《密码学》课程设计实验报告

***本次实验源文件和实验报告均已上传至：**

sterzhang/Cryptology: Jianshu Zhang's cryptological experiments (github.com)

实验序号：01　　实验项目名称：分组密码 DES

学　　号	2021302181216	姓　　名	张鉴殊	专业、班	21 信安
实验地点	C202	指导教师	余荣威	时间	2023.12.1

一、　实验目的及要求

教学目的：

(1) 掌握分组密码的基本概念；

(2) 掌握 DES（3DES）密码算法；

(3) 了解 DES（3DES）密码的安全性；

(4) 掌握分组密码常用工作模式及其特点；

(5) 熟悉分组密码的应用。

实验要求：

(1) 复习掌握（古典密码）使用的置换、代替、XOR、迭代等技术；

(2) 比较 DES 中代替技术与古典密码中的联系与区别；

(3) 理解 S 盒、P 置换等部件的安全性准则；

(4) 实现 DES 算法的编程与优化。

二、实验设备（环境）及要求

Linux

python

```
import binascii
import base64
```

三、实验内容与步骤（采用源码+解释关键部分的展示方式）

1. DES 子密钥扩展算法的实现

输入：64 位密钥

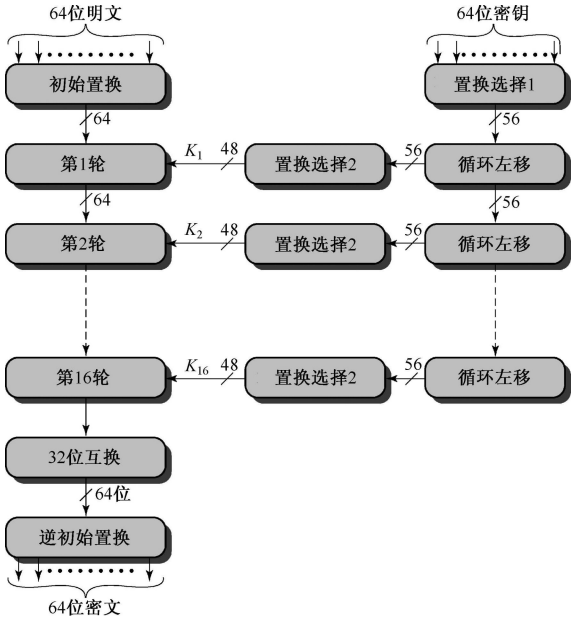
子过程：

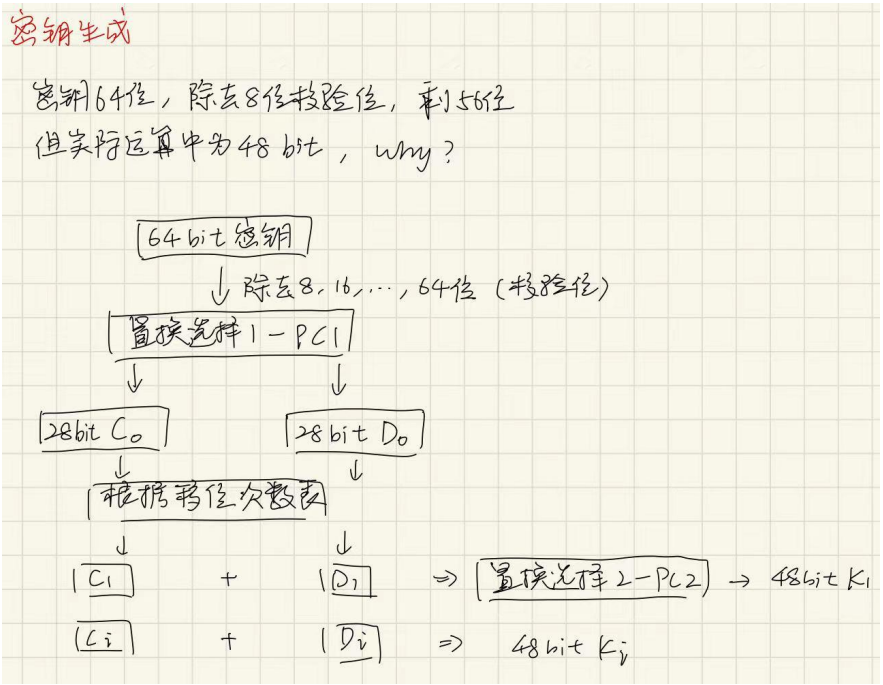
（1）置换选择 1（教材 图 3-3）

（2）循环左移（教材 表 3-1）

（3）置换选择 2（教材 图 3-4）

输出：16 个 48 位长的子密钥。





<---我的流程图

1)密钥生成函数的输入为 C0，D0，这两是初始 64bit 密钥在经过 PC1 的 permutation 后拆分后得到的 2 个 28bit 的部分：

```
#generate keys for every 16 rounds
def gen_keys(C0, D0):
```

```
#save each round's key in a dict
keys= dict.fromkeys(range(0,17))
#import left rotate values
shifts = init.left_rotate_values
```

2）上面定义了一个 keys 的字典，用来存每一轮的 keys，这里定义一个循环移位的函数 f_sl，这可以保证左移溢出的部分在右侧重新进来：

```
#define a function of shift left(the overflow parts will go to the base of the value)
f_sl = lambda val, shift_left_number, max_bit: \
    (val << shift_left_number % max_bit) & (2**max_bit - 1) | \
    ((val & (2**max_bit - 1))) >> (max_bit - (shift_left_number % max_bit))
```

3）初始化 keys[0]，再进入迭代生成，根据次数 i 选择对应的 sf 移位的数值，保存在 keys[i]中：

```
#initialize C0, D0
C0 = f_sl(C0, 0, 28)
D0 = f_sl(D0, 0, 28)
keys[0] = (C0, D0)
#generate keys for each round
for i, sf in enumerate(shifts, start=1):
    C_old = keys[i-1][0]
    D_old = keys[i-1][1]
    Ci = f_sl(C_old, sf, 28)
    Di = f_sl(D_old, sf, 28)
    keys[i] = (Ci, Di)
```

4）定义一个合并函数，用于后续合并 C 和 D 到一个完整的 56 位的密钥 K，且由于第 0 次是初始化的，后续用的是 1~16 次的，所以删除第一次的，最后返回 keys 字典，便于后续取用：

```
#merge 28,28 -> 56
f_merge = lambda c, d, bits: \
  (c & (2**bits - 1)) << bits | (d & (2**bits - 1))

#save 1st ~ 16th keys
del keys[0]
for key in keys.keys():
    c = keys[key][0]
    d = keys[key][1]
    K_i = f_merge(c, d, 28)
    K_i = Permu(K_i, 56, init.PC2)
    keys[key] = K_i
```

```
    return keys
```

