密码学第五次实验报告

# AES加解密算法

## 原理

AES是DES之后的一种加密标准，比DES更加安全，能有效地抵抗所有已知攻击。AES采用了非标准的Feistel密码结构，但是同样有足够的代替和置换，来增加密码分析的难度。AES工作的最小单位是字节，而不是DES中的比特。AES支持128位的明文分组长度，以及128、192和256位的密钥分组长度。但是这次实验可以只实现对128位的密钥分组长度的支持。

对密钥生成算法来说，首先种子密钥被按字节以先列后行的顺序分组，填充到密钥状态里。把密钥状态的一列视为一个整体，每轮通过一次非线性变换和多次线性变换，来生成四个新的密钥状态的列，从而得到下一轮的轮密钥。线性变换与已经生成的最近一轮的轮密钥以及正在生成的轮的轮密钥的一部分都有关，非线性变换提供了足够的非线性。

对加密算法来说，首先明文以与填充种子密钥相似的方式被填充进加密状态中。然后做一次AddRoundKey操作。该操作和以下的所有操作一样，都是在现有的状态上就地进行的。然后持续9轮，每轮按顺序进行SubByte、ShiftRow、MixColumn、AddRoundKey操作。最后一轮去掉MixColumn操作，只有按顺序进行的SubByte、ShiftRow、AddRoundKey操作。SubByte和AddRoundKey主要提供了混淆，ShiftRow和MixColumn主要提供了扩散。混淆和扩散交替进行，使得AES能有足够的密码强度来抵抗已知的攻击。

SubByte是把状态中的每一位按已经公开生成算法的S盒进行代替的。ShiftRow是按照明文和密钥的长度，对状态的每一行进行按字节的循环移位。MixColumn是在一列中做线性变换，使得一列中的加密结果可以扩散。AddRoundKey是加密状态和密钥按字节异或。

对解密算法来说，由于它是加密算法的逆操作，所以要知道所有加密的部件的逆。各部件的逆包括InvSubByte、InvShiftRow、InvMixColumn、AddRoundKey操作。（易知AddRoundKey操作是自逆的。）它们都容易构造。于是，只要弄明白加密时以什么顺序运用了各个子部件，就不难按顺序构造出解密操作。实际上，解密操作也是可以看成有10轮的，其中前9轮相同，最后一轮略有不同，少了一个步骤。

实际上，解密算法中InvSubByte和InvShiftRow可以互换，InvMixColumn和AddRoundKey在改变要加的轮密钥的情况下可以互换，而且互换前后对状态的改变是等价的。由此可以构造出与加密算法结构相同的解密算法。

## 伪代码

### ByteSub部件

**def** bytesub(state) -> None:

把state中的每一个字节，都更改为查ByteSub部件对应的代替表后得到的字节

### InvByteSub部件

**def** invbytesub(state) -> None:

把state中的每一个字节，都更改为查InvByteSub部件对应的代替表后得到的字节

### ShiftRow部件

**def** shiftrow(state) -> None:

查ShiftRow部件对应的与每行循环右移列数的关系表，得到每行循环右移列数

对state中的每一行:

循环右移相应列数

### InvShiftRow部件

**def** shiftrow(state) -> None:

查InvShiftRow部件对应的与每行循环右移列数的关系表，得到每行循环右移列数

对state中的每一行:

循环右移相应列数

### MixColumn部件

**def** mixcolumn(state) -> None:

对state中的每一列:

在中左乘MixColumn部件对应的变换矩阵

### InvMixColumn部件

**def** invmixcolumn(state) -> None:

对state中的每一列:

在中左乘InvMixColumn部件对应的变换矩阵

### AddRoundKey部件

**def** addroundkey(state, round\_key: List[bytearray]) -> None:

对state中的每一列:

与round\_key中的相应列进行按字节异或

### 函数

**def** \_g(k: bytearray, round\_: int) -> bytearray:

把k循环左移一位

把k中的每一个字节，都更改为查ByteSub部件对应的代替表后得到的字节

把k与round\_对应的轮常数异或

**return** k

### 生成密钥列表算法

**def** get\_key\_list(key\_state) -> List[List[bytearray]]:

key\_list = []

key\_list追加key\_state对应的初始密钥

old\_result = key\_state对应的初始密钥

result = 空密钥

重复10次:

result的第0个双字old\_result的第3个双字

old\_result的第0个双字

result的第个双字result的第个双字old\_result的第个双字

old\_result = result

result = 空密钥

key\_list追加old\_result

**return** key\_list

### 加密算法

**def** encrypt(p: bytes, k: bytes) -> bytes:

由p生成state

由k生成key\_state

key\_list = get\_key\_list(key\_state)

addroundkey(state, key\_list[0])

for i in range(9):

bytesub(state)

shiftrow(state)

mixcolumn(state)

addroundkey(state, key\_list[i + 1])

bytesub(state)

shiftrow(state)

addroundkey(state, key\_list[10])

