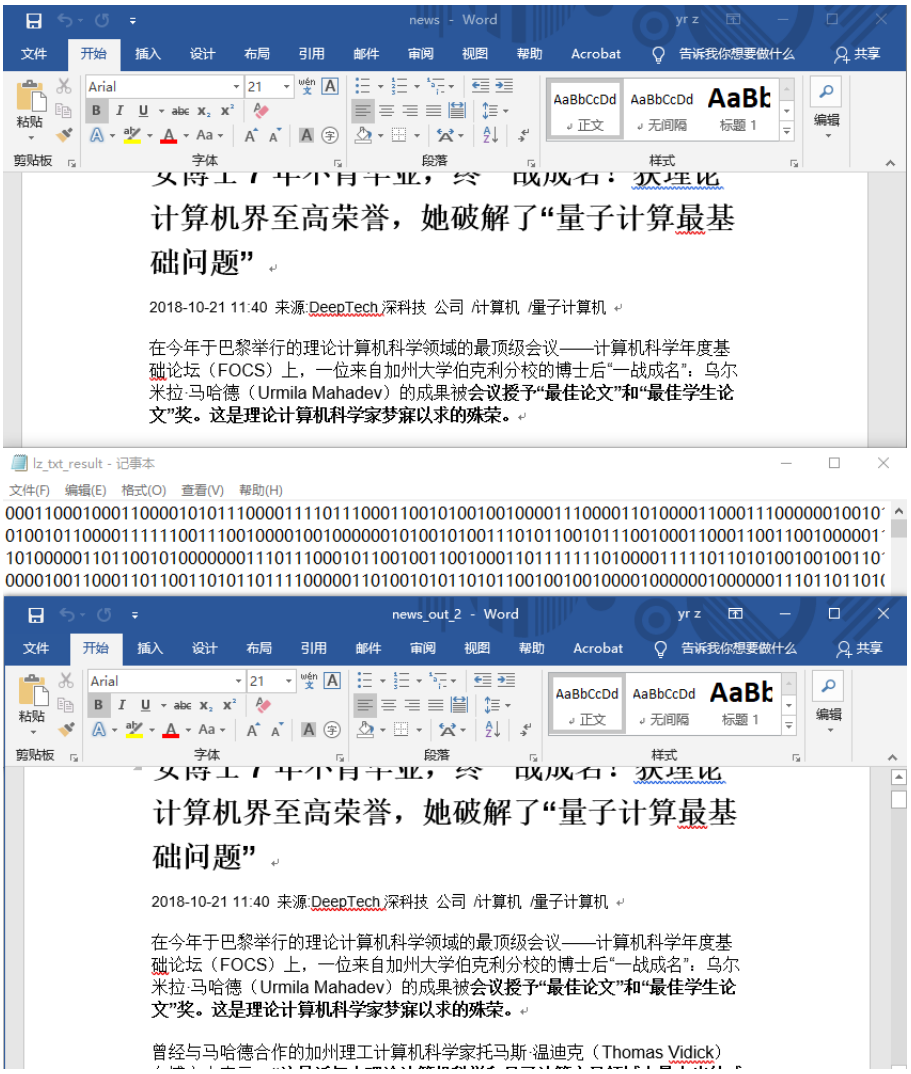


图 **15**　**LZ** 编解码（**2bit**）压缩 **txt** 文件结果

以 4 bit 为单位对 txt 文件进行 LZ 编解码的结果：

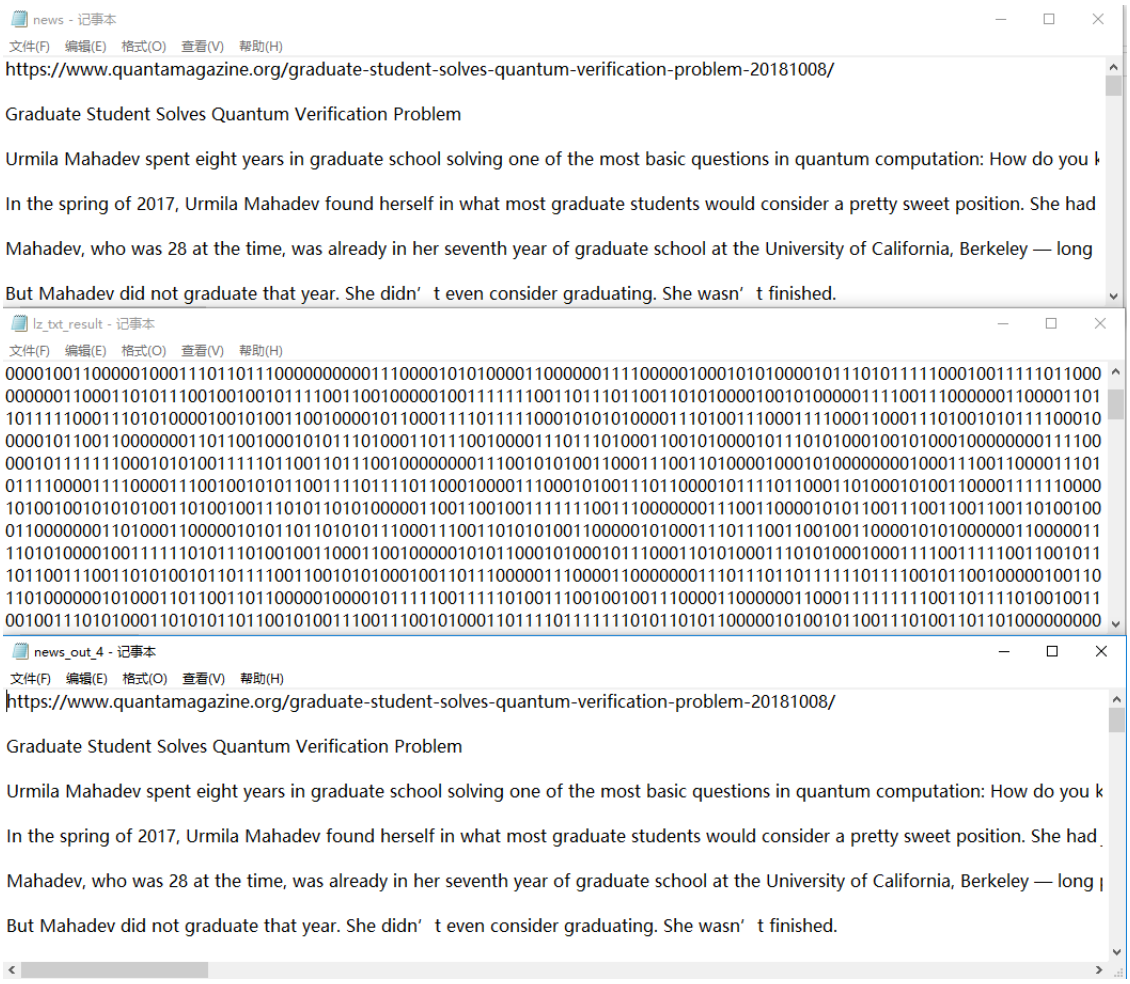