2．DES 局部加密函数 f 的实现

加密函数是 DES 的核心部分。它的作用是在第 i 次加密迭代中用子密钥 K_i 对 R_{i-1} 进行加密。

输入：32 位 R_{i-1} 和 48 位子密钥 K_i

子过程：

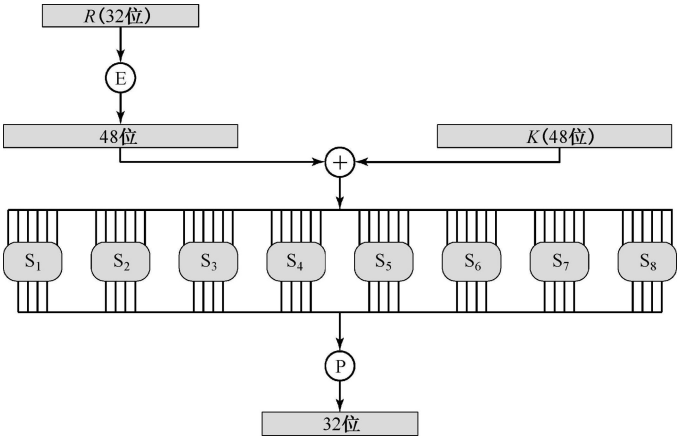
（1） 扩展置换 E（教材 图 3-7）：将 32 位 R_{i-1} 扩展为 48 位；

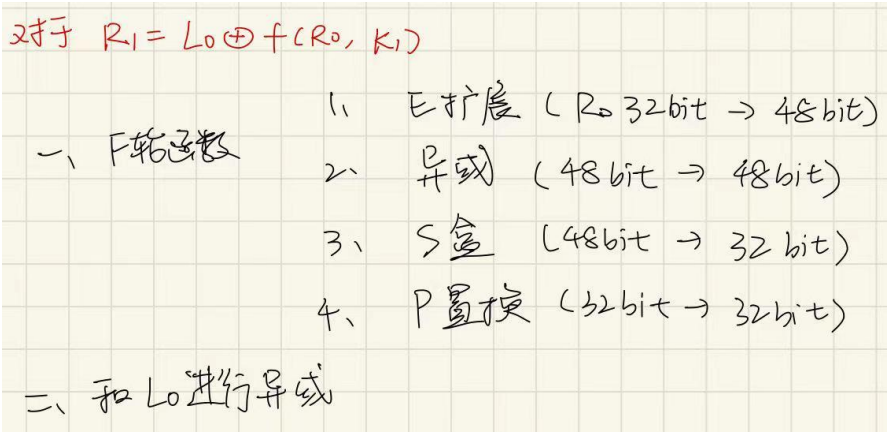
（2） 异或操作：步骤（1）的 48 位结果与子密钥 K_i 按位模 2 相加；

