**密码学第一次实验报告**

# 埃拉斯托尼筛法

## 基本原理

通过标记质数，将质数的倍数筛去的方法来找到指定范围内的所有质数。要找出内的所有质数，首先需要找出内的所有质数，然后标记这些质数在内的所有倍数。质数的倍数一定是合数，因此这样就能找到内的所有质数。

另一种更加简化计算的方法是一边记下质数列表一边求质数。对内的所有数，若之前没被筛掉，就把它当作质数。对选中的数，将它从小到大与已知的质数列表乘一遍，但一旦乘到它的最小因子，就立即停止相乘。这样能保证每个合数只被筛掉一次，优化了速度。

## 伪代码

### 提前算出以内的质数

**def** eratosthenes\_precompute(upper\_bound:int) -> list:

生成大小为 upper\_bound+1 的全为**True**的dist\_table

initial\_prime\_list, prime\_list = [], []

dist\_table[0:1] = **False**

**for** i **in** range(2, +1)

**if** (dist\_table[i]):

initial\_prime\_list.append(i)

prime\_list.append(i)

dist\_table[[i\*i, ]内的所有i的倍数] = **False**

**for** i **in** range(+1, upper\_bound+1):

**if** (i 能整除initial\_prime\_list中某个数):

dist\_table[i] = **False**

**if** (dist\_table[i]):

prime\_list.append(i)

**return** prime\_list

### 一边算出以内的质数，一边做标记

**def** eratosthenes(upper\_bound:int) -> list:

生成大小为 upper\_bound+1 的布尔类型的dist\_table，初始化为**True**

prime\_list = []

dist\_table[0:1] = **False**

**for** i **in** range(2, +1)

**if** (dist\_table[i]):

initial\_prime\_list.append(i)

prime\_list.append(i)

dist\_table[[i\*i, upper\_bound]内的所有i的倍数] = **False**

**for** i **in** range(+1, upper\_bound+1):

**if** (dist\_table[i]):

prime\_list.append(i)

**return** prime\_list

### 每个数只被其最小质因子筛去

**def** euler(upper\_bound:int) -> list:

生成大小为 upper\_bound+1 的布尔类型的dist\_table，初始化为**True**

prime\_list = []

dist\_table[0:1] = **False**

**for** i **in** range(2, upper\_bound):

**if** (dist\_table[i]):

prime\_list.append(i)

dist\_table[i与prime\_list中所有小于其最小质因子且不超过upper\_bound的质数之积] = **False**

**return** prime\_list

## 时空复杂度分析和算法优化

事实上提前算出以内的质数的方法，效率上并不好。这是因为对中的数是否是质数的判断，消耗了时间复杂度。由于中平均有个质数，这使得算法时间复杂度至少为。因此，一边算出以内的质数，一边做标记的方法更好，因为中的数有一部分已经提前被筛掉，而且可以跳过了。

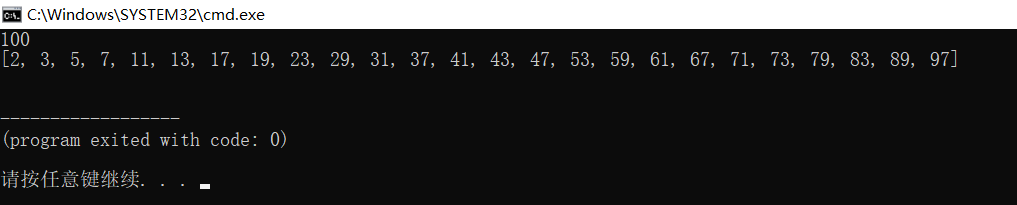
欧拉筛法实际上更加高效，因为能保证中每个质数只加入一次指数列表，每个合数只被筛掉一次，而且是通过其最小质因子和另一个数相乘筛掉的。这样欧拉筛法的时间复杂度为。