**return** state转换成字节后的结果

### 解密算法

**def** decrypt(p: bytes, k: bytes) -> bytes:

由p生成state

由k生成key\_state

key\_list = get\_key\_list(key\_state)

addroundkey(state, key\_list[10])

invshiftrow(state)

invbytesub(state)

for i in range(8, 0, -1):

addroundkey(state, key\_list[i + 1])

invmixcolumn(state)

invshiftrow(state)

invbytesub(state)

addroundkey(state, key\_list[0])

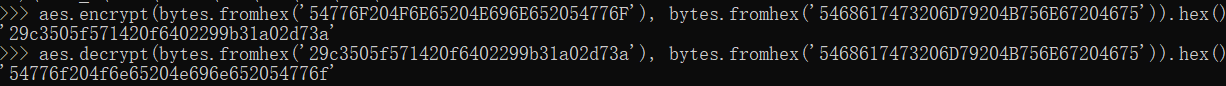
**return** state转换成字节后的结果

## 分析

由于明密文长度给定，各部件以及密钥生成算法的各步骤也能有常数级别的的时空复杂度，因此整个算法的时空复杂度为。。

## 测试

采用明文54776F204F6E65204E696E652054776F，密钥5468617473206D79204B756E67204675进行测试。得到密文为29C3505F571420F6402299B31A02D73A。这里明文、密文、密钥均采用16进制表示。采用相同的密钥解密密文，得到原来的明文。



经过验算，结果正确。

## 优化

### ByteSub部件和InvByteSub部件

事实上，ByteSub部件的并行潜力非常高，因为各字节之间没有数据以来。所以其实可以用查表的方式，来让ByteSub部件并行执行。

用硬件实现ByteSub部件的时候，其实可以用纯组合逻辑的实现，不用查表。但是纯组合逻辑实现可能会受到侧信道攻击，因此查表理论上更安全。

因为InvByteSub部件的计算操作和ByteSub部件类似，所以它的的情况和ByteSub部件的情况类似。

### ShiftRow部件和InvShiftRow部件

ShiftRow部件其实难以并行，因为对加密状态同一列上的各字节，要移动的位数可能都是不一样的。因此并行的力度要精细到多个字节，开销有些大。在非并行的时候，ShiftRow用查表来实现是一个比较好的选择，这里查表指的是查InvShiftRow部件对应的与每行循环右移列数的关系表，得到每行循环右移列数，并对每行循环右移相应的列数。

类似地，InvShiftRow部件的情况和ShiftRow部件的情况类似。

### MixColumn部件和InvMixColumn部件

MixColumn部件和ShiftRow部件的情况类似，并行力度比较精细，因此开销也有些大，也难以并行。一般MixColumn都是对一列进行上的矩阵乘法，因此把上的乘法表提前预置，在运算时查表也是一个好的选择，因为这样能避免依次判断每一位可能造成的侧信道攻击。类似地，InvMixColumn部件的情况和MixColumn部件的情况类似。

实际上MixColumn部件还有一种优化的实现方法，操作的最小单位是字节。

令

然后用以下公式计算：

可以证明，这样生成的是正确的。这样只需要用到多次异或操作和4次上的乘法操作，简化了操作步骤。这里用到了性质。

### AddRoundKey部件

该部件很好并行化，也很好硬件实现，因为实际上该部件就是批量化的异或操作。其实在现代处理器中，也可以利用并行化指令实现。

### 函数

该函数其实不好也没有必要并行化，因为数据规模太小、数据依赖程度太高了。轮函数也应该用表的方式提前存储起来。

### 生成密钥列表算法

该算法其实是用当前密钥状态来计算下一个密钥状态，因此不必多次保存所有的密钥状态，之前的密钥状态只保存到结果列表中即可。这样能降低算法的空间需求。

### 加密算法

事实上，在整数位长不小于32位的处理器上，有优化的加密实现方法。对任意一列，有：

其中为字节经过ByteSub部件后的结果，列下标需要模4。

根据每一个完整轮的加密方式，易知上式成立。

因此可以定义4个1024字节的表：

这些表中的取值范围为一个字节的所有可能取值。易知各表的大小确实是1024字节，而且能被事先计算。可以把这些表看成一种函数，这种函数输入一个字节值，产生一个32位的列向量，该向量是对应输入字节的函数。