（3） 代替 S 盒（教材 表 3-2）：步骤（2）的 48 位结果分成 6 位×8 组压缩为 4 位×8 组，即 32 位输出；

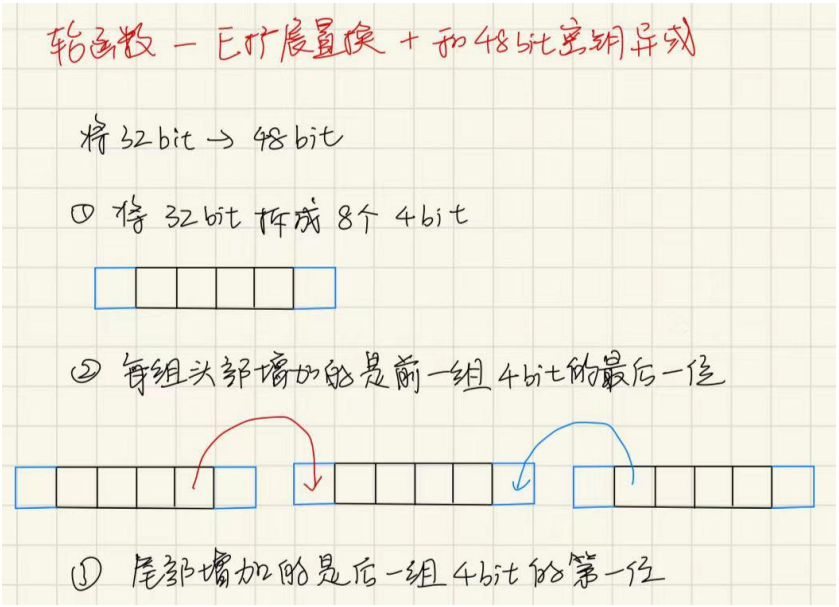
（4） 置换运算 P（教材 图 3-8）：32 位输入/输出。

输出：32 位 $f(R_{i-1}, K_i)$

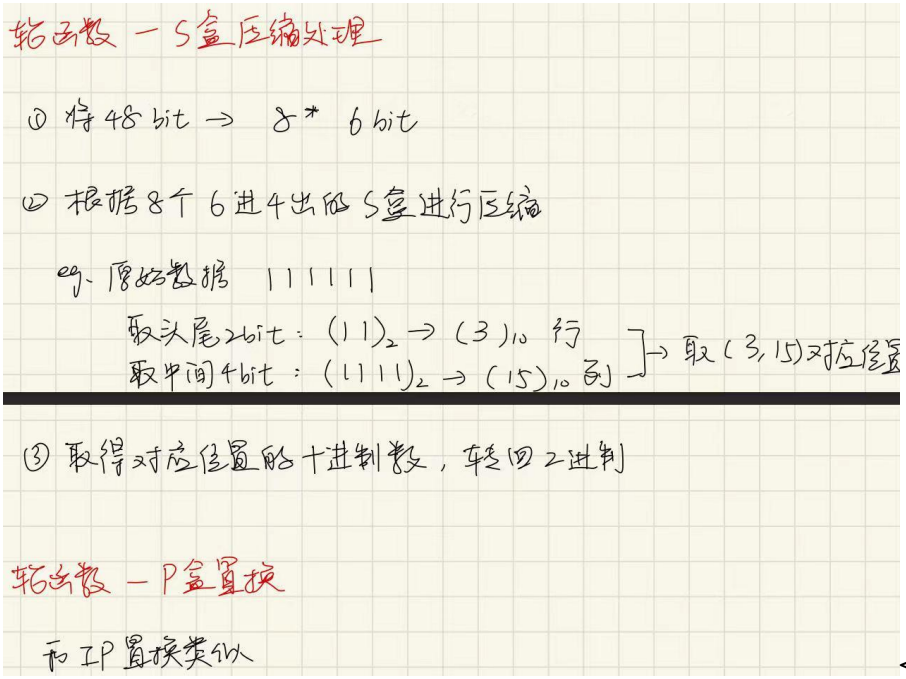




<---我的流程图



<--- 我的流程图



<---我的流程图

1）这个函数用来加密迭代，首先要对 R 进行 E 扩展置换，这里用到了 E 表，接着和 48 位的该轮对应密钥进行异或运算：

```
# F round function
def Func(R_i, K_i):
    #Step1. R_i(32bits) --E--> R_i(48bits)
    #permutate R_i using E table
    R_i = Permu(R_i, 32, init.E)

    #Step2. R_i | K_i+1
    #R_i | K_i+1
```

```
    R_i = R_i ^ K_i
    print("R_i ^ K_i:{}".format(format_binary(R_i, 48)))
```

2）需要用到 S 盒进行压缩处理：

```
    #Step3. 48bits -> 32bits using S box
    R_i = S_box(R_i)
    print("S_box(R_i):{}".format(format_binary(R_i, 32)))
```

这里是 S 盒的实现代码，首先现将 48 位分成 8*6 位，再根据 8 个 6 进 4 出的 S 盒进行压缩，取头尾的 2bit 作为行，取中间 4bit 对应十进制数作为列，取出对应位置的数，转换为 2 进制，作为转换后的结果：

```
def S_box(R_i):
    #save 8 partitions, each parts have 6 bits
    parts = []
    for i in range(7, -1, -1):
        #0x3f -> 11 1111
        x = (R_i >> (i*6) & (0x3f))
        parts.append(x)

    #transfer the orgin 6 digits to 4 digits corresponds to Sbox
    for i in range(8):
        part = parts[i]
        #head and rear combine to the row
        row = (((part >> 5) & 0x1) << 1) | (part & 0x1)
        #4 middle digits combine to the col
        col = (part >> 1) & 0xf
        parts[i] = init.Sboxes[i][row*16+col]

    R_i = 0
    for i in range(8):
        part = parts[i]
        R_i = (R_i << 4) | (part & 0xf)

    return R_i
```

3）最后再用个 P 盒置换即可：

```
    #Step4. Pertubation using P
    R_i = Permu(R_i, 32, init.P)

    return R_i
```

3. DES 加密过程完整实现

① 64 位密钥经子密钥产生算法产生出 16 个子密钥：K_1 ，K_2 ，...，K_{16}，分别供第一次，第二次，...，第十六次加密迭代使用。

② 64 位明文首先经过初始置换 ***IP***（Initial permutation），将数据打乱重新排列并分成左右两半。左边 32 位构成 L_0，左边 32 位构成 R_0。

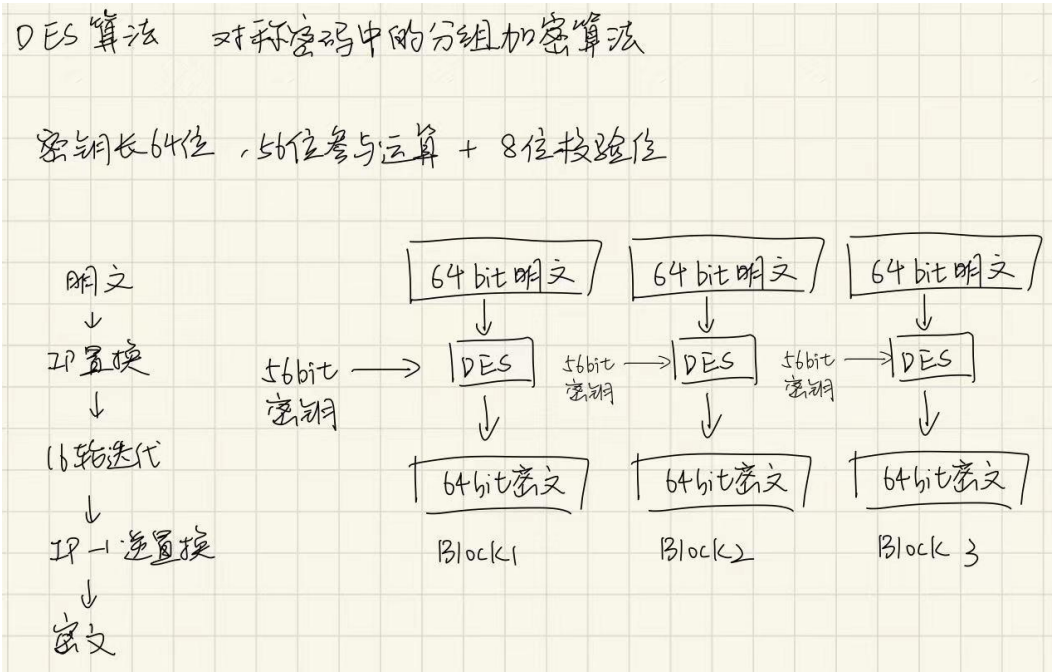
③ 由加密函数 f 实现子密钥 K_1 对 R_0 的加密，结果为 32 位的数据组 $f(R_0 , K_1)$。$f(R_0 , K_1)$再与 L_0 模 2 相加，又得到一个 32 位的数据组 $L_0 \oplus f(R_0 , K_1)$。以 $L_0 \oplus f(R_0 , K_1)$ 作为第二次加密迭代的 R_1，以 R_0 作为第二次加密迭代的 L_1。至此，第一次加密迭代结束。