三种筛法的空间复杂度均为，因为都需要空间来保存中所有的数是否是质数的状态信息。保存质数列表需要的空间为级别，可以不计。

三种筛法的时间比较见“算法测试”一节。

## 测试

采用进行测试。三种筛法得到的结果均为：

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

三种筛法的时间比较如下。（单位：s）

|  |  |  |  |
| --- | --- | --- | --- |
| 数据规模 | 埃拉斯托尼筛法 | 埃拉斯托尼筛法（提前算出以内的质数） | 欧拉筛法 |
| 10^2 | 0.000013 | 0.000020 | 0.000012 |
| 10^3 | 0.000104 | 0.000213 | 0.000132 |
| 10^4 | 0.001416 | 0.002885 | 0.001350 |
| 10^5 | 0.015806 | 0.043884 | 0.015310 |
| 10^6 | 0.169105 | 0.855118 | 0.176000 |
| 10^7 | 2.131774 | 时间太长 | 2.020350 |
| 10^8 | 25.869392 | 时间太长 | 20.690915 |

可以看出，两种埃拉斯托尼筛法增长率实际上是随着的增大略有增加的。但是欧拉筛法增长率基本上随着的增大保持恒定。这就很好地说明了欧拉筛法的线性性质。同时，欧拉筛法确实在相同数据规模下，运行时间比两种埃拉斯托尼筛法快。

三种筛法的内存占用比较如下。（单位：MB）

|  |  |  |  |
| --- | --- | --- | --- |
| 数据规模 | 埃拉斯托尼筛法 | 埃拉斯托尼筛法（提前算出以内的质数） | 欧拉筛法 |
| 10^2 | 0.001496 | 0.001641 | 0.001436 |
| 10^3 | 0.013512 | 0.013696 | 0.013425 |
| 10^4 | 0.125150 | 0.125401 | 0.125063 |
| 10^5 | 1.115685 | 1.116296 | 1.115598 |
| 10^6 | 11.010888 | 11.012296 | 11.010801 |
| 10^7 | 100.674438 | 时间太长 | 100.674370 |

可以看出，所有的筛法内存占用都大约与数据规模成正比，但略大一点。这是因为要存储质数表的缘故。这与理论分析相符。

# 欧几里得算法

## 算法原理

由欧几里得除法原理，。由最大公因数性质，。令，且为和做欧几里得除法后的余数，为和做欧几里得除法后的商。由于绝对值随增加不断减小，算法最终一定能在有限步内终止，最终一定。

从欧几里得除法原理可以推导出，，其中。这样就可以实现扩展欧几里得算法和广义欧几里得算法。

根据扩展欧几里得算法或者广义欧几里得算法，可以求一个数的逆元。根据欧几里得除法原理，。因此。若，则可求出在模意义下的逆元。

## 伪代码

### 求最大公因数

**def** gcd(a:int, b:int) -> int:

调整a和b使得其均为非负且a > b

**while** (b != 0):

a = a % b

a, b = b, a

**return a**

### 扩展欧几里得算法

**def** gcd\_extended(a:int, b:int) -> list:

last\_last\_result = (0, a, 1, 0)

last\_result = (0, b, 0, 1)

result = (None, None, None, None)

log\_list = [last\_last\_result, last\_result]

**while** **True**:

result = (相应的商, 相应的余数, last\_last\_result[2] – 相应的商 \* last\_last\_result[2],

last\_last\_result[3] – 相应的商 \* last\_last\_result[3])

log\_list.append(result)

**if** (余数 == 0):

**break**

循环移动结果

**return** log\_list

### 广义欧几里得算法

**def** gcd\_generalized(a:int, b:int) -> tuple:

q = [0, 0]

r = [a, b]

**while** (r[-1] != 0):

q.append(r[-2] // r[-1])

r.append(r[-2] % r[-1])

gcd = r[-2]

根据和求出s

根据和求出t

**return** (gcd, s, t)

### 求模逆元

**def** get\_modular\_inverse(x:int, modulus:int) -> int:

gcd, s, t = gcd\_generalized(x, modulus)