图 **16** **LZ** 编解码（**4bit**）压缩 **txt** 文件结果

以 4 bit 为单位对 docx 文件进行 LZ 编解码的结果：

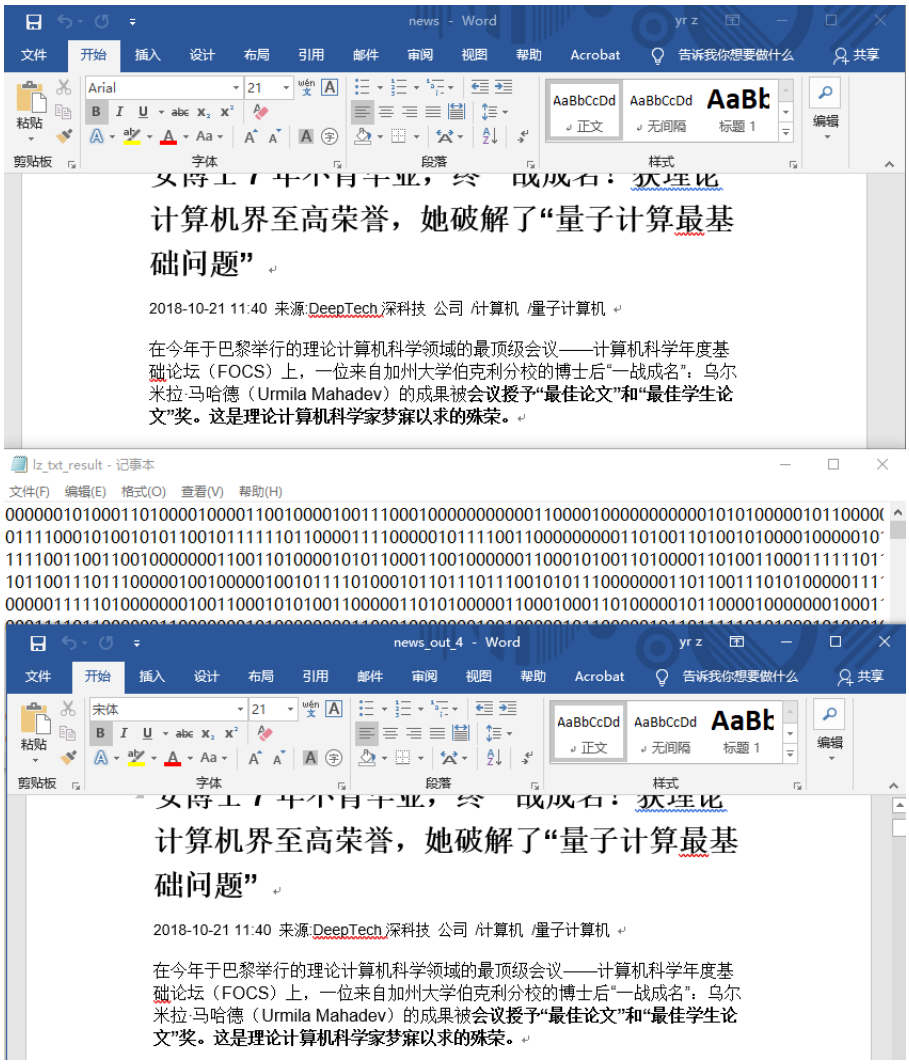