这样轮函数对一列的操作就能简化成查表和异或：

显然这种方案能大大减小加密操作需要的时间。但是需要的空间略有增加，因此这种方案是一种以空间换时间的方案。

### 解密算法

通过加密算法的对应步骤可以推出，解密算法也可以分轮。前面的9轮依次执行InvShiftRow、InvByteSub、InvMixColumn、AddRoundKey，最后一轮依次执行InvShiftRow、InvByteSub、AddRoundKey操作。

事实上，由于InvShiftRow和InvByteSub操作一个只变换字节的顺序，不变换字节的内容，而且变换不依赖于字节的内容，另一个恰好反过来，因此二者可以互换。易知InvMixColumn相当于对一列左乘一个固定的矩阵，因此是线性运算。因此。这样，可以得出在既有InvMixColumn又有AddRoundKey的轮中，如果想把两个操作互换，只需要在互换后的AddRoundKey操作中，把原来的相应轮密钥换成经InvMixColumn部件变换后的相应轮密钥即可。因此InvMixColumn和AddRoundKey在有条件的情况下可以互换。

这样，就可以构造与加密算法结构大致相同的解密算法。这样可以使得加密算法和解密算法部分共用一个软件或硬件模块，提高了效率。

# AES加解密文件算法

## 原理

AES属于分组密码，因此加密一次，数据长度应该是固定的。如果想把分组密码改造成能够加密任意长度数据的密码，有多种改造形式，如ECB、CBC、CTB、OFB和CTR模式等。其中CBC模式是把明文按顺序分成大小为分组大小的块，但是每块在加密前与上一块的加密结果异或。对第一块，没有所谓上一块的加密结果，于是用初始向量与其异或。实际上，初始向量应该是收发加密信息的双方共同约定好的。解密时，类似地把密文按顺序分成大小为分组大小的块。把解密出来的结果与上一段密文进行异或，就得到明文。对于第一块密文，同样用初始向量与其解密结果异或。

加密算法的公式为：

解密算法的公式为：

可以推导出，加密算法和解密算法互为逆运算。

## 伪代码

### 文件加密算法

**def** encrypt\_file(p\_path: str, c\_path: str, k: bytes) -> None:

分别打开p\_path和c\_path对应的文件作为明文文件和密文文件

last\_ciphertext\_bytes = 初始向量

**while** True:

从p\_path中读16个字节，放入curr\_bytes中

若文件已经读完:

**break**

若curr\_bytes长度小于16个字节:

用0字节在curr\_bytes后面填充，使其长度为16个字节

curr\_bytes = curr\_bytes ^ last\_ciphertext\_bytes

ciphertext\_bytes = encrypt(curr\_bytes, k)

往密文文件中写入ciphertext\_bytes

last\_ciphertext\_bytes = ciphertext\_bytes

关闭明文文件和密文文件

### 文件解密算法

**def** decrypt\_file(p\_path: str, c\_path: str, k: bytes) -> None:

分别打开p\_path和c\_path对应的文件作为明文文件和密文文件

last\_ciphertext\_bytes = 初始向量

**while** True:

从c\_path中读16个字节，放入curr\_bytes中

若文件已经读完:

**break**

若curr\_bytes长度小于16个字节:

用0字节在curr\_bytes后面填充，使其长度为16个字节

plaintext\_bytes = decrypt(curr\_bytes, k)

plaintext\_bytes = plaintext\_bytes ^ last\_ciphertext\_bytes 往明文文件中写入plaintext\_bytes

last\_plaintext\_bytes = curr\_bytes

关闭明文文件和密文文件

## 分析

设文件长度为。

文件可能在加密前会被填充，但是长度总是不超过一个块的大小，因此增加的长度也是常数数量级。对待加密或解密的文件分块以后，易知块数与成正比。而每次加解密的时间复杂度为，因此算法时间复杂度为。算法只需要保存对当前正在加解密的块进行加解密时必要的信息和上一个块的相应信息，因此算法空间复杂度为。

## 测试

由于下一节中AES加解密文件程序封装了该算法，对该算法的测试在下一节进行。

## 优化

事实上CFB模式能够保证相同的明文不输出相同的密文，但是有一大缺点，就是前后数据高度依赖，并行性比较弱。但是CFB模式能够提供比ECB模式更强的安全。因此，从我有限的认知来看，该算法似乎性能优化空间并不大。

在易用性上，可以考虑创建一种自定义的文件格式，从而能够记住文件的长度，这样就能够实现任意长度文件的加解密。当然也可以用密文窃取的方式来达到对任意长度文件加解密的效果。

# AES加解密文件程序

## 原理

Python中自带了tkinter库，可以比较方便地构建图形界面程序。因此，可以通过tkinter库，实现一个简单的利用AES加解密文件的程序。程序中应该封装上一节中提到的AES加解密文件算法。

