一、初始置换

DES 算法使用 64 位的密钥 key 将 64 位的明文输入块变为 64 位的密文输出块，并把输出块分为 L0、R0 两部分，每部分均为 32 位。初始置换规则如下：

```
# 初始置换矩阵 IP
IP_MATRIX = [58, 50, 42, 34, 26, 18, 10, 2,
             60, 52, 44, 36, 28, 20, 12, 4,
             62, 54, 46, 38, 30, 22, 14, 6,
             64, 56, 48, 40, 32, 24, 16, 8,
             57, 49, 41, 33, 25, 17, 9, 1,
             59, 51, 43, 35, 27, 19, 11, 3,
             61, 53, 45, 37, 29, 21, 13, 5,
             63, 55, 47, 39, 31, 23, 15, 7]

# 运行代码
def IP(plaintext):
    #如果长度不是 64 就退出
    assert len(plaintext) == 64

    IPResult = ""
    #通过循环 进行 IP 置换，其实就是将字符串按照 IP 矩阵重新置换排列一下，并组合
    for i in IP_MATRIX:
        IPResult = IPResult + plaintext[i - 1]#连接
    return IPResult
```

二、加密处理--迭代过程

经过初始置换后，进行 16 轮完全相同的运算，在运算过程中数据与秘钥结合。

函数 f 的输出经过一个异或运算，和左半部分结合形成新的右半部分，原来的右半部分成为新的左半部分。每轮迭代的过程可以表示如下：

Ln = R(n - 1)；　　Rn = L(n - 1)⊕f(Rn-1,kn-1)　　⊕:异或运算

Kn 是向第 N 层输入的 48 位的秘钥，f 是以 Rn-1 和 Kn 为变量的输出 32 位的函数

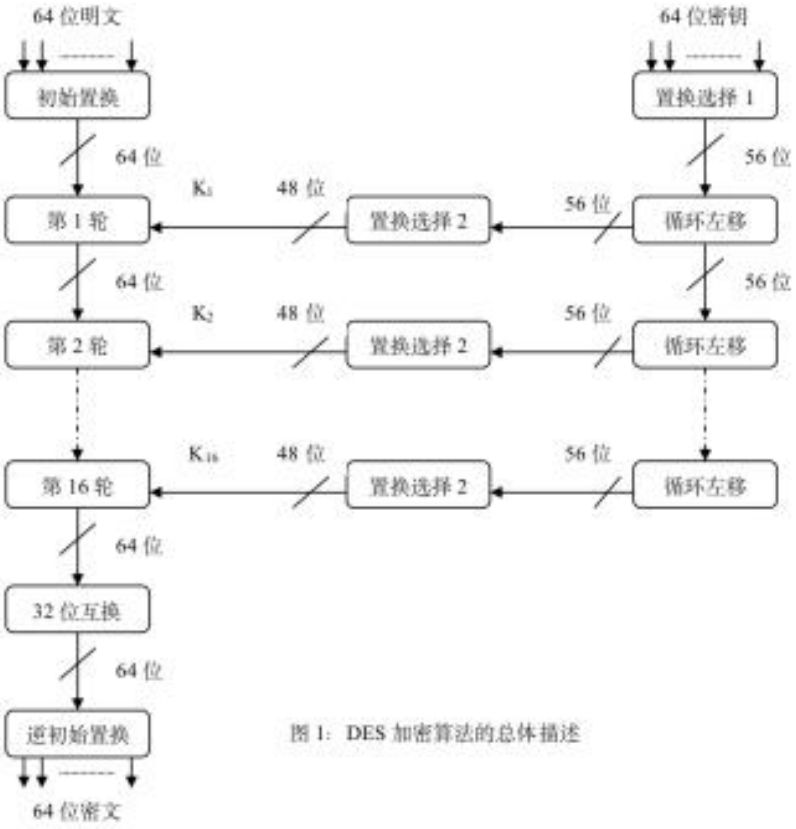


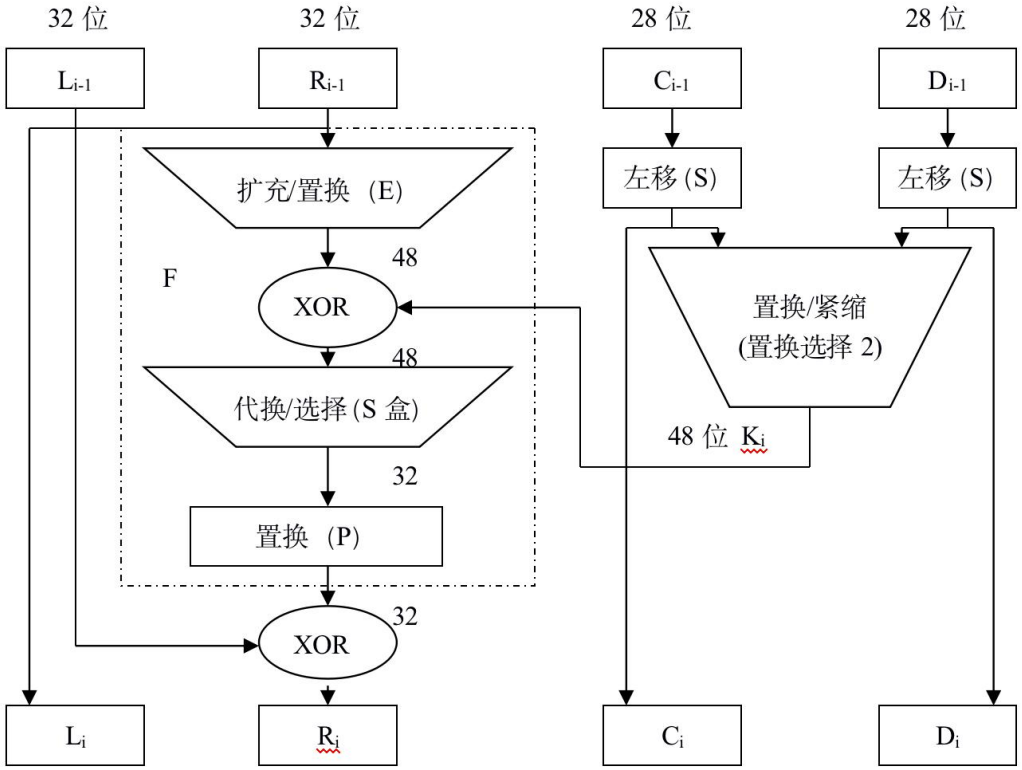
图 1：DES 加密算法的总体描述

16 轮的轮密钥产生：

```
# 置换选择 2 矩阵  从 56 位密钥流中选 48（6*8）位
# 去掉第 9、18、22、25、35、38、43、54 位。
REPLACE2_MATRIXS = [14, 17, 11, 24, 1, 5, 3, 28,
                    15, 6, 21, 10, 23, 19, 12, 4,
                    26, 8, 16, 7, 27, 20, 13, 2,
                    41, 52, 31, 37, 47, 55, 30, 40,
                    51, 45, 33, 48, 44, 49, 39, 56,
                    34, 53, 46, 42, 50, 36, 29, 32]

#shift 函数
def shift(str, shift_times):
    try:
        if len(str) > 28:
            raise NameError
    except TypeError:
        pass

    str = str[shift_times:] + str[0:shift_times]#其实就是将首尾相连
    return str

#生成轮密钥
```

```
def createRoundSecretkey(secretKey): #
    # 如果 key 长度不是 64 就退出
    assert len(secretKey) == 64

    #DES 的密钥由 64 位减至 56 位，每个字节的第 8 位作为奇偶校验位
    #把 56 位 变成 2 个 28 位

    #置换选择 1
    #注意：64 位秘钥降至 56 位秘钥不是说将每个字节的第八位删除，而是通过缩小选择换位表 1（置换选择表 1）的变换变成 56 位
    #C0 28 位（左）
    Clist = [57, 49, 41, 33, 25, 17, 9,
             1, 58, 50, 42, 34, 26, 18,
             10, 2, 59, 51, 43, 35, 27,
             19, 11, 3, 60, 52, 44, 36]
    #D0 28 位（右）
    Dlist = [63, 55, 47, 39, 31, 23, 15,
             7, 62, 54, 46, 38, 30, 22,
             14, 6, 61, 53, 45, 37, 29,
             21, 13, 5, 28, 20, 12, 4]