图 **17**　**LZ** 编解码（**4bit**）压缩 **txt** 文件结果

以 8 bit 为单位对 txt 文件进行 LZ 编解码的结果：

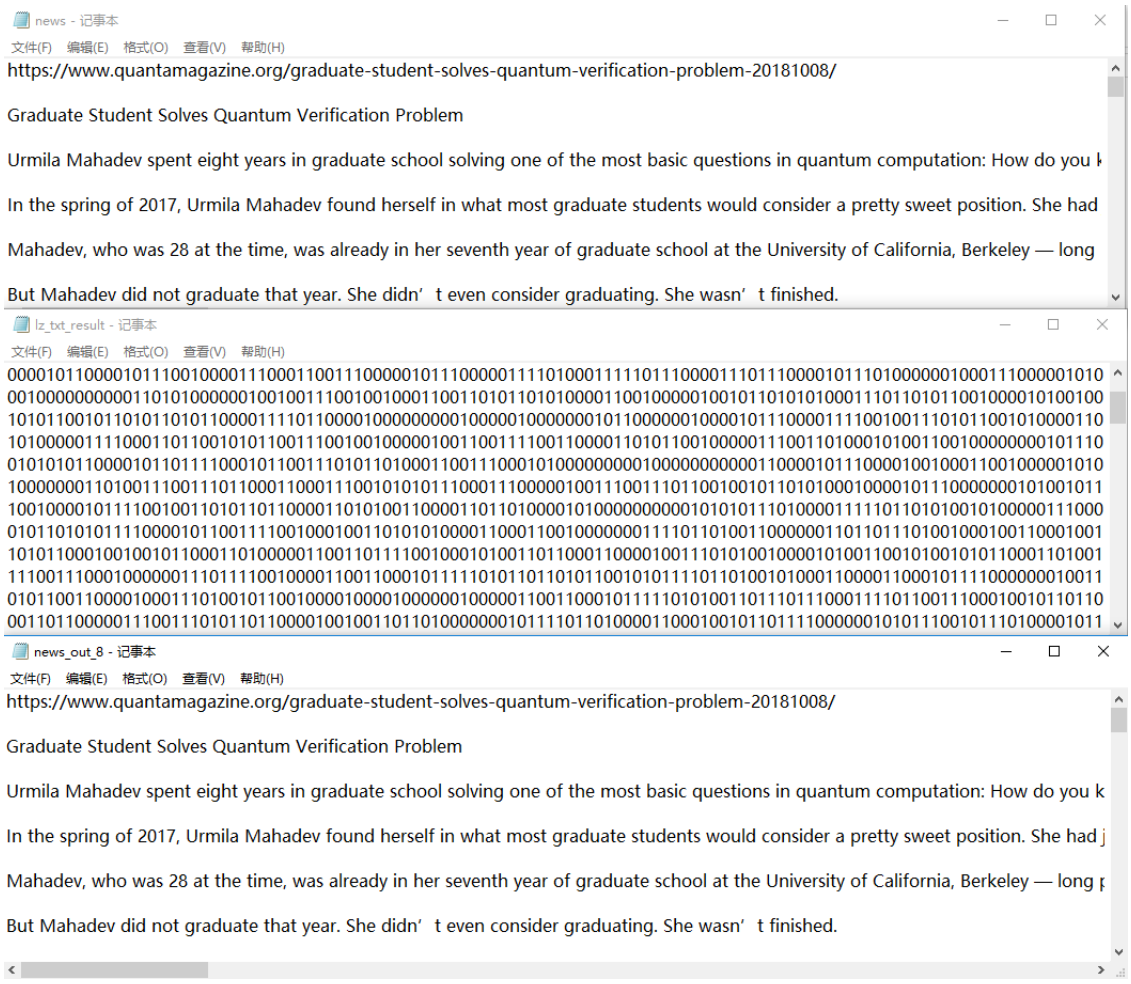


图 **18**　**LZ** 