由于要实现的是一个比较完整的程序，所以应该具有一定的异常判断与处理功能。这里用Python内置的异常处理机制来处理异常。

## 伪代码

### 文件加密功能

**def** button\_encrypt(self) -> None:

弹出询问明文文件的文件选择对话框，并把明文文件路径放入open\_filename中

若open\_filename为空:

提示用户已取消操作

**return**

弹出询问密文文件的文件选择对话框，并把明文文件路径放入save\_filename中

若save\_filename为空:

提示用户已取消操作

**return**

提示用户输入k，格式为32位16进制数

**try**:

把k转换成合法的二进制数据类型

检查k的长度

**except** ValueError:

提示用户k格式错误

**return**

**try:**

encrypt\_file(open\_filename, save\_filename, k)

**except** OSError **as** e:

提示用户e中的错误信息

**return**

提示用户加密成功

### 文件解密功能

**def** button\_decrypt(self) -> None:

弹出询问密文文件的文件选择对话框，并把明文文件路径放入open\_filename中

若open\_filename为空:

提示用户已取消操作

**return**

弹出询问明文文件的文件选择对话框，并把明文文件路径放入save\_filename中

若save\_filename为空:

提示用户已取消操作

**return**

提示用户输入k，格式为32位16进制数

**try**:

把k转换成合法的二进制数据类型

检查k的长度

**except** ValueError:

提示用户k格式错误

**return**

**try:**

decrypt\_file(open\_filename, save\_filename, k)

**except** OSError **as** e:

提示用户e中的错误信息

**return**

提示用户解密成功

### 界面初始化

**def** main() -> None:

申请控件系统资源

创建窗口实例

进入事件循环

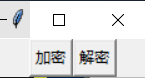
清理并退出

## 分析

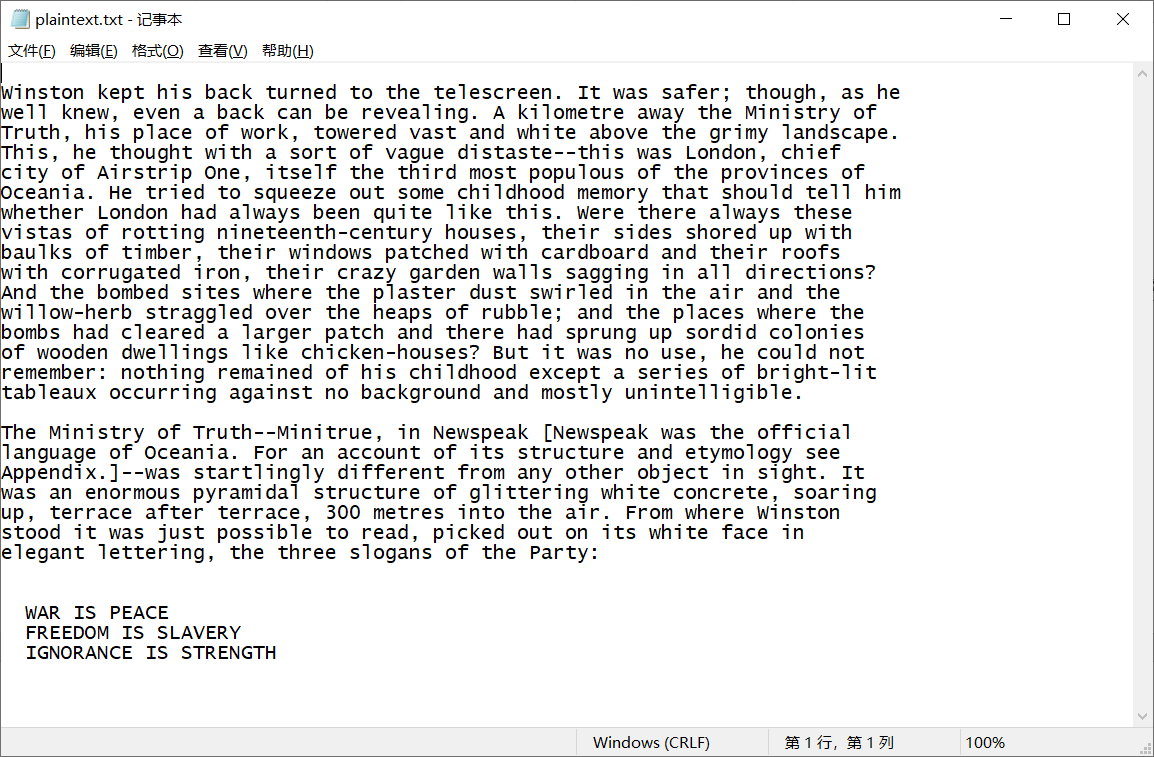
实际上加了用户界面后，由于用户界面基本上只是提供输入输出功能，并不涉及实际上的运算，因此用户界面的时空复杂度为。