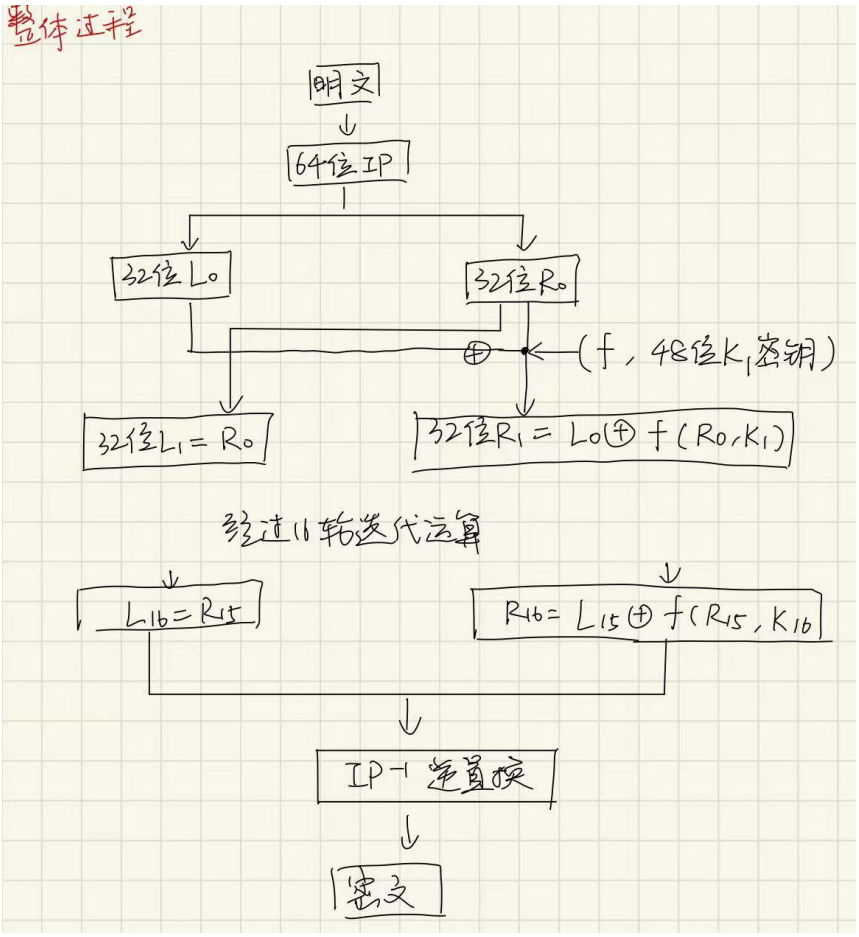
④ 第二次加密迭代至第十六次加密迭代分别用子密钥 K_2 ，...，K_{16} 进行，其过程与第一次加密迭代相同。

⑤ 第十六次加密迭代结束后，产生一个 64 位的数据组。以 R_{16} 作为其左边 32 位，以 L_{16} 作为其右边 32 位，两者合并再经过逆初始置换 IP^{-1}，将数据重新排列，便得到 64 位密文。至此加密过程全部结束。

综上可将 DES 的加密过程用如下的数学公式描述：

$$\begin{cases} L_i = R_{i-1} \\ R_i = L_{i-1} \oplus f(R_{i-1}, K_i) \\ i = 1,2,3,\ldots 16 \end{cases} \qquad (3\text{-}1)$$





<---我的流程图

1）来看我定义的 DES 函数，这个函数对于整个过程都有个概括性的认识，尽管调用了各个具体的实现，但是可以从整个宏观的流程对 DES 算法进行系统性的认识，由于 DES 对于加密解密的宏观过程类似，只是 input 和 output 相反，所以共同在 DES 这个函数中实现：

```
def DES(message, key, decrypt=False):
    if decrypt:
        print("Decrypting")
    else:
        print("Crypting")
```

1.1）这部分是后续做验证性实验查看 Original Key 的：

```
    key_binary_str = "{:064b}".format(key)
    key_binary_str_grouped = ' '.join([key_binary_str[i:i+8] for i in range(0,
len(key_binary_str), 8)])
    print("Original Key: ", key_binary_str_grouped)
```

2）第一步：先求密钥，将 64 位密钥经过 PC1 的 permutation 后，自动去除 8 位

的校验位，得到 56 位的 key：

```
   print("\n\n*****DES step1. 64bits key --PC1--> 56bits key*****")
    #DES step1. 64bits key --PC1--> 56bits key
    key = Permu(key, 64, init.PC1)
```

2.1）验证 key：

```
   key_binary_str = "{:056b}".format(key)
    key_binary_str_grouped = ' '.join([key_binary_str[i:i+8] for i in range(0,
len(key_binary_str), 8)])
    print("Key after PC1 permutation: ", key_binary_str_grouped)
```

2.2）开始生成 16 轮各自的 key：

```
   #generate 16 keys for each iteration
    C0 = (key >> 28) & 0xfffffff
    D0 = (key) & 0xfffffff
    keys = gen_keys(C0, D0)
    print("--------16 keys----------")
    for key, value in keys.items():
        print("Key {}: {}".format(key, format_binary(value, 48)))
    print("-------------------------")
```

3）第二步，对 64 位的明文，进行 IP permutation：

```

    print("\n\n*****DES step2. permutate 64bits message using IP******")
    #DES step2. permutate 64bits message using IP
```

3.1）验证：

```
    message_binary_str = "{:064b}".format(message)
    message_binary_str_grouped = ' '.join([message_binary_str[i:i+8] for i in range(0,
len(message_binary_str), 8)])
    print("Original Message: ", message_binary_str_grouped)
```

3.2）进行 IP permutation：

```
    message = Permu(message, 64, init.IP)
```

3.3）验证

```
    message_binary_str = "{:064b}".format(message)
    message_binary_str_grouped = ' '.join([message_binary_str[i:i+8] for i in range(0,
len(message_binary_str), 8)])
    print("Message after IP: ", message_binary_str_grouped)
```

4）第三步：拆分 64 位的 IP permutation 后的 message 为 32 位的 L0，R0：

```
print("\n\n*****DES step3. split the 64bits message to 32bits L0 and R0******")
#DES step3. split the 64bits message to 32bits L0 and R0
L0 = (message >> 32) & 0xffffffff
R0 = message & 0xffffffff
```

5）第四步：开始 16 轮的迭代运算：

```
print("\n\n*****DES step4. apply round function 16 times******")
#DES step4. apply round function 16 times
L_last = L0
R_last = R0
print("L0:{}".format(format_binary(L_last, 32)))
print("R0:{}\n".format(format_binary(R_last, 32)))
```