编解码（**8bit**）压缩 **txt** 文件结果

以 8 bit 为单位对 docx 文件进行 LZ 编解码的结果：

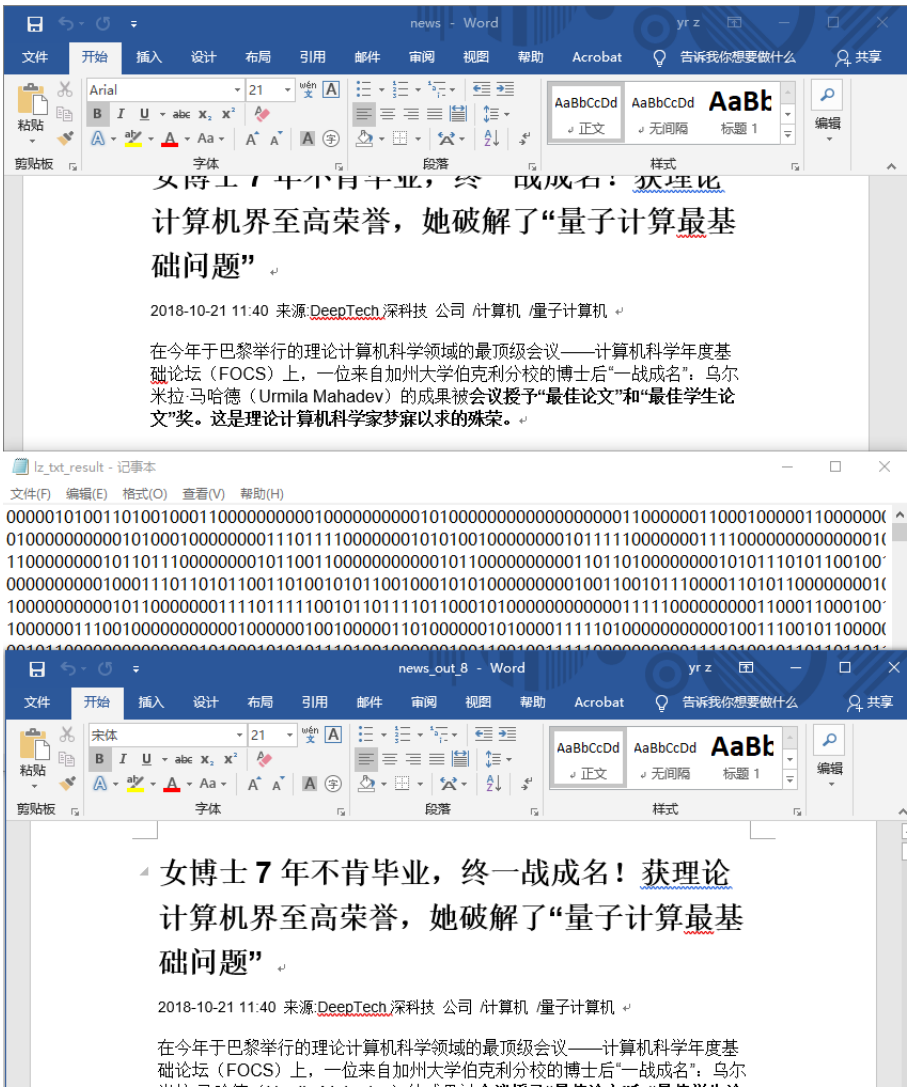


图 **19**　**LZ** 编解码（**8bit**）压缩 **txt** 文件结果