## 测试

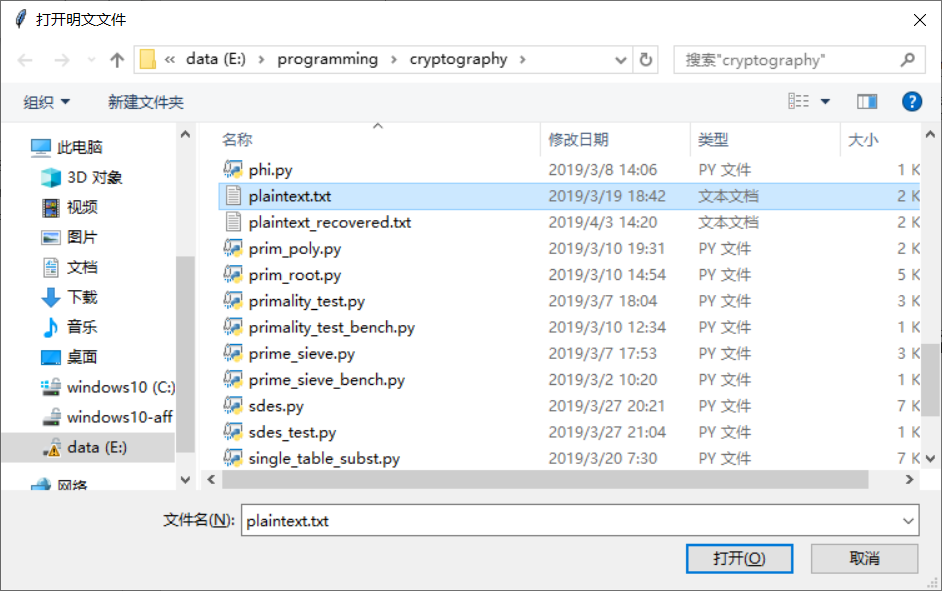
程序用户界面如下：

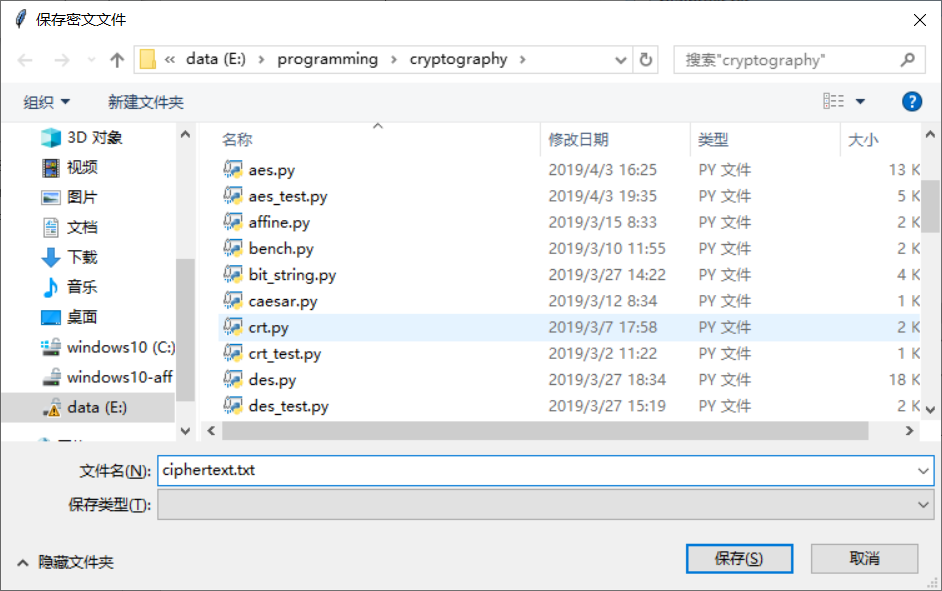


初始向量在代码中预设为0123456789ABCDEF，密钥为ABCDABCDABCDABCDABCDABCDABCDABCD。明文文件如下：

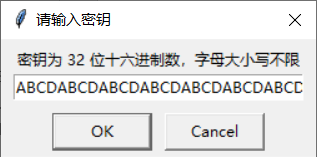


单击“加密”按钮，选择明文文件和密文文件路径：

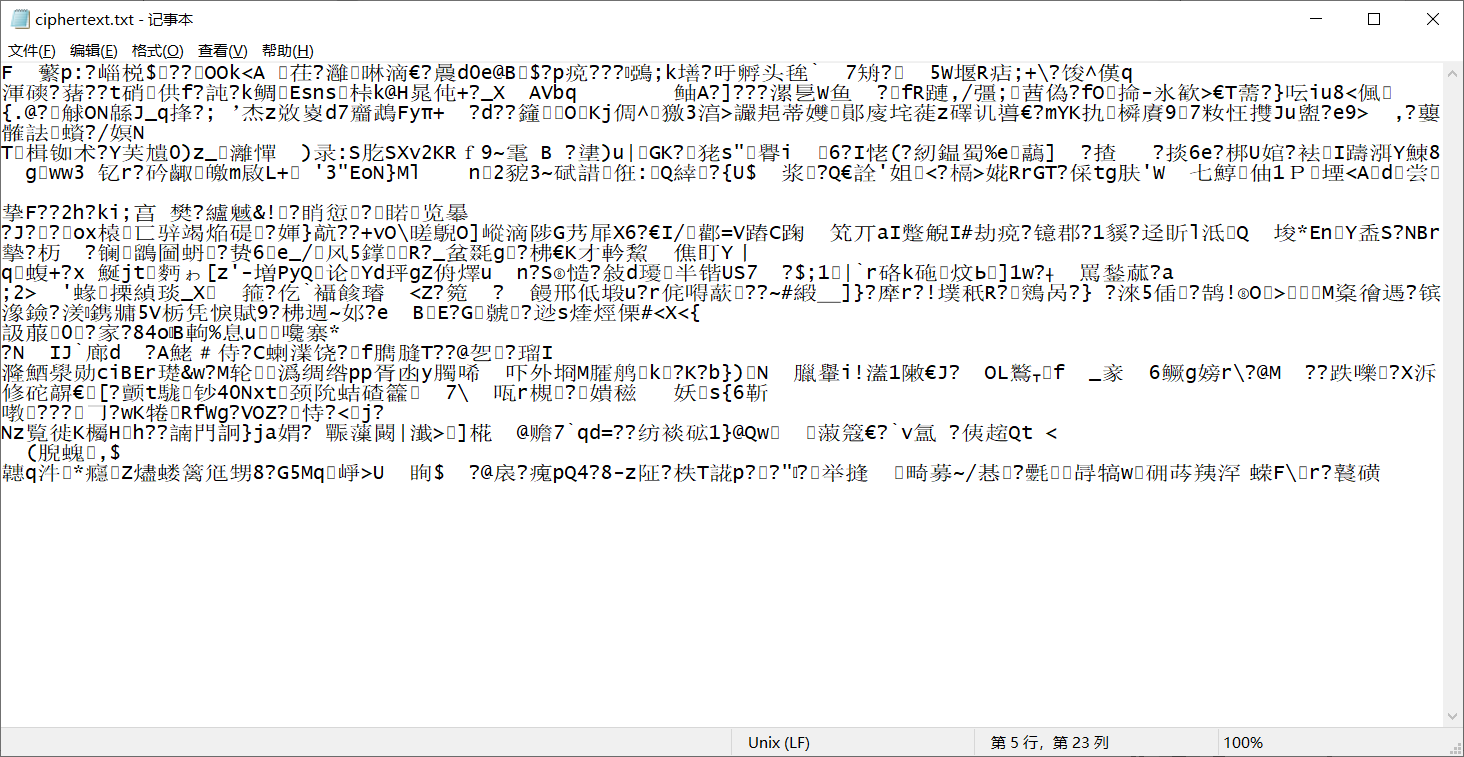




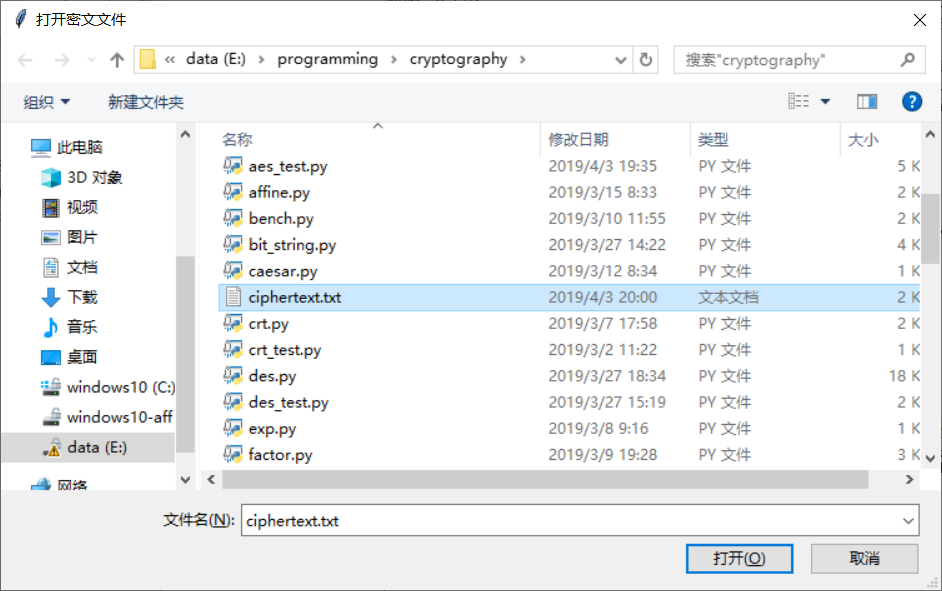
输入密钥：

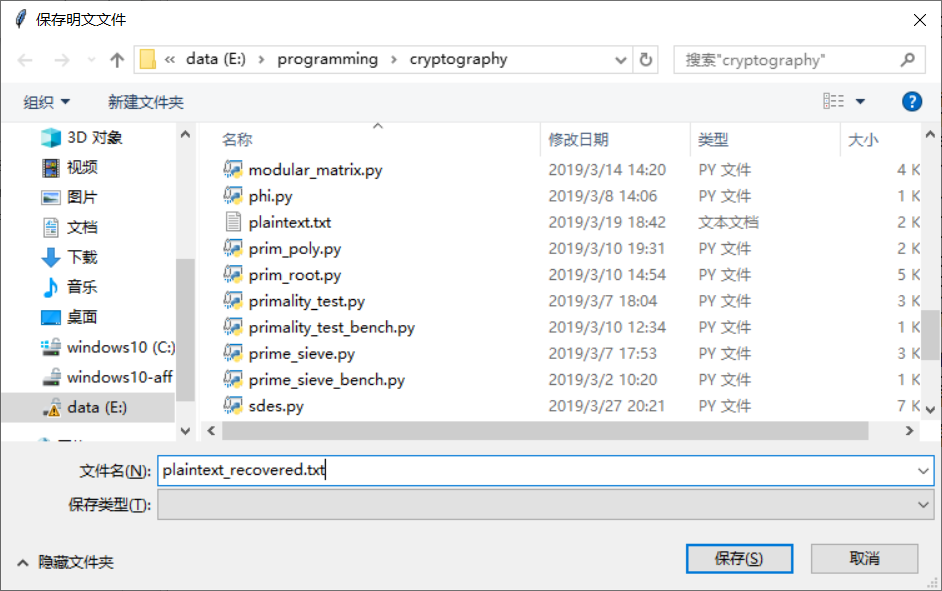


加密后的密文文件如下：