**if** (gcd != 1):

返回错误

**return** (s % modulus)

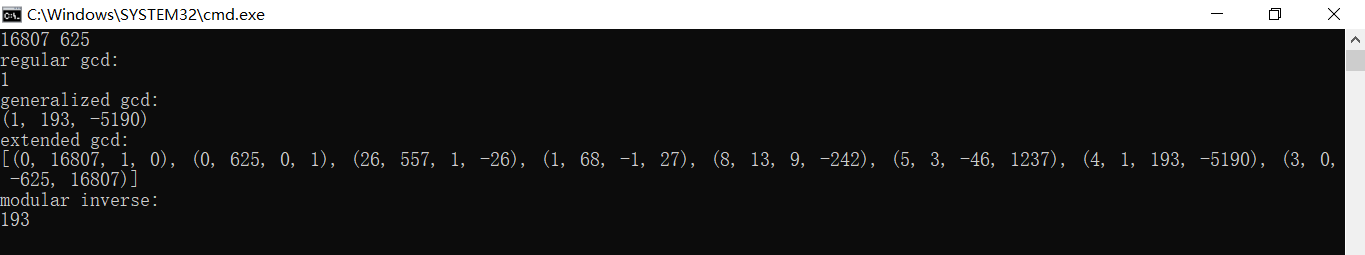
## 时空复杂度分析和两种欧几里得算法的比较

首先证明。若，则。若，则（由辗转相除法原理），同样有。这样做两次带余除法可将余数缩小一半。因此要得到（这里不妨假设），所要做的带余除法的次数不会超过。因此辗转相除法所需的时间复杂度为。易知这一结论对扩展欧几里得算法和广义欧几里得算法都成立。

对于扩展欧几里得算法和广义欧几里得算法，需要把每一步都记录下来，每一步花费的空间复杂度为，因此空间复杂度为。但扩展欧几里得算法多记录了每一步的和，因此二者空间复杂度实际上应该差一个常数。对于求逆元算法，由于它是简单包裹了广义欧几里得算法，而本身除了调用广义欧几里得算法的部分空间和时间复杂度均为，因此它的空间和时间复杂度与广义欧几里得算法相同。

## 测试

首先来验证扩展欧几里得算法的正确性。



令，得到以下表格。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| -2 | 0 | 16807 | 1 | 0 |
| -1 | 0 | 625 | 0 | 1 |
| 0 | 26 | 557 | 1 | -26 |
| 1 | 1 | 68 | -1 | 27 |
| 2 | 8 | 13 | 9 | -242 |
| 3 | 5 | 3 | -46 | 1237 |
| 4 | 4 | 1 | 193 | -5190 |
| 5 | 3 | 0 | -625 | 16807 |

用扩展欧几里得算法和欧几里得除法得到的结果与表格中的数据相符，经过手工验算得到的结果也相符。

显然互质，因此能求出在模意义下的逆元为193，经过验算是正确的。

两种欧几里得算法的时间比较如下。（模数为7^100，单位：s）

|  |  |  |
| --- | --- | --- |
| 底数 | 广义欧几里得算法 | 扩展欧几里得算法 |
| 10^1000 | 0.000719 | 0.000645 |
| 10^1300 | 0.001067 | 0.000762 |
| 10^1600 | 0.000900 | 0.000868 |
| 10^1900 | 0.000795 | 0.000831 |
| 10^2200 | 0.000845 | 0.000848 |
| 10^2500 | 0.000785 | 0.001084 |
| 10^2800 | 0.001042 | 0.001431 |

两种欧几里得算法的空间比较如下。（模数为7^100，单位：MB）

|  |  |  |
| --- | --- | --- |
| 底数 | 广义欧几里得算法 | 扩展欧几里得算法 |
| 10^1000 | 0.012414 | 0.102704 |
| 10^1300 | 0.012292 | 0.113317 |
| 10^1600 | 0.013169 | 0.138002 |
| 10^1900 | 0.014051 | 0.161260 |
| 10^2200 | 0.014322 | 0.172418 |
| 10^2500 | 0.015458 | 0.204229 |
| 10^2800 | 0.015908 | 0.225091 |

可以看出，随着底数的位数线性增长，两种算法需要的时间和空间都线性增长，但是扩展欧几里得算法需要的时间大致与广义欧几里得算法相同，空间大致是广义欧几里得算法的常数倍。这与理论分析相符。

# 模幂算法

## 原理

模幂算法是为了求解的值。

常规模幂算法是按照指数递增顺序一遍一遍做乘法，再进行运算。也就是逐步计算。

平方乘模幂算法是利用的二进制表示，以及幂运算的性质得到公式，进而只做次模平方运算和模乘运算。

## 伪代码

### 常规模幂算法

**def** generic\_modular\_pow(x:int, power:int, modulus:int) -> int:

**if** (modulus <= 0):

返回错误

result = 1

重复modulus遍模乘运算

**return** result

### 模平方快速幂算法

**def** square\_modular\_pow(x:int, power:int, modulus:int) -> int:

**if** (modulus <= 0):

返回错误

result = 1

**for** i **in** modolus 从高到低的每一位:

result = (result \* result) % modulus

**if** (i == 1):

result = (result \* x) % modulus

**return** result

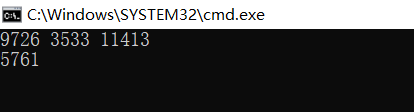
## 两种模幂算法的对比分析

对常规模幂算法，由于要做次模幂运算，只需要一个变量保存中间结果，所以时间复杂度为，空间复杂度为。对模平方快速幂算法，只需要做次模幂运算，因此时间复杂度为。要把的每一位都保存下来，因此空间复杂度也为。