5.1）对于 L_i 的求解是直接将上一轮的 R 赋值给它；对于 R_i 的求解复杂一些，涉及到两步的运算，先是用我定义 Func 函数，将上一轮的 R 和本轮所用到的 K 密钥进行计算，随后用计算所得的 32 位的值和上一轮的 L 进行异或运算，得到 R：

```
if decrypt:
    for i in range(16,0,-1):
        tmp = L_last
        L_last = R_last
        R_last = Func(R_last, keys[i]) ^ tmp
else:
    for i in range(1,17):
        tmp = L_last
        L_last = R_last
        R_last = Func(R_last, keys[i]) ^ tmp
        print("K{}:{}".format(i,format_binary(keys[i], 48)))
        # if i%8 == 0:
        print("L{}:{}".format(i,format_binary(L_last, 32)))
        print("R{}:{}\n".format(i,format_binary(R_last, 32)))
```

6）第五步：将最后一轮的 L 和 R 结合起来：

```
#DES step5. combine the last round of R,L together to get the final result
result_message = ((R_last & 0xffffffff) << 32) | (L_last & 0xffffffff)
```

7）第六步：将拼接起来的部分用 IP 逆做 permutation，得到最终的密文：

```
    ##DES step6. permutate result using IP_INV
    result_message = Permu(result_message, 64, init.IP_INV)
    return result_message
```

4. DES 解密过程实现

由于 DES 的运算是对和运算，所以解密和加密可共用同一个运算，只是子密钥使用的顺序不同。

把 64 位密文当作明文输入，而且第一次解密迭代使用子密钥 K_{16}，第二次解密迭代使用子密钥 K_{15}，…，第十六次解密迭代使用子密钥 K_1，最后的输出便是 64 位明文。

解密过程可用如下的数学公式描述：

$$\begin{cases} R_{i-1} = L_i \\ L_{i-1} = R_i \oplus f(L_i, K_i) \\ i = 16,15,14,\ldots,1 \end{cases} \quad (3\text{-}2)$$

1）这里解密还是用的 DES，只在下面两部分有区别，进函数的时候会判断是不是 decrypt 为 true，接着在具体实现解密的时候，如果是解密的话，DES 的 message 对应的是密文，key 对应的是第 16 次的作为第一轮的 key，在格式上本质上和加密过程没有区别：

```
def DES(message, key, decrypt=False):
    if decrypt:
        print("Decrypting")
    else:
        print("Crypting")
```

```
if decrypt:
        for i in range(16,0,-1):
            tmp = L_last
            L_last = R_last
            R_last = Func(R_last, keys[i]) ^ tmp
    else:
        for i in range(1,17):
            tmp = L_last
            L_last = R_last
            R_last = Func(R_last, keys[i]) ^ tmp
            print("K{}:{}".format(i,format_binary(keys[i], 48)))
            # if i%8 == 0:
            print("L{}:{}".format(i,format_binary(L_last, 32)))
            print("R{}:{}\n".format(i,format_binary(R_last, 32)))
```

5. DES 的 S 盒密码学特性（**重点**）

通过编程实现或者手工计算，试验证 ***S*** 盒的以下准则：

① 输出不是输入的线性和仿射函数；

② 任意改变输入中的一位，输出至少有两位发生变化；

③ 对于任何 ***S*** 盒和任何输入 x，$\boldsymbol{S}(x)$和 $\boldsymbol{S}(x \oplus 001100)$至少有两位不同，这里 x 是一个 6 位的二进制串；

④ 对于任何 ***S*** 盒和任何输入 x，以及 $y,z \in GF(2)$，$\boldsymbol{S}(x) \neq \boldsymbol{S}(x \oplus 11yz00)$，这里 x 是一个 6 位的二进制串；

⑤ 保持输入中的 1 位不变，其余 5 位变化，输出中的 0 和 1 的个数接近相等。

例如，可通过如下步骤验证②、③两条：

设 S 盒的输入为 X，输出为 Y。（X 和 Y 都以二进制表示）

（1）对于已知输入值 X_1=110010 和 X_2=100010，分别求出对应的输出值 Y_1 和 Y_2。

（2）比较输出值 Y_1 和 Y_2 各位的异同，即按位计算 $Y_1 \oplus Y_2$。

根据上面得出的结果试说明 S 盒对于 DES 的安全性影响。

1）为了实现验证，定义了一个计算位不同的函数：

```
def count_bit_diffs(a, b):
    """Count the number of different bits"""
    return bin(a ^ b).count('1')
```

2）先验证①：输出不是输入的线性和仿射函数：在二进制域中当满足加性和齐次性，一个函数被认为是线性的。即对于任意两个输入 x 和 y，以及标量 a，满足 $f(x \oplus y) = f(x) \oplus f(y)$ 和 $f(a \cdot x) = a \cdot f(x)$，在二进制域中，加法和乘法分别是 XOR（⊕）和 AND（·）操作；一个仿射函数是一个线性函数加上一个常数偏移，即如果 f(x) 是线性的，那么 $g(x) = f(x) \oplus c$（其中 c 是一个常数）是仿射的：

```
    # ①:输出不是输入的线性和仿射函数
    for x in range(64):
        for y in range(64):
            if S_box_func(x ^ y) == S_box_func(x) ^ S_box_func(y):
                print(f"非线性未通过")
                test_passed = False
                break

    for c in range(1, 64):  # 常数 c 从 1 到 63
        constant = None
        for x in range(64):
            result = S_box_func(x) ^ S_box_func(x ^ c)
```

```
            if constant is None:
                constant = result  # 第一次设定比较标准
            elif result != constant:
                break  # 如果发现不一致，则终止内层循环
        else:
            # 如果完成了所有 x 的遍历且没有中断，则 S 盒可能是仿射的
            print(f"S 盒可能是仿射的，c={c:06b}")
            test_passed = False
```

3）开始对所有 64 种 6 位的输入进行枚举实验，首先验证② 任意改变输入中的一位，输出至少有两位发生变化：

```
def test_s_box_properties(S_box_func, s_boxes):
    test_passed = True
    # ②:任意改变输入中的一位，输出至少有两位发生变化
    for i in range(64):  # 对于所有 64 种可能的 6 位输入
        original_input = i
        original_output = S_box_func(original_input)
        for j in range(6):  # 改变输入的每一位
            modified_input = original_input ^ (1 << j)
            modified_output = S_box_func(modified_input)
            if count_bit_diffs(original_output, modified_output) < 2:
                print(f"②未通过: 输入 {original_input:06b} 和 {modified_input:06b}")
                test_passed = False
```