    # 初试生成 左右两组 28 位密钥
    C0 = ""
    D0 = ""
    # 变换+拼接
    for i in Clist:
        C0 += secretKey[i - 1]
    for i in Dlist:
        D0 += secretKey[i - 1]

    #轮函数生成 48 位密钥
    #定义轮数 左移循环
    Movetimes = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]

    #定义返回的轮密钥（子密钥）
    roundKey = []
    #开始轮置换
    for i in range(0, 16):
        #获取左半边 C0 和 右半边 D0  shift 函数用来左移生成轮数
        C0 = shift(C0, Movetimes[i])
        D0 = shift(D0, Movetimes[i])
        #合并左右部分
        mergedKey = C0 + D0
```

```
        tempkey = ""
        # 压缩置换矩阵   从 56 位里选 48 位
        # 置换选择 2   选出 48 位子密钥
        for i in REPLACE2_MATRIXS:
            tempkey += mergedKey[i - 1]
        assert len(tempkey) == 48
        #把每一轮的生成子密钥加入进去，一共 16 轮
        roundKey.append(tempkey)

    return
roundKey
```




```
    # E 扩展置换矩阵
    E_MATRIX = [32, 1, 2, 3, 4, 5,
                4, 5, 6, 7, 8, 9,
                8, 9, 10, 11, 12, 13,
                12, 13, 14, 15, 16, 17,
                16, 17, 18, 19, 20, 21,
                20, 21, 22, 23, 24, 25,
                24, 25, 26, 27, 28, 29,
                28, 29, 30, 31, 32, 1]
```

#通过扩展置换 E，数据的右半部分 Rn 从 32 位扩展到 48 位。扩展置换改变了位的次序，重复了某些位。

#扩展置换的目的：a、产生与秘钥相同长度的数据以进行异或运算，R0 是 32 位，

子秘钥是 48 位，所以 R0 要先进行扩展置换之后与子秘钥进行异或运算；b、提供更长的结果，使得在替代运算时能够进行压缩。

```
#扩充置换 将 R32 位进行扩充到 48 位 其中 16 位是重复的
def E_expand(Ri):
    retRn = ""
    for i in E_MATRIX:
        retRn += Ri[i - 1]
    assert len(retRn) == 48
    return retRn

#异或

#S 盒运算
# S 盒 的置换矩阵
S_MATRIX = [
      #S1
     (14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7,
      0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8,
      4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0,
      15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13),
      #S2
     (15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10,
      3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5,
      0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15,
      13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9),
      #S3
     (10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8,
      13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1,
      13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7,