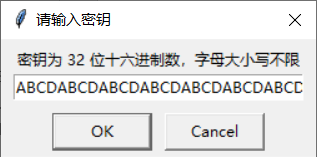


单击“解密”按钮，选择密文文件路径和恢复出的明文文件要存放的路径：

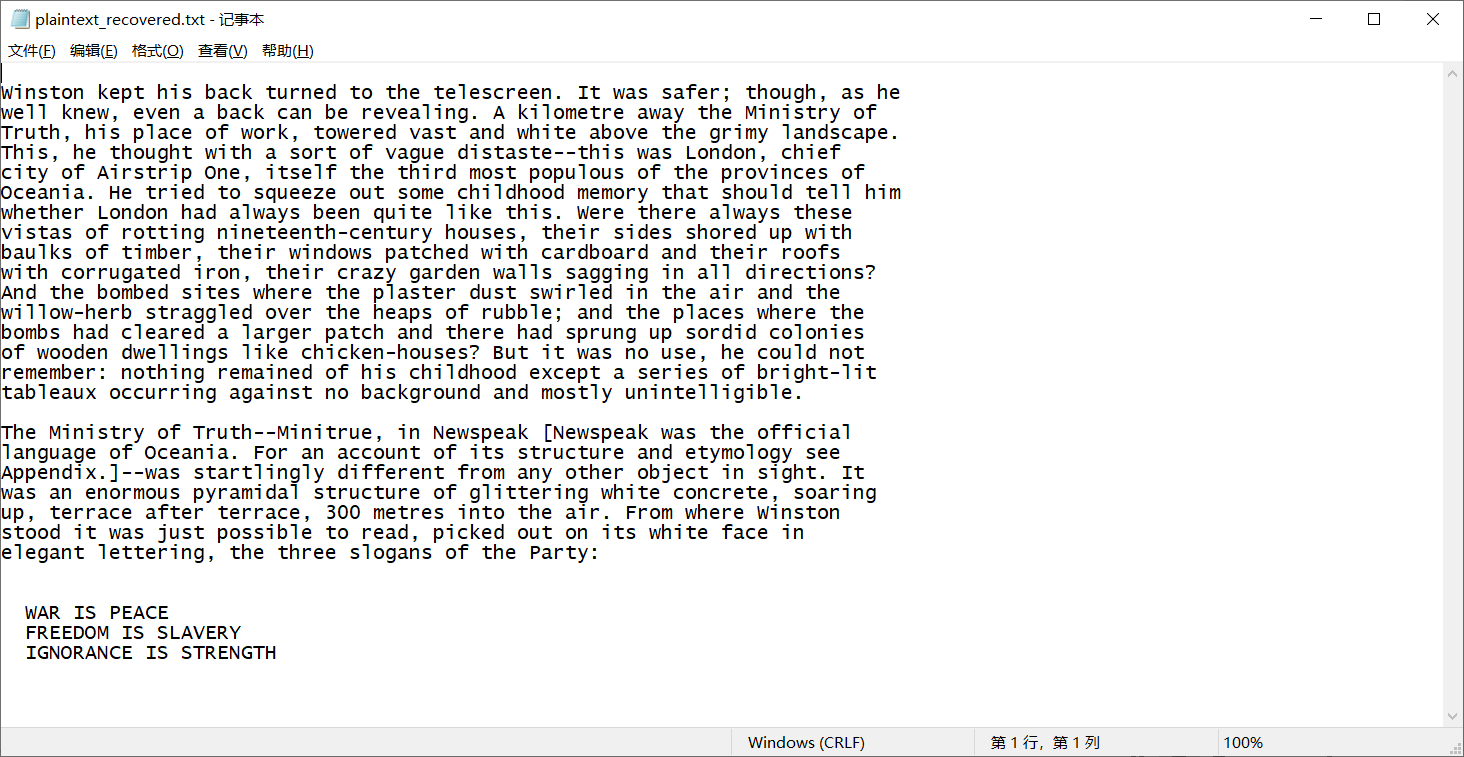




输入相同密钥：



恢复出来的明文文件如下：



可以发现，明文文件和恢复出来的密文文件相同。

## 优化

实际上该程序是封装了AES加解密文件算法，对该核心算法的优化，上节已经讲过了。对程序本身，可以考虑对易用性进行优化、对程序的用户界面美观性进行优化，等等。

# 总结

## AES加解密算法

这是我写的第二个正规的加解密算法。其实AES比DES更好写，因为它用到的是字节，比以位为基本单位的DES更加方便处理。AES加解密算法其实有点对对象操作的意味，因此可以用面向对象的方式来写，这里也可以用到Python中的面向对象特性。事实上，我有点遗憾的地方就是没有写支持192和256位密钥长度的AES加密算法，虽然它应该不在基础要求里。