4）接着验证③ 对于任何 S 盒和任何输入 x，S(x)和 S(x⊕001100)至少有两位不同，这里 x 是一个 6 位的二进制串：

```
# ③:对于任何 S 盒和任何输入 x，S(x)和 S(x⊕001100)至少有两位不同，这里 x 是一个 6 位的二进制串
    for i in range(64):  # 对于所有 64 种可能的 6 位输入
        original_input = i
        original_output = S_box_func(original_input)
        modified_input = original_input ^ 0b001100  # XOR with 001100
        modified_output = S_box_func(modified_input)

        if count_bit_diffs(original_output, modified_output) < 2:
            print(f"③未通过: 输入 {original_input:06b} 和 {modified_input:06b}")
            test_passed = False
    return test_passed
```

5）接着验证④:对于任何 S 盒和任何输入 x，以及 y,z∈GF(2)，S(x)≠S(x⊕11yz00)，这里 x 是一个 6 位的二进制串：

```
# ④:对于任何 S 盒和任何输入 x，以及 y,z∈GF(2)，S(x)≠S(x⊕11yz00)，这里 x 是一个 6 位的二进制串
    for x in range(64):
        for y in range(2):  # y 为 0 或 1
            for z in range(2):  # z 为 0 或 1
                x_prime = x ^ (0b110000 | (y << 3 | z << 2))
                if S_box_func(x) == S_box_func(x_prime):
                    print(f"④未通过: 输入 {x:06b} 和 {x_prime:06b}")
                    test_passed = False
```

6）验证⑤:保持输入中的 1 位不变，其余 5 位变化，输出中的 0 和 1 的个数接近相等，这里将阈值设置成 2，如果超过 2 就是不平衡：

```
# ⑤:保持输入中的 1 位不变，其余 5 位变化，输出中的 0 和 1 的个数接近相等
    for fixed_bit in range(6):  # 对于每个固定位
        count_0 = 0
        count_1 = 0
        for x in range(64):
            if (x >> fixed_bit) & 1 == 0:  # 如果固定位是 0
                if S_box_func(x) & 1 == 0:  # 如果输出的最低位是 0
                    count_0 += 1
                else:
                    count_1 += 1
        if abs(count_0 - count_1) > 2:  # 根据某个阈值 2 判断是否平衡
            print(f"⑤未通过: 固定位 {fixed_bit}")
            test_passed = False

    return test_passed
```

7）进行了验证性代码实现，结果如下，可以看到均满足了这些性质：

```
# test of S box
if test_s_box_properties(S_box, init.Sboxes):
    print("Successful")
else:
    print("Error")
```

```
(py38) root@6dceeba50b41:~/project/test/DES# python des-main.py
Successful
```

6.验证教材 P64 页实例（重点）

这里设置的所有数值都和书本上一致，并打印了每一轮的细节数值，都能一一与

书上对应上，这里还设置了自动化验证，对与加密解密都进行了自动化测试，可以看到结果均为 Successful：

```
successful encrypting!
------------------------------------------------------------------------
```

```
successful decrypting!
```

这是代码的具体实现：

```
# test data
key_binary = "00110001 00110010 00110011 00110100 00110101 00110110 00110111 00111000"
input_data_binary = "00110000 00110001 00110010 00110011 00110100 00110101 00110110 00110111"
expected_output_binary = "10001011 10110100 01111010 00001100 11110000 10101001 01100010 01101101"

# bin str -> int
key = binary_str_to_int(key_binary)
input_data = binary_str_to_int(input_data_binary)
expected_output = binary_str_to_int(expected_output_binary)
```

```
# encrypt
encrypted_data = DES(input_data, key,decrypt=False)
```

```
# int -> bin str
encrypted_data_binary = int_to_binary_str(encrypted_data, 64)
```

```
# test
if encrypted_data_binary == expected_output_binary.replace(" ", ""):
    print("successful encrypting!")
else:
    print("error when testing")
    print("encrypted_data_binary: ", encrypted_data_binary)
```

```
print("------------------------------------------------------------------------\n\n")
# decrypt
decrypted_data = DES(encrypted_data, key,decrypt=True)
```

```
# int -> bin str
decrypted_data_binary = int_to_binary_str(decrypted_data, 64)
```

```
# test
if decrypted_data_binary == input_data_binary.replace(" ", ""):
    print("successful decrypting!")
else:
    print("error when testing")
```

```
print("decrypted_data_binary: ", decrypted_data_binary)
```