      1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12),
      #S4
     (7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15,
      13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9,
      10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4,
      3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14),
      #S5
     (2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9,
      14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
      4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
      11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3),
      #S6
     (12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11,
      10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8,
```

```
        9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6,
        4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13),
        #S7
       (4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1,
        13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6,
        1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2,
        6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12),
        #S8
       (13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7,
        1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2,
        7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8,
        2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11)
]
```

#Rn 扩展置换之后与子秘钥 Kn 异或以后的结果作为输入块进行 S 盒代替运算功能是把 48 位数据变为 32 位数据

代替运算由 8 个不同的代替盒(S 盒)完成。每个 S-盒有 6 位输入，4 位输出。

6 个 6 个为一组，一共 8 组

```
# S 盒替代运算
def S_sub(S_Input):
    #从第二位开始的子串    去掉 0X
    S_Input = bin(S_Input)[2:] #返回二进制

    while len(S_Input) < 48:
        S_Input = "0" + S_Input

    index = 0
    resultStr = ""

    for Slist in S_MATRIX:
        #  输入的首尾两位做为行数 row
        row = int(S_Input[index] + S_Input[index + 5], base=2)
        #  中间四位做为列数 col
        col = int(S_Input[index + 1:index + 5], base=2)
        #  得到  表中目标的  单个四位输出
        target = bin(Slist[row * 16 + col])[2:]

        while len(target) < 4:#补满 4 位
            target = "0" + target

        #  合并单个输出
        resultStr += target
        # index + 6  进入下一个六位输入
        index += 6
```

```
    assert len(resultStr) == 32
    return resultStr
```

