

哈尔滨工业大学（深圳）

《密码学基础》实验报告

RSA 密码算法实验

学 院: 计算机科学与技术
姓 名: 吴梓滔
学 号: 220110430
专 业: 计算机科学与技术
日 期: 2024-10-20

一、实验步骤

请说明你的字符分组方式，以及关键的算法例如扩展欧几里德，素数检测，快速幂等函数的实现。

1. 字符分组方式

将每个字符对应成 2 位十进制数字进行两两拼接，得到 4 位十进制数字。

1) 字符的对应规则：基于 ASCII 码。在 ASCII 中，可显示字符的值为 32-126，为了将字符对应成 2 位十进制数，取字符的 ASCII 值减去 32，得到 0-94 的值。对于 0-9，在前面填充 0。基于该规则，可以将所有可显示字符转换成 00-94 之间的值。示意代码如下：

```
# 将 char 取 ascii 值减去 32，以 2 位十进制的形式存入列表
int_list.append(f"{ord(char) - 32:02d}")
```

2) 分组与填充：字符串输入通过对应规则转换为一个数字列表，直接进行两两分组拼接成 4 位十进制数字。若字符个数为奇数，则需要进行填充，填充一个空格，对应的数字为 '00'。示意代码如下：

```
# 将数字两两拼接成四位数字
four_digit_list = []
for i in range(0, len(int_list), 2):
    four_digit = int_list[i] + int_list[i + 1] # 两两进行拼接
    four_digit_list.append(int(four_digit)) # 转换为整数，用于后续计算
```

3) 其他细节：字符串进行填充后，解密时需要删去末尾多余的空格。实现的方式是在读取字符串进行分组时，先记录原始字符串的长度。在解密后，根据原始字符串的长度判断加密时是否填充了一个空格。若有填充一个空格，就在解密完成后删去。

2. 扩展欧几里得算法

给定两个整数 a 和 b , 返回满足 $ax + by = \gcd(a, b)$ 的整数 x 和 y 及 $\gcd(a, b)$.

为了实现该算法, 采用递归的思想。

```
def extended_gcd(a, b):
    if b == 0:
        return a, 1, 0
    else:
        gcd, x1, y1 = extended_gcd(b, a % b)
        x = y1
        y = x1 - (a // b) * y1
        return gcd, x, y
```

使用递归的思想, 规定返回值为三个值 \gcd, x, y 。基本情况为当 b 为 0 时, \gcd 为 a , $x = 1$, $y = 0$ 。

对于其他情况, 先使用辗转相除法, 计算 $\text{extended_gcd}(b, a \% b)$ 。假设 $\text{extended_gcd}(b, a \% b)$ 已经成功返回 $\gcd, x1, y1$ 的值, 那么原问题的解 x 即为 $y1$, y 为 $x1 - (a // b) * y1$ 。

3. 素数的检测算法

素数检测采用概率性的素性检测法, Miller-Rabin 算法。由于 Miller-Rabin 算法具有速度快、概率性的特点, 对输入数进行 5 次的 Miller-Rabin 测试, 如果都通过, 就确信这个数是素数。

单次 Miller-Rabin 测试的实现:

```
def _miller_rabin_test(d: int, n: int) -> bool:
    .....
```

使用 Miller-Rabin 算法测试 n 是否为素数的单轮测试。

参数:

d : $n-1$ 的奇数部分

n : 要测试的数

返回:

如果 n 可能是素数, 返回 True; 否则返回 False。

.....

$a = 2 + \text{random.randint}(1, n - 4)$

$x = \text{quick_pow}(a, d, n)$

if $x == 1$ or $x == n - 1$:

 return True

while $d \neq n - 1$:

$x = (x * x) \% n$

$d *= 2$

 if $x == 1$:

 return False

 if $x == n - 1$:

 return True

return False

随机生成正整数 a , d 是奇数, 计算 $a^d \bmod n$, 存为变量 x 。根据素数性质, 若 x 为 1, 或 $a^{2^{k-1}d} \bmod n = n - 1 (k \leq d)$ 至少有一个 k 成立。因此检查 $x=1$, 或者将 x 每次进行平方, 检查是否 $x=n-1$, 若成立, 则返回 True。

调用 Miller-Rabin 算法进行素数检验:

在外层测试函数中, 先确保 n 是大奇数的前提下, 计算出 $n-1$ 的最大奇数部分 d , 即可调用 `miller_rabin_test(d, n)` 若干次, 检测 n 是否为素数。示意代码如下:

$d = n - 1$

while $d \% 2 == 0$:

$d //= 2$

for $_$ in range(k):

 if not `_miller_rabin_test(d, n)`:

 return False

4. 快速取模幂

在 RSA 的加密和解密以及上述 Miller-Rabin 算法中都需要用到取模的幂函数。实现快速幂函数，以提高效率，避免溢出。

快速幂的原理是把指数用二进制的形式表示，同时可以将原本的指数式拆成若干个指数为 2 的次数的式子相乘,例如底数 m ，指数 $e=13$ ，可表示为

$$m^{1101} = m^1 \cdot m^4 \cdot m^8$$

因此可以每次将指数向右移位，同时通过平方的形式更新，根据指数 e 的最低位是否为 1，决定当前底数是否乘入。实现代码如下：

```
def quick_pow(m,e,n): # 快速取幂模
    ans = 1
    while e:
        if e & 1:
            ans = (ans * m) % n
        m = m * m % n
        e >>= 1
    return ans
```

二、实验结果与分析

程序正确运行的结果截图，包括 $p, q, n, e, d, \phi(n)$ 这些参数的值，输出加密解密时间作为时长参考，对比上次的 AES 密码算法效率上有什么不同。

```

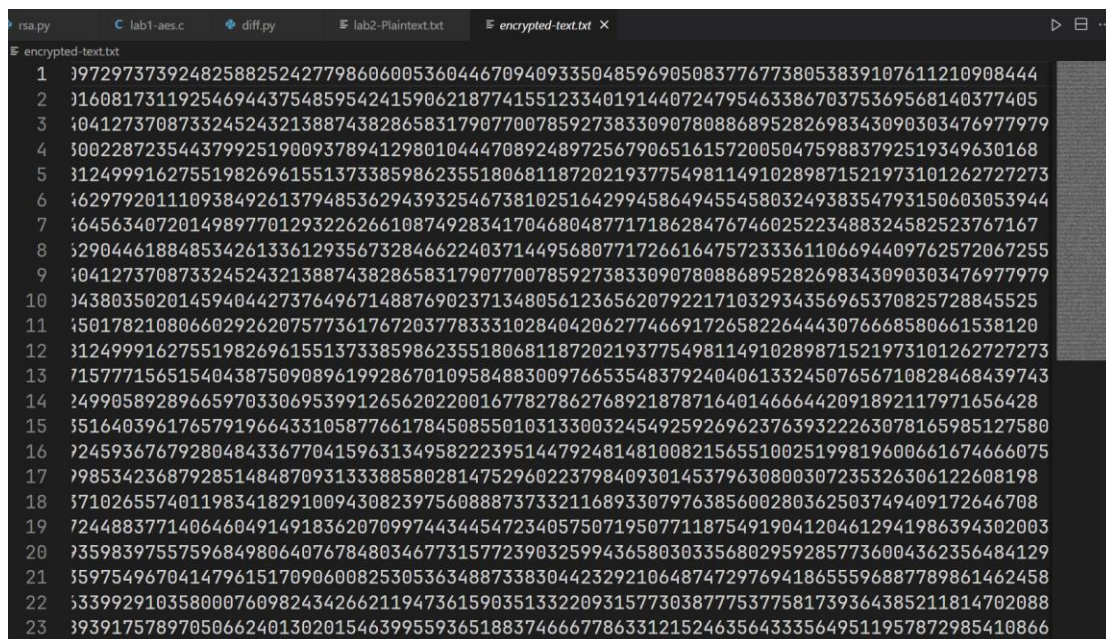
PS C:\Users\83923\Desktop\CryLab> & C:/Users/83923/AppData/Local/Microsoft/WindowsApps/python3.8.exe c:/Users/83923/Desktop/CryLab/rsa.py
生成的素数 p: 1657872583514131045703058296936978377112640981190749242994363269086776621912803649497800896092823
818615207536066218970458795385456368078115832129450646435772528484182672520260927272770439380205317702562581496
07996789116607245813151509446400596631502430479229378458256730559298734130956056583239503223619474001
生成的素数 q: 9931533041490727819835599721447151404402228298785196366392830363356234797291539695219680196303743
476281916567318799938863808321335457811147079132573059977066061727351951916101211155398665605415529832083412863
5941050058862657964289183335301139218234906701206420985562233303266705808031663383400814993091405881
模数 n: 1646521634175218856927690730349611812206769784448012374542127860126809256292681410466561918838458803409
932069715010839676267114728066484946687646467393577627477959603967829263219222618747450498633865129792081486217
187313216217705454931976860799677900539486091873109664650194534130459841086851017829404428878663541235480906162
561435225114971094763825422941352828263015715330697585492749041019267233344702809594190366931025421141710048933
51088268824372156236119084802519423828268452171024684163693855882252798189813813013110052530701825708438532244
1293277446456831080866594804012010414448985727578915531775022817999881
欧拉函数  $\phi(n)$ : 164652163417521885692769073034961181220676978444801237454212786012680925629268141046656191883845
880340993206971501083967626711472806648494668764646739357762747795960396782926321922261874745049863386512979208
148621718731321621770545493197686079967790053948609187310966465019453413045984108685101782940442887863703097660
427412428456984242415415958829678483066013946667925227647358447632946482906954418747082793175637500304443149825
82867576117496765189717291565947500111728546495913334054080106063444841354594048058598861932613498371144796436
92540739557427709119650645067150985048147849009046739858948891456806107120000
公钥指数 e: 65537
私钥指数 d: 179909691049769477309904226925760565592149143635384845417034310486718664026609267747242777374646436
230198568003253011625825851176704519564570093784472975714334950771686430952803955829117147459003572153050049521
992176607869468162850950880972583220065699730837601711068687307064084535803012389034265779103639164438085714378
244411581227593563451741944142639613611528157989007001042745853415591817858704647429678186480940488596883577636
92984509116123033346849724201669743943301428560033430469450771370143421311316988514343088552308136742281045031
4848649934232952466824378990008757339926251640809071668416493311999833473