两种模幂算法的时间比较见“算法测试”一节。

## 测试

首先使用进行测试。



发现结果与预期相符。

两种模幂算法的时间比较如下。（底数为，模数为，单位：s）

|  |  |  |
| --- | --- | --- |
| 指数 | 模平方快速幂算法 | 常规模幂算法 |
| 10^1 | 0.000061 | 0.000133 |
| 10^2 | 0.000109 | 0.001489 |
| 10^3 | 0.000192 | 0.014323 |
| 10^4 | 0.000233 | 0.161286 |
| 10^5 | 0.000297 | 1.568101 |
| 10^6 | 0.000365 | 16.718450 |
| 10^7 | 0.000406 | 时间太长 |
| 10^100 | 0.006946 | 时间太长 |
| 10^200 | 0.015104 | 时间太长 |
| 10^300 | 0.023835 | 时间太长 |
| 10^400 | 0.032052 | 时间太长 |
| 10^500 | 0.037038 | 时间太长 |
| 10^600 | 0.048077 | 时间太长 |

两种模幂算法的内存占用比较如下。（单位：MB）

|  |  |  |
| --- | --- | --- |
| 指数 | 模平方快速幂算法 | 常规模幂算法 |
| 10^1 | 0.001378 | 0.001318 |
| 10^2 | 0.001413 | 0.001318 |
| 10^3 | 0.001474 | 0.001345 |
| 10^4 | 0.001474 | 0.001345 |
| 10^5 | 0.001542 | 0.001345 |
| 10^6 | 0.001542 | 0.001345 |
| 10^7 | 0.001542 | 时间太长 |
| 10^100 | 0.004052 | 时间太长 |
| 10^200 | 0.006032 | 时间太长 |
| 10^300 | 0.009896 | 时间太长 |
| 10^400 | 0.012277 | 时间太长 |
| 10^500 | 0.015290 | 时间太长 |
| 10^600 | 0.017083 | 时间太长 |

可以看出，随着底数的位数线性增长，模平方快速幂算法需要的时间和空间也是线性增长的；而常规模幂算法需要的时间倾向于指数增长，空间基本上不变。这与理论预测相符，显然模平方快速幂算法更为实用。

# 中国剩余定理算法

## 原理

根据中国剩余定理的通解公式求解即可。注意要先检查各是否互素。

## 伪代码

**def** crt(target:list, modulus:list) -> int:

对每个，

对每个，

对每个，为在模意义下的逆

利用求出

**return**

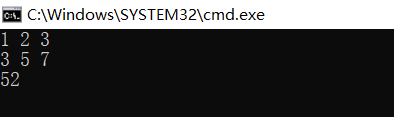
## 分析

设方程的个数为，各中最大值为。首先要做次乘法运算和次除法运算，之后还要做次求逆元运算，最后是次模乘运算和次模加法运算。总时间复杂度为。同样的，需要保存各、各、和。求逆元时，需要的辅助空间。因此总空间复杂度为。

## 测试

利用中国剩余定理算法求解方程

得到，经过验算结果正确。



# 总结

这次实验是密码学的第一次实验，我感觉很有体会。其实我在之前学过第一次实验中的算法，但是只是听说过，并没有真正走到实践这一步。在这次实验中实践了，其实是另一种体验和感受。真正实践的时候，还是需要考虑实践上的一些小问题的。

## 埃拉斯托尼筛法

实现埃拉斯托尼筛法其实并不难，我记得当时我们学习Python的时候，有一道题就必须要用这个算法才能通过。真正难的是针对筛法（其实也是针对以后的其它算法）写一套评测机制，这才是相对比较难的地方，在后面我会专门写出来。