举例：

S-盒计算过程

S-盒8

行＼列	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
2	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
3	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

假设 S-盒 8 的输入(即异或函数的第 43~18 位)为 110011。

第 1 位和最后一位组合形成了 11(二进制)，对应 S-盒 8 的第 3 行。中间的 4 位组成形成 1001(二进制),对应 S-盒 8 的第 9 列。所以对应 S-盒 8 第 3 行第 9 列值是 12。则 S-盒输出是 1100(二进制)。

```
#P 盒置换
# P 置换的置换矩阵
P_MATRIX = [16, 7, 20, 21, 29, 12, 28, 17, 1, 15, 23, 26, 5, 18, 31, 10,
                2, 8, 24, 14, 32, 27, 3, 9, 19, 13, 30, 6, 22, 11, 4, 25]
```

S-盒代替运算，每一盒得到 4 位，8 盒共得到 32 位输出。这 32 位输出作为 P 盒置换的输入块。

P 盒置换将每一位输入位映射到输出位。任何一位都不能被映射两次，也不能被略去。

经过 P-盒置换的结果与最初 64 位分组的左半部分异或，然后左右两部分交换，开始下一轮迭代。

```
def P(Li, S_sub_str, oldRn):
    # P  盒置换
    tmp = ""
    for i in P_MATRIX:
        tmp += S_sub_str[i - 1]
    # P 盒置换的结果与最初的 64 位分组左半部分 L0 异或  Ri=Li-1⊕F(Ri-1, Ki)
    LnNew = int(tmp, base=2) ^ int(Li, base=2)
    LnNew = bin(LnNew)[2:]#返回二进制
    while len(LnNew) < 32:
        LnNew = "0" + LnNew#补满 32 位
    assert len(LnNew) == 32
    # 左、右半部分交换，接着开始另一轮
    (Li, Ri) = (oldRn, LnNew)

    return (Li, Ri)
```