## AES加解密文件算法

事实上用分组密码加密任意长度的文件（或者说字节串）有多种方法。CBC属于难度一般但是用途较大的一种。其余的方法其实也可以尝试。对CBC本身来说，可以使用密文窃取的方式，使得任意字节串加密后与原字节串等长。

## AES加解密文件程序

这是我第一次在密码学实验中写一个GUI程序。其实写GUI程序，查文档了解API虽然没有算法那么硬核，但是还是挺有意思的，比较考验人的学习能力。因此，我查了三四个小时的文档，终于写出了GUI程序。其实这个程序也算另一种类型的挑战。

## 算法评估与优化

这次算法评估相对简单，并没有太难的地方。但是实际上算法优化也不能让放松，尤其是像加密算法这种常用的算法。

## 系统设计与维护

这次好的系统设计真的能加速程序的实现过程。在分析问题的时候，我就想到AES的加密步骤与面向对象的思想是契合的，于是我采用了面向对象的思想来写AES加密算法，发现写得速度还是比较快的。通过对对象的字符串表示进行重载，我可以随时打印出对象，进行高效的调试。通过提前布下的错误检测机制，我能精确追溯到错误发生地点，精准地进行修正。这进一步凸显了良好的系统设计与维护的重要性。

## 对课程的建议

实际上，我感觉对加密算法的封装很有意义。我感觉对加密算法的封装能让我们实际上了解到加密算法的强大之处，所以我感觉这一点其实挺适合高级的要求。

感觉我能希望老师多布置一些巩固基础的作业，但是也别太多了。我有这个想法是因为我们这个学期的课实在是太多了……布置得多感觉时间匆忙，什么都巩固不了。

## 总结

这次写AES算法，感觉我对密码算法的理解更加深化，写得也更为熟练了。我会用更强的信心，应对以后的挑战。