用埃拉斯托尼筛法简单地求出以内的质数，如何估计时间复杂度，也是一个比较重要的点。时间复杂度是，但是严格证明比较复杂。

实现欧拉筛法就有一定挑战性了。欧拉筛法也是一种相对常见的筛法，而且性能优秀一些，时间复杂度为。欧拉筛法线性性的关键，就是保证每个合数只被筛一次，而且被筛的原因是它是其最小质因子和另一个数的乘积。实际上在数据规模小于时，需要的时间和空间与埃拉斯托尼筛法并没有什么区别，而且该算法需要的乘法次数较多，在乘法比较昂贵的体系结构里可能并没有埃拉斯托尼筛法有优势。

对埃拉斯托尼筛法，很显然的一个改进思路就是两个数两个数地去查找素数，直接跳过所有的大于2的偶数。其实还有一种办法，对大于10的所有数，只选除以10余数为1、3、7、9的。这样可以看成每次跳过的数的个数有变化。再拓展开，其实对于一个大小合适的合数，也可以这么干，只需要判断在它的某些剩余类里面的数是否为素数。这其实对算法的理论时间复杂度级别没有什么影响，但是能够让总运行时间减小一个常数。

埃拉斯托尼筛法的一个好处就是并行计算相对容易些。如果将内的数分成若干个区间，那么后面区间的素数判定不受前面区间的数的影响。但是前面计算内的素数，还是要用一次埃拉斯托尼筛法（或者其它筛法）。判定后面的数是否为素数，是要用内的素数逐个判定的，因此只有并行化程度较高时，才能体现出优势。

## 欧几里得算法

几个欧几里得算法相对确定，实现起来也相对容易。只需要跟着公式走，就基本上能实现正确。但是，它们也有一些坑需要避免。

对单纯的欧几里得除法来说，实现成迭代的更好，因为迭代式欧几里得除法空间复杂度一定是，而递归式欧几里得除法空间复杂度是。

对广义欧几里得除法和扩展欧几里得除法来说，广义欧几里得除法只需要保存三步，这样计算时需要的空间复杂度是，但保存计算结果所需要的空间复杂度是避免不了的。对扩展欧几里得除法，情况类似。

求模逆元的一个大坑是一定一定要判断互质。判断互质之后，还有一个小坑，是求其它数模1的逆元的情况。其实一个数模1并没有意义，所以判断完后，还需要判断模数是不是1。

欧几里得算法如果直接从算法层面分析，其实并不适合并行，因为它的各步之间依赖关系太强了。唯一能并行的部分是和的计算，但这还得等串行部分（和）计算完了才行。

应该可以通过改进欧几里得除法的余数取法来改进欧几里得算法。可以取最小绝对值余数，来使余数的绝对值下降得更快。但是缺点是算出来的最大公因数还是要取绝对值，这时要看的符号来决定和是否取相反数。

## 模幂算法

模幂算法在实践中基本上是指模平方快速幂算法，因为常规模幂算法根本不实用，尤其是面对公钥加密几百个十进制位大数的时候。

其实模幂算法判断某个数某位是不是1，可以用位运算（比特掩码）来解决。虽然这样时间复杂度和空间复杂度并没有什么变化，但是位运算在常见的体系结构中应该都比较快，利用数字电路也比提前算出指数的每一位更好实现。

模幂算法也可以提前把模数进行因子分解，然后通过欧拉定理直接模指数，这样更能简化计算。但是由于因子分解时间复杂度较大，这种算法只在已知因子分解时才特别有用。

模幂算法其实很适合并行，虽然并行化需要多余的计算。可以提前把各算出来，再选择需要的相加，来得到所需结果。这对加法相对昂贵但模乘运算相对便宜的体系结构很有利。这里可以稍微进行调度，让和，和，……一起算，这样能减少总完成时间。

## 中国剩余定理算法

中国剩余定理算法其实也是一个相对平稳的算法。但是最重要的一点是判断各是否互质。判断各是否互质，其实也是一个两两判断的算法。这里判断是否互质，是靠定理来判断的。所以时间复杂度为，其中是所有的最大值。