将初始置换进行 16 次的迭代，即进行 16 层的加密变换，这个运算过程我们暂时称为函数 f。

```
    # 得到 L16 和 R16，将此作为输入块，进行逆置换得到最终的密文输出块。逆置换是初始置换的逆运算。
    def IP_inverse(L16, R16):
        tmp = L16 + R16
        resultStr = ""
        for i in IP_INVERSE_MATRIX:
            resultStr += tmp[i - 1]
        assert len(resultStr) == 64
        return resultStr


    # DES 算法实现 flag 是标志位 当为-1 时， 是 DES 解密， flag 默认为 0
    def DES (plaintext, secretKey, flag = "0"):
        # 初始字段
        # IP 置换
        InitKeyCode = IP(plaintext)
        # 产生子密钥 集合
        roundKeyList = createRoundSecretkey(secretKey)
        # 获得 Li 和 Ri
        Li = InitKeyCode[0:32]
        Ri = InitKeyCode[32:]
    # 如果是解密的过程 把子密钥数字逆过来 就变成解密过程了
        if (flag == "-1") :
            roundKeyList = roundKeyList[::-1]

        for subkey in roundKeyList:
            while len(Ri) < 32:
                Ri = "0" + Ri
            while len(Li) < 32:
                Li = "0" + Li

            # 对右边进行 E-扩展
            Rn_expand = E_expand(Ri)
            # 压缩后的密钥与扩展分组异或以后得到 48 位的数据，将这个数据送入 S 盒
            S_Input = int(Rn_expand, base=2) ^ int(subkey, base=2)

            # 进行 S 盒替代
            S_sub_str = S_sub(S_Input)

            #P 盒置换 并且
            # 左、右半部分交换，接着开始另一轮
            (Li, Ri) = P(Li, S_sub_str, Ri)
```

```
            #进行下一轮轮置换

        # 最后一轮之后  左、右两半部分并未进行交换
        # 而是两部分合并形成一个分组做为末置换的输入。
        # 所以要重新置换 一次

        (Li, Ri) = (Ri, Li)
        # 末置换得到密文
        re_text = IP_inverse(Li, Ri)

        return re_text

    #计算不同位数的函数，利用异或
    def count(a, b):
        #保证 a，b 两个数是一样长的
        y = '{0:b}'.format(int(a,2) ^ int(b,2))
        #计算其中的 0，就是
        print ("两个密文块间不同数据位的数量为:",64-y.count("0"))

    #实验要求的密文和明文测试
    if __name__ == "__main__":
        print("问题一")
        #明文和密钥。DES 的明文长为 64 位，密钥长为 56 位（其中 8 位为校验位）。
        secretKey                                                                =
"0000001010010110010010001100010000111000001100000011100001100100"
        plaintext1                                                               =
"0000000000000000000000000000000000000000000000000000000000000000"
        plaintext2                                                               =
"1000000000000000000000000000000000000000000000000000000000000000"


    #     print("明文的 2 进制形式:              " + plaintext)
        ciphertext11 = DES(plaintext1, secretKey)
        ciphertext12 = DES(plaintext2, secretKey)

        print("加密后的密文 1:                  " + ciphertext11)
        print("加密后的密文 2:                  " + ciphertext12)
        count(ciphertext11,ciphertext12)
    #       decode_ciphertext = DES(ciphertext, secretKey, "-1")
    #       print("解密:                    " + decode_ciphertext)

        print("问题二")
        secretKey1                                                               =
"1110001011110110110111100011000000111010000010000110001011011100"
```

```
        secretKey2                                                    =
"01100010111101101101111000110000001110100000100001100010111011100"
        plaintext                                                    =
"01101000100001010010111101111010000100110111011011101010111010010 0"

        ciphertext21 = DES(plaintext, secretKey1)
        #打印加密后的密文
        print("加密后的密文 1:                  " + ciphertext21)
        ciphertext22 = DES(plaintext, secretKey2)
        print("加密后的密文 2:                  " + ciphertext22)

        count(ciphertext21,ciphertext22)
```

```
    secretKey = "0000001010010110010010001100010000111000001100000011100001100100"
    plaintext1 = "0000000000000000000000000000000000000000000000000000000000000000"
    plaintext2 = "1000000000000000000000000000000000000000000000000000000000000000"


#   print("明文的2进制形式:          " + plaintext)
    ciphertext11 = DES(plaintext1, secretKey)
    ciphertext12 = DES(plaintext2, secretKey)

    print("加密后的密文1:              " + ciphertext11)
    print("加密后的密文2:              " + ciphertext12)
    count(ciphertext11,ciphertext12)
#     decode_ciphertext = DES(ciphertext, secretKey, "-1")
#     print("解密:              " + decode_ciphertext)

    print("问题二")
    secretKey1 = "1110001011110110110111100011000000111010000010000110001011011100"
    secretKey2 = "0110001011110110110111100011000000111010000010000110001011011100"
    plaintext = "0110100010000101001011110111101000010011011101101110101011101001 00"

    ciphertext21 = DES(plaintext, secretKey1)
    #打印加密后的密文
    print("加密后的密文1:              " + ciphertext21)
    ciphertext22 = DES(plaintext, secretKey2)
    print("加密后的密文2:              " + ciphertext22)

    count(ciphertext21,ciphertext22)
```

```
问题一
加密后的密文1:              1100010011010111001011001001110111101110110111100101111010001011
加密后的密文2:              0010110010010111011000000111011010100111000001011000110101000100
两个密文块间不同数据位的数量为: 34
问题二
加密后的密文1:              0101101010001100101100001111000000101000111111011111110100011111
加密后的密文2:              1001011100011011001010000000010111110000010000100010011000101000
两个密文块间不同数据位的数量为: 41
```

雪崩效益:雪崩效应就是一种不稳定的平衡状态也是加密算法的一种特征，它指明文或密钥的少量变化会引起密文的很大变化，雪崩效应是指少量消息位的变化会引起信息摘要的许多位变化。