明文整型数组: ['1816', '1618', '0033', '1445', '1400', '5253', '5041', '4639', '0033', '5533', '5036', '1400',
'5051', '3312', '0065', '7800', '6567', '8279', '7889', '7700', '7079', '8200', '5073', '8669', '8384', '1200',
'5172', '6577', '7382', '0065', '7868', '0033', '6876', '6977', '6578', '1200', '8583', '6983', '0065', '7671',
'7982', '7384', '7277', '7367', '0078', '8577', '6669', '8200', '8472', '6979', '8289', '0084', '7900', '8082',
'7986', '7368', '6900', '6578', '0069', '7070', '7367', '7369', '7884', '0082', '6965', '7673', '9065', '84',
'73', '7978', '0079', '7800', '6500', '8085', '6676', '7367', '1375', '6989', '8084', '7983', '8',
'983', '8469', '7712', '0065', '0067', '7978', '6769', '8084', '0070', '7382', '8384', '0069', '7886', '7383',
'7379', '7869', '6800', '8472', '6979', '8269', '8473', '6765', '7676', '8900', '6689', '0055', '7273', '8470',
'7369', '7668', '0036', '7370', '7073', '6912', '0045', '6582', '8473', '7800', '4069', '7676', '7765', '7800',
'6578', '6800', '5065', '7680', '7200', '4569', '8275', '7669', '1400', '5051', '3300', '7383', '0078', '7987',
'0084', '7269', '0077', '7983', '8400', '8773', '6869', '7689', '0085', '8369', '6800', '6978', '6782', '8980',
'8473', '7978', '0077', '6984', '7279', '6812', '0087', '7384', '7200', '6580', '8076', '7367', '6584', '737',
'9', '7883', '0084', '7282', '7985', '7172', '7985', '8400', '8472', '6900', '4178', '8469', '8278', '6984', '80',
'84', '7900', '8369', '6785', '8269', '0079', '7813', '7673', '7869', '0084', '8265', '7883', '6567', '8473', '7',
'978', '8314', '0041', '8400', '7265', '8300', '6576', '8379', '0073', '7883', '8073', '8269', '6800', '6682',
'6965', '7584', '7282', '7985', '7172', '0087', '7982', '7500', '7378', '0066', '7984', '7200', '8472', '6979',
'8269', '8473', '6765', '7600', '6779', '7780', '8584', '6982', '0083', '6773', '6978', '6769', '0065', '7868',
'0077', '6584', '7269', '7765', '8473', '6783', '1400']
该密文已经被同时写入到文件encrypted-text.txt中
加密时间: 0.030217秒

已读取加密文件encrypted-text.txt中的密文, 开始解密
解密时间: 6.147177秒
解密后的明文整型数组: [1816, 1618, 33, 1445, 1400, 5253, 5041, 4639, 33, 5533, 5036, 1400, 5051, 3312, 65, 7800,
6567, 8279, 7889, 7700, 7079, 8200, 5073, 8669, 8384, 1200, 5172, 6577, 7382, 65, 7868, 33, 6876, 6977, 6578,
1200, 8583, 6983, 65, 7671, 7982, 7384, 7277, 7367, 78, 8577, 6669, 8200, 8472, 6979, 8289, 84, 7900, 8082, 79
86, 7368, 6900, 6578, 69, 7070, 7367, 7369, 78848470, 7369, 7668, 36, 7370, 7073, 6912, 45, 6582, 8473, 7800, 4
069, 7676, 7765, 7800, 6578, 6800, 5065, 7680, 7200, 4569, 8275, 7669, 1400, 5051, 3300, 7383, 78, 7987, 84, 72
8470, 7369, 7668, 36, 7370, 7073, 6912, 45, 6582, 8473, 7800, 4069, 7676, 7765, 7800, 6578, 6800, 5065, 7680, 7
200, 4569, 8275, 7669, 1400, 5051, 3300, 7383, 78, 7987, 84, 7269, 77, 7983, 8400, 8773, 6869, 7689, 85, 8369,
6800, 6978, 6782, 8980, 8473, 7978, 77, 6984, 7279, 6812, 87, 7384, 7200, 6580, 8076, 7367, 6584, 7379, 7883, 8
4, 7282, 7985, 7172, 7985, 8400, 8472, 6900, 4178, 8469, 8278, 6984, 84, 7900, 8369, 6785, 8269, 79, 7813, 7673
, 7869, 84, 8265, 7883, 6567, 8473, 7978, 8314, 41, 8400, 7265, 8300, 6576, 8379, 73, 7883, 8073, 8269, 6800, 6
682, 6965, 7584, 7282, 7985, 7172, 87, 7982, 7500, 7378, 66, 7984, 7200, 8472, 6979, 8269, 8473, 6765, 7600, 67
79, 7780, 8584, 6982, 83, 6773, 6978, 6769, 65, 7868, 77, 6584, 7269, 7765, 8473, 6783, 1400]
解密结果已写入sb.txt文件。
PS C:\Users\83923\Desktop\CryLab>