这里判断互质的算法并行性很好，一种简单的方法是可以把分成若干块来并行。这样做的缺点是每一块都要做一次相对较大的数的模乘运算，而不是对所有的只做一次。但是可以缓存，然后对每一块的第一个元素，都先求它与的公因数。还有一种方法是类似于归并排序，把分成大致等长的两块，每个中再分……一直到基本情况：求两个或者三个数的最大公因数。本来这种算法形式优美，但时间复杂度较大。但是如果缓存，那就快多了，因为。更进一步，对于这种算法，可以把每个小块放到不同的处理器上分别求最大公因数，因为除了这个不变数据，其它的小块间的运算都是独立的！

实际上，在原来的算法中，如果求最大公因数的某步求到了1，那之后就不用判断了，因为各肯定互质了。所以在并行计算的时候，如果，直接退出判断，并通过线程通信通知所有其它线程退出。

之后相乘、求逆元、做模加法各步，并行性都很强。尤其是求逆元，因为数据基本上是独立的。这里通过相乘求出可以通过一个一个地乘来实现，这对除法比较昂贵的体系结构相对有利。

由于各互素，可以通过扩展欧几里得算法求出在模意义下的乘法逆元。由扩展欧几里得算法原理，可以求出在模意义下的乘法逆元。由于，对求出的逆元对应相乘，再乘上对应的，就可以得出。这种算法其实计算上并不太实用，但是对需要求逆元的场合比较有利。

## 算法评估与优化

等到真正要求评估时间和空间复杂度的时候，我才能想起去看时间和空间复杂度的资料，来真正地去估计算法的时间和空间复杂度。其实对于时空复杂度分析，伪代码是重要的工具，因为伪代码提供了理想的计算机模型，而真正的实现通常多少会遮蔽这一点。

一般来说评估算法时间和空间效率的优劣是看时间和空间复杂度的，但是时空复杂度只是用大O符号表示，大O符号表示不出来的常数也很重要。对埃拉斯托尼筛法来说，如果真的能只去查找比2大的所有偶数，时间应该能节省一半。如果一边找素数一边筛，也能节省相当一部分时间。

算法的实现本身也能反过来影响算法的时空复杂度，但是一般是影响一个常数因子，因为算法的运算框架是定死了的。扩展欧几里得算法就是一个例子。虽然可以用只保存三步的办法在计算中达到的空间复杂度，但是历史记录还是得保存，总体上空间复杂度还是的。

估计算法的时空复杂度，本身就是有趣但比较难的数学问题。比如埃拉斯托尼筛法的复杂度就是这样。

## 系统设计与维护

事实上，密码学实验虽然显得不大正式，没有OJ，甚至连真正的评测机也没有，但它确实是最需要系统设计的领域，因为每个环节确实是环环相扣的。设计好了欧几里得算法，以后在公钥密码领域求乘法逆元就需要经常用。设计好评测算法，以后每次都需要经常用。虽然看起来不正式，但是密码学课程本身的依赖关系决定了密码学实验真的是环环相扣。

实际上，代码的鲁棒性和可维护性，在长远来看比效率更为重要。虽然不写注释、把数据写成模数、把调用关系和依赖关系写死确实一时能达到效率的提升，但局部最优往往不意味着整体最优，多写两行代码有时确实是必要的。同时写出伪代码确实有着先设计后实现的思想。这次实验中，我在写筛法和评测机制里确实吃了没设计先实现的亏，把代码重写和分块了两三遍。但是一旦重写以后，写外层的包装代码就很快了，这就是好的系统设计和先设计后实现原则的明显好处。

对系统维护来说，其实对于每个接口的意义，我感觉其实也应该加注释，标准格式的注释最好。因为以后可能会出现想用某个接口，却用不出来的情况。

总之，系统设计就应该遵循高内聚、低耦合原则，这也是通常模块化的根本目的。模块化不仅是为了可维护性，更是为了可追溯性和可解释性。对于安全相关的代码来说，可追溯性和可解释性更为重要，因为这些能够增强安全和信任。

## 对课程的建议

感觉可以设定一个比较合理的接口标准，然后造一个OJ系统，进行自动阅卷。设定比较合理的接口标准，系统和我们的代码之间就会有更好的可操作性。OJ自动阅卷可以初步判定算法的正确性，也可以节省老师的精力。

可以出密码学的谜题，然后鼓励我们自己去解决，这样有了任务驱动，写起程序来更有动力。

## 总结

这次密码学实验，虽然是第一次，但是我获得的经验教训都不少。不仅是算法方面的，还有算法之外的东西。这些经验不仅能知道我以后的密码学实验，也能指导我以后的编程。