如问题一种，一位明文的输入差异就会导致最后输出的数据位差别 34 个，所以 DES 是具有雪崩效益的。

64位密钥:

00000010 10010110 01001000 11000100 00111000 00110000 00111000 01100100

64位明文块1:

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

64位明文块2（与明文块1仅有一位的不同）：

10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

进一步分析，将每一步的输出进行排列(已 P 置换的结果进行对比)

雪崩分析-IP置换 00000000000000000000000000000000000000000000000000000000000000000000000000000000	雪崩分析-IP置换 00000000000000000000000000000000000000000000000001000000000000000000000000000000
雪崩分析-P置换矩阵 ('00000000000000000000000000000000', '10000101011111110001010100100011')	雪崩分析-P置换矩阵 ('00000001000000000000000000000000', '11000001011111110010101101010011')
雪崩分析-P置换矩阵 ('10000101011111110001010100100011', '11010111001011110000110101111011')	雪崩分析-P置换矩阵 ('11000001011111110010101101010011', '00011111110100010010000111011001')
雪崩分析-P置换矩阵 ('11010111001011110000110101111011', '11000111011011100110110010110001')	雪崩分析-P置换矩阵 ('00011111110100010010000111011001', '01001010100101001101011111101001')
雪崩分析-P置换矩阵 ('11000111011011100110110010110001', '01001100101100000111011110001010')	雪崩分析-P置换矩阵 ('01001010100101001101011111101001', '10100000000111011010101000101111')
雪崩分析-P置换矩阵 ('01001100101100000111011110001010', '01110010001010111011110010000001')	雪崩分析-P置换矩阵 ('10100000000111011010101000101111', '11111100011000100111111010010110')
雪崩分析-P置换矩阵 ('01110010001010111011110010000001', '01011001100001010111001001111011')	雪崩分析-P置换矩阵 ('11111100011000100111111010010110', '11000010000111001000111001010001')
雪崩分析-P置换矩阵 ('01011001100001010111001001111011', '10000010011001111010111010011100')	雪崩分析-P置换矩阵 ('11000010000111001000111001010001', '10110100010111011001111010110000')
雪崩分析-P置换矩阵 ('10000010011001111010111010011100', '11100111110111011101101110010100')	雪崩分析-P置换矩阵 ('10110100010111011001111010110000', '00100110100100100010100000010101')
雪崩分析-P置换矩阵 ('11100111110111011101101110010100', '01110001100100000000111100010001')	雪崩分析-P置换矩阵 ('00100110100100100010100000010101', '00001111011010111011001010101110')
雪崩分析-P置换矩阵 ('01110001100100000000111100010001', '00001010101011010011001111100100')	雪崩分析-P置换矩阵 ('00001111011010111011001010101110', '11001100100001100000100110011111')
雪崩分析-P置换矩阵 ('00001010101011010011001111100100', '01010001011000011011001010000001')	雪崩分析-P置换矩阵 ('11001100100001100000100110011111', '11110000000000110011101111001000')
雪崩分析-P置换矩阵 ('01010001011000011011001010000001', '01111101110111010100101010011110')	雪崩分析-P置换矩阵 ('11110000000000110011101111001000', '00101000111111100111010111111010')
雪崩分析-P置换矩阵 ('01111101110111010100101010011110',	雪崩分析-P置换矩阵 ('00101000111111100111010111111010',

'01110101000101110011100100101000') 雪崩分析-P置换矩阵 ('01110101000101110011100100101000', '10011101101000000001111001001110') 雪崩分析-P置换矩阵 ('10011101101000000001111001001110', '10111011000101001111110011110010') 雪崩分析-P置换矩阵 ('10111011000101001111110011110010', '01110011011010100111111110001010')	'11011001000000101011101011100100') 雪崩分析-P置换矩阵 ('11011001000000101011101011100100', '00111111001001101101011100001111') 雪崩分析-P置换矩阵 ('00111111001001101101011100001111', '01010010000111010100000100011010') 雪崩分析-P置换矩阵 ('01010010000111010100000100011010', '10001100000010101111101101110010')