这是终端返回的打印的具体细节，均与书本对应：

```
(py38) root@6dceeba50b41:~/project/test/DES# python des-main.py
Crypting
Original Key:  00110001 00110010 00110011 00110100 00110101 00110110 00110111 00111000


*****DES step1. 64bits key --PC1--> 56bits key*****
Key after PC1 permutation:  00000000 00000000 11111111 11110110 01100111 10001000 00001111
--------16 keys----------
Key 1: 01010000 00101100 10101100 01010111 00101010 11000010
Key 2: 01010000 10101100 10100100 01010000 10100011 01000111
Key 3: 11010000 10101100 00100110 11110110 10000100 10001100
Key 4: 11100000 10100110 00100110 01001000 00110111 11001011
Key 5: 11100000 10010110 00100110 00111110 11110000 00101001
Key 6: 11100000 10010010 01110010 01100010 01011101 01100010
Key 7: 10100100 11010010 01110010 10001100 10101001 00111010
Key 8: 10100110 01010011 01010010 11100101 01011110 01010000
Key 9: 00100110 01010011 01010011 11001011 10011010 01000000
Key 10: 00101111 01010001 01010001 11010000 11000111 00111100
Key 11: 00001111 01000001 11011001 00011001 00011110 10001100
Key 12: 00011111 01000001 10011001 11011000 01110000 10110001
Key 13: 00011111 00001001 10001001 00100011 01101010 00101101
Key 14: 00011011 00101000 10001101 10110010 00111001 10010010
Key 15: 00011001 00101100 10001100 10100101 00000011 00110111
Key 16: 01010001 00101100 10001100 10100111 01000011 11000000
------------------------


*****DES step2. permutate 64bits message using IP******
Original Message:  00110000 00110001 00110010 00110011 00110100 00110101 00110110 00110111
Message after IP:  00000000 11111111 11110000 10101010 00000000 11111111 00000000 11001100


*****DES step3. split the 64bits message to 32bits L0 and R0******
```

```
*****DES step4. apply round function 16 times******
L0:00000000 11111111 11110000 10101010
R0:00000000 11111111 00000000 11001100

R_i ^ K_i:01010000 00111011 01010010 11010111 00111100 10011010
S_box(R_i):01101101 10000010 00001110 11110000
K1:01010000 00101100 10101100 01010111 00101010 11000010
L1:00000000 11111111 00000000 11001100
R1:00010010 10000111 00110111 10110011

R_i ^ K_i:11011010 11111000 10101010 11001010 01011110 11100001
S_box(R_i):01110010 01101011 10010010 00100010
K2:01010000 10101100 10100100 01010000 10100011 01000111
L2:00010010 10000111 00110111 10110011
R2:11100001 10011100 10000110 10001010

R_i ^ K_i:10100000 10010000 11011111 10110110 01010000 11011001
S_box(R_i):11011111 01111001 00100010 00000000
K3:11010000 10101100 00100110 11110110 10000100 10001100
L3:11100001 10011100 10000110 10001010
R3:11010110 00101110 11110111 01100101

R_i ^ K_i:00001010 01100111 01111011 00110010 11011100 11000000
S_box(R_i):01001011 11110111 10111111 01011101
K4:11100000 10100110 00100110 01001000 00110111 11001011
L4:11010110 00101110 11110111 01100101
R4:00011110 11100101 01111111 00100110

R_i ^ K_i:11101111 01000001 00101100 10000001 00011001 00100101
S_box(R_i):00001100 10010111 01000110 10111110
K5:11100000 10010110 00100110 00111110 11110000 00101001
L5:00011110 11100101 01111111 00100110
R5:01011000 01000000 11100010 01011100
```

```
R_i ^ K_i:11001111 10010000 01110011 00010010 00011111 10011010
S_box(R_i):10110000 11010100 01000100 00100000
K6:11100000 10010010 01110010 01100010 01011101 01100010
L6:01011000 01000000 11100010 01011100
R6:00011010 01100000 01101000 00101100

R_i ^ K_i:10101011 10010001 01110010 10111001 10101000 01100010
S_box(R_i):01100000 00000001 10000111 01101011
K7:10100100 11010010 01110010 10001100 10101001 00111010
L7:00011010 01100000 01101000 00101100
R7:11010001 01110010 01001100 01010100

R_i ^ K_i:11001100 01111000 11110110 11000000 11011100 11111001
S_box(R_i):10110111 10101110 11111001 01010011
K8:10100110 01010011 01010010 11100101 01011110 01010000
L8:11010001 01110010 01001100 01010100
R8:01101001 10110110 00010011 11111010

R_i ^ K_i:00010011 01101110 11111111 11000001 11100101 10110100
S_box(R_i):11010110 01011110 11111011 01111010
K9:00100110 01010011 01010011 11001011 10011010 01000000
L9:01101001 10110110 00010011 11111010
R9:10101110 10000101 11111000 10000110

R_i ^ K_i:01111010 10000101 01011010 00101111 11010011 00110001
S_box(R_i):01111010 01011100 01111000 10001111
K10:00101111 01010001 01010001 11010000 11000111 00111100
L10:10101110 10000101 11111000 10000110
R10:00010101 10111001 10001001 00011001

R_i ^ K_i:10000101 11111100 00101010 11011100 00110110 01111110
S_box(R_i):11110101 10111011 10011111 00101000
K11:00001111 01000001 11011001 00011001 00011110 10001100
L11:00010101 10111001 10001001 00011001
R11:00010011 01100101 00011111 11011000
```

```
R_i ^ K_i:00010101 00101010 10010011 01010111 10001110 01000001
S_box(R_i):01110111 11110111 11110001 11100001
K12:00011111 01000001 10011001 11011000 01110000 10110001
L12:00010011 01100101 00011111 11011000
R12:11110000 11101100 01110110 10001110

R_i ^ K_i:01100101 00011110 11010001 00011001 10111110 01110000
S_box(R_i):10011100 01010100 00011011 11100000
K13:00011111 00001001 10001001 00100011 01101010 00101101
L13:11110000 11101100 01110110 10001110
R13:00100111 11011100 00101011 11001011

R_i ^ K_i:10001011 11010110 01110101 10100111 01000111 11000100
S_box(R_i):00011110 11000101 00010100 01101000
K14:00011011 00101000 10001101 10110010 00111001 10010010
L14:00100111 11011100 00101011 11001011
R14:00011000 11110101 01100011 10010100

R_i ^ K_i:00010110 00111011 00100110 00010101 01111111 10011111
S_box(R_i):01111000 00110000 00101110 00100010
K15:00011001 00101100 10001100 10100101 00000011 00110111
L15:00011000 11110101 01100011 10010100
R15:00110011 11110110 10101101 01000101

R_i ^ K_i:11001011 01010011 00100001 11110010 11101001 11001010
S_box(R_i):11000111 11110011 00000011 10001111
K16:01010001 00101100 10001100 10100111 01000011 11000000
L16:00110011 11110110 10101101 01000101
R16:11010100 00010110 10001010 10100001

successful encrypting!
```

```
Decrypting
Original Key:  00110001 00110010 00110011 00110100 00110101 00110110 00110111 00111000


*****DES step1. 64bits key --PC1--> 56bits key*****
Key after PC1 permutation:  00000000 00000000 11111111 11110110 01100111 10001000 00001111
--------16 keys----------
Key 1: 01010000 00101100 10101100 01010111 00101010 11000010
Key 2: 01010000 10101100 10100100 01010000 10100011 01000111
Key 3: 11010000 10101100 00100110 11110110 10000100 10001100
Key 4: 11100000 10100110 00100110 01001000 00110111 11001011
Key 5: 11100000 10010110 00100110 00111110 11110000 00101001
Key 6: 11100000 10010010 01110010 01100010 01011101 01100010
Key 7: 10100100 11010010 01110010 10001100 10101001 00111010
Key 8: 10100110 01010011 01010010 11100101 01011110 01010000
Key 9: 00100110 01010011 01010011 11001011 10011010 01000000
Key 10: 00101111 01010001 01010001 11010000 11000111 00111100
Key 11: 00001111 01000001 11011001 00011001 00011110 10001100
Key 12: 00011111 01000001 10011001 11011000 01110000 10110001
Key 13: 00011111 00001001 10001001 00100011 01101010 00101101
Key 14: 00011011 00101000 10001101 10110010 00111001 10010010
Key 15: 00011001 00101100 10001100 10100101 00000011 00110111
Key 16: 01010001 00101100 10001100 10100111 01000011 11000000
------------------------
```

```
*****DES step2. permutate 64bits message using IP******
Original Message:  10001011 10110100 01111010 00001100 11110000 10101001 01100010 01101101
Message after IP:  11010100 00010110 10001010 10100001 00110011 11110110 10101101 01000101


*****DES step3. split the 64bits message to 32bits L0 and R0******


*****DES step4. apply round function 16 times******
L0:11010100 00010110 10001010 10100001
R0:00110011 11110110 10101101 01000101
```

```
R_i ^ K_i:11001011 01010011 00100001 11110010 11101001 11001010
S_box(R_i):11000111 11110011 00000011 10001111
R_i ^ K_i:00010110 00111011 00100110 00010101 01111111 10011111
S_box(R_i):01111000 00110000 00101110 00100010
R_i ^ K_i:10001011 11010110 01110101 10100111 01000111 11000100
S_box(R_i):00011110 11000101 00010100 01101000
R_i ^ K_i:01100101 00011110 11010001 00011001 10111110 01110000
S_box(R_i):10011100 01010100 00011011 11100000
R_i ^ K_i:00010101 00101010 10010011 01010111 10001110 01000001
S_box(R_i):01110111 11110111 11110001 11100001
R_i ^ K_i:10000101 11111100 00101010 11011100 00110110 01111110
S_box(R_i):11110101 10111011 10011111 00101000
R_i ^ K_i:01111010 10000101 01011010 00101111 11010011 00110001
S_box(R_i):01111010 01011100 01111000 10001111
R_i ^ K_i:00010011 01101110 11111111 11000001 11100101 10110100
S_box(R_i):11010110 01011110 11111011 01111010
R_i ^ K_i:11001100 01111000 11110110 11000000 11011100 11111001
S_box(R_i):10110111 10101110 11111001 01010011
R_i ^ K_i:10101011 10010001 01110010 10111001 10101000 01100010
S_box(R_i):01100000 00000001 10000111 01101011
R_i ^ K_i:11001111 10010000 01110011 00010010 00011111 10011010
S_box(R_i):10110000 11010100 01000100 00100000
R_i ^ K_i:11101111 01000001 00101100 10000001 00011001 00100101
S_box(R_i):00001100 10010111 01000110 10111110
R_i ^ K_i:00001010 01100111 01111011 00110010 11011100 11000000
S_box(R_i):01001011 11110111 10111111 01011101
R_i ^ K_i:10100000 10010000 11011111 10110110 01010000 11011001
S_box(R_i):11011111 01111001 00100010 00000000
R_i ^ K_i:11011010 11111000 10101010 11001010 01011110 11100001
S_box(R_i):01110010 01101011 10010010 00100010
R_i ^ K_i:01010000 00111011 01010010 11010111 00111100 10011010
S_box(R_i):01101101 10000010 00001110 11110000
successful decrypting!
```