```

完整运行过程如上图所示。运行后, 先生成 p , q , n , e , d , $\phi(n)$ 六个值并输出。随后将字符串转化成的明文数组输出。此后自动开始加密, 完成后会输出加密时间以及密文写入文件的提示。密文为一个 145kB 的大文件, 如下图所示。随后自动开始解密过程, 完成后会将解密得到的明文写入文件, 并输出解密使用的时间。



```
rsa.py lab1-aes.c diff.py lab2-Plaintext.txt encrypted-text.txt x
encrypted-text.txt
1 397297373924825882524277986060053604467094093350485969050837767738053839107611210908444
2 316081731192546944375485954241590621877415512334019144072479546338670375369568140377405
3 4041273708733245243213887438286583179077007859273833090780886895282698343090303476977979
4 300228723544379925190093789412980104447089248972567906516157200504759883792519349630168
5 3124999162755198269615513733859862355180681187202193775498114910289871521973101262727273
6 4629792011109384926137948536294393254673810251642994586494554580324938354793150603053944
7 464563407201498977012932262661087492834170468048771718628476746025223488324582523767167
8 4290446188485342613361293567328466224037144956807717266164757233361106694409762572067255
9 4041273708733245243213887438286583179077007859273833090780886895282698343090303476977979
10 43803502014594044273764967148876902371348056123656207922171032934356965370825728845525
11 450178210806602926207577361767203778333102840420627746691726582264443076668580661538120
12 3124999162755198269615513733859862355180681187202193775498114910289871521973101262727273
13 7157771565154043875090896199286701095848830097665354837924040613324507656710828468439743
14 449905892896659703306953991265620220016778278627689218787164014666442091892117971656428
15 3516403961765791966433105877661784508550103133003245492592696237639322263078165985127580
16 7245936767928048433677041596313495822239514479248148100821565510025199819600661674666075
17 798534236879285148487093133388580281475296022379840930145379630800307235326306122608198
18 371026557401198341829100943082397560888737332116893307976385600280362503749409172646708
19 7244883771406460491491836207099744344547234057507195077118754919041204612941986394302003
20 7359839755759684980640767848034677315772390325994365803033568029592857736004362356484129
21 3597549670414796151709060082530536348873383044232921064874729769418655596887789861462458
22 339929103580007609824342662119473615903513322093157730387775377581739364385211814702088
23 3939175789705066240130201546399559365188374666778633121524635643335649511957872985410866
```

经过验证，加密后解密输出的明文，和原来的明文是一样的。使用了一个自动校验脚本 `diff.py` 进行检验，运行结果如图。

```
PS C:\Users\83923\Desktop\CryLab> python .\diff.py .\sb.txt .\lab2-Plaintext.txt
The files are identical.
```

加密用时 0.03 秒，而解密用时需要 6.147 秒。

三、总结

1、实验过程中遇到的问题有哪些？你是怎么解决的？

难以理解 RSA 算法的输入是数字，而加密过程的输入是明文字符串，在明文到数字还要分组之类的过程纠结。后面先实现了一个简单的从字符到数字的转换，直接每个字符取 ASCII 值得到数字进行加密。后面先用简单的字符转换，完善了 RSA 算法本身之后，再去把字符串的转换方式做了改进，变成两两分组，定成 4 位的数字进行加密。

2、关于本实验的意见或建议。