7.扩展思考

（1）Feistel 结构为什么可以保证算法的对合性？

在每一轮中，左半部分成为新的右半部分，而原右半部分经过轮函数处理后与原左半部分进行 XOR 操作成为新的左半部分。正是由于 XOR 操作的对称性（即 A XOR B XOR B = A），解密时以相反的密钥顺序执行相同的操作可以

恢复原左半部分和原右半部分。经过所有轮的逆向操作后，原始的左半部分和右半部分都被恢复，从而得到原始的明文数据。

（2）第 16 轮为什么不做左右互换？

保持 Feistel 网络的对合性（即使用相同的过程进行加密和解密）不依赖于最后一轮是否交换。在整个加密过程的最后，不进行交换只是意味着加密和解密过程的开始和结束是对称的。因为在解密过程中，我们以相反的顺序应用密钥，从而逆转之前的每一步操作，包括交换。另外还可以避免多余的操作，如果最后一轮进行了交换，那么在加密结束后，解密之前，我们需要再次交换左右部分以便正确地开始解密过程。通过省略最后一轮的交换，我们避免了这种额外的、不必要的步骤。

（3）如果去掉初始置换和逆初始置换，对算法安全性有影响吗？（提示：算法所有的细节都是公开的）

加密算法的安全性不应依赖于算法的保密性，而应依赖于密钥的保密性。此外，初始置换和逆初始置换在 DES 中并不涉及到密钥，也不提供任何加密强度。

（4）证明 DES 解密算法是加密算法的逆，即 DES 的对合性。

可以在前面的具体实现中看出，在 Feistel 结构中，加密和解密过程使用相同的步骤，但子密钥的应用顺序相反，这也是我为什么用了同一个 DES 函数去简化加密和解密的实现，因为本质上 DES 是具有对合性的。

四、实验结果与数据处理

已经在三中的 6.验证教材 P64 页实例（重点）展示

五、分析与讨论

1. 在用 python 实现的过程中， ^ 和 | 都是位操作，但为什么需要采用^呢？
因为它们在加密算法中的作用和结果大不相同。1. R_i ^ K_i（位异或操作）
操作：异或操作对于每一位，当两个输入位不相同时结果为 1，相同时结果为 0。这种异或操作是可逆的。也就是说，对同一个值再次执行异或操作可以恢复原始值（例如，(a ^ b) ^ b 会得到 a）。在加密算法中，异或操作用于混合密钥和数据，因为它将密钥的特性引入到数据中，而且这个过程是可逆的，这对于加密和解密都是必要的。 R_i | K_i（位或操作）或操作对于每一位，只要有一个输入位是 1，结果就是 1。只有当两个输入位都是 0 时，结果才是 0。或操作不是可逆的。一旦两个数进行了或操作，就无法从结果中分离出原始的数。位或操作通常不用于传统的加密算法中密钥和数据的混合，因为它不是可逆的。相比之下，R_i | K_i 可能会让数据中的一些位信息丢失（因为任何与 1 进行或操作的位都会变为 1）。

2. 接着上面的问题，为什么在加密算法中需要可逆操作的算法。
因为加密的主要目的是允许信息安全地传输或存储，同时确保只有授权的接收者可以访问原始信息。为了实现这一点，接收者必须能够将加密后的数据（密文）转换回原始的数据（明文）。这意味着加密过程必须是可逆的，即必须存在一个相应的解密过程来恢复原始数据。在许多传统的加密系统中（如 DES，AES 等），加密和解密过程是对称的，这意味着它们使用相同的基本操作，但以相反的顺序或以稍微不同的方式（如使用不同的密钥）。这种对称性要求所有加密操作都是可逆的，以便可以通过执行逆操作来解密数据。

六、教师评语 签名： 日期：	成绩