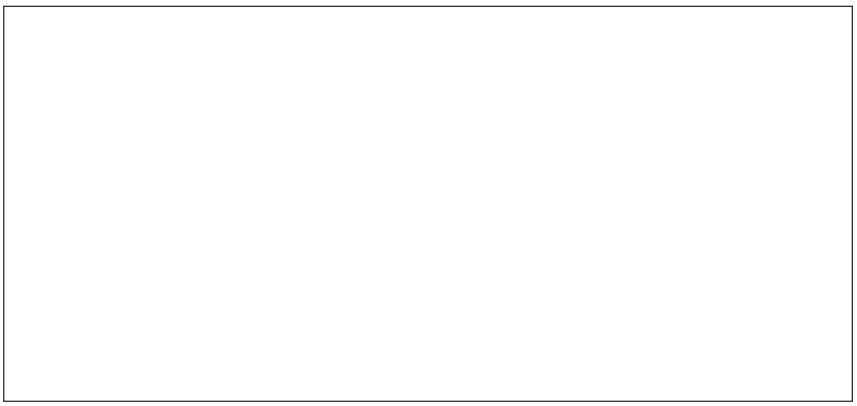
1.2



Public Key Cryptography

Contents

1 Introduction **1**

2 Prime Factorisation **2**

3 Euclid's Algorithm **5**

4 RSA **6**

A Programs **8**
A.1 Documentation 8
A.2 PrimeFactor.py 9
A.3 Euclid.py 10
A.4 RSA.py 11
A.5 Q9.py 11

1 Introduction

This project is programmed in **Python 3.4**. Consult section A for program documentation, listings and information on the structure of the programming for this project. The graph and flowchart are produced using Microsoft Excel 2010. This report is written in LaTeX 2_ε.

Notation. $x, y \in \mathbb{Z}$, $a, b, n \in \mathbb{N}$.

$$\begin{aligned} \mathbb{P} &:= \{t \in \mathbb{Z} : t \text{ is prime.}\} \\ [x, y] &:= \{t \in \mathbb{Z} : x \leq t \leq y\} \\ a \perp b &:\iff \text{hcf}(a, b) = 1 \\ x \equiv_n y &:\iff x \equiv y \text{ (modulo } n) \end{aligned}$$

2 Prime Factorisation

Q1. The function `Prime(n)` ($n \in \mathbb{N}$) uses trial division with the standard modification of only dividing by every number up to and including $\lfloor\sqrt{n}\rfloor$. If a proper factor r greater than $\lfloor\sqrt{n}\rfloor$ exists, there exists a proper factor less than or equal to $\lfloor\sqrt{n}\rfloor$, namely $\frac{n}{r}$, whose discovery would have already revealed that n is not prime.

Sample output of `Prime(n)`.

```
>>> for i in range(11,40,2) :
print(str(i)+', '+str(Prime(i)
   ))


11, prime
13, prime
15, non-prime
17, prime
19, prime
21, non-prime
23, prime
25, non-prime
27, non-prime
29, prime
31, prime
33, non-prime
35, non-prime
37, prime
39, non-prime
```

Q2. `Factor(n)` returns the prime factorisation of $n \in \mathbb{N}$.

Sample output of `Factor(n)`.

```
>>> for i in range(11,40,2):
print(str(i)+', '+str(Factor(i
   )))


11, [11]
13, [13]
15, [3, 5]
17, [17]
19, [19]
21, [3, 7]
23, [23]
25, [5, 5]
27, [3, 3, 3]
29, [29]
31, [31]
33, [3, 11]
35, [5, 7]
37, [37]
39, [3, 13]
>>> Factor(144)
[2, 2, 2, 2, 3, 3]
>>> Factor(13*19*31)
[13, 19, 31]
>>> Factor(2**3*3**4*5**2)
[2, 2, 2, 3, 3, 3, 3, 5, 5]
```

Flowchart for `Factor(n)`. An *arithmetic operation* is henceforth defined[1] as a single addition, subtraction, multiplication, division (simultaneously quotient and remainder), or square root. In figure 1, a step represented with a blue border requires a single arithmetic operation.

[1]This weighting doesn't correlate well with the actual amount of time required by a computer to perform these operations, but simplifies the analysis of complexity.

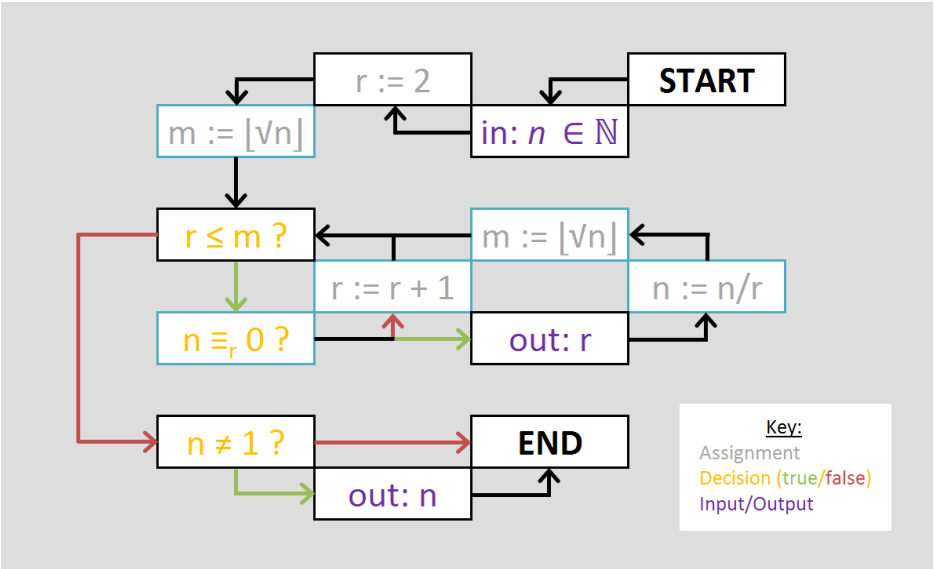


Figure 1: `Factor(n)` algorithm flowchart.

The function `FactorTrace(n)` counts arithmetic operations as detailed in the flowchart. Denoting this number as $\chi(n)$,

$$\chi(n) = 1 + 3a + 2b$$

where a and b are the number of successes and failures of the $n \equiv_r 0$ trial, respectively. This will be used to determine the complexity of the algorithm.

An **upper bound** on $\chi(n)$ can be determined by comparing the values of r, m, n at each step to their initial values, denoted r_0, m_0, n_0 respectively so that $r_0 = 2$. At each $n \equiv_r 0$ trial, there is either a success or a failure. Denoting the next value of each variable with a prime, the following occurs at each step:

$$\begin{array}{lll} \text{Success :} & r' = r & n' = \dfrac{n}{r} \leq \dfrac{n}{2} \\ \text{Failure :} & r' = r + 1 & n' = n \end{array}$$

Thus, successive values of r form an increasing sequence and those of n consequently form a decreasing sequence (noting that $r \geq 2$ always). Moreover, each step modifies either r or (n and m) only. Hence, $r = r_0 + b$ and $n \leq 2^{-a} n_0$ (the latter by induction).

Now, the algorithm terminates when $r > m$, which is guaranteed to occur if either $n < r_0$ or $m_0 < r$ (noting that $n < r_0 \implies m < r_0$ – a simplification). Denoting the number of successes and failures up to the current point as α and β respectively, this means:

$$\begin{aligned} m = \lfloor \sqrt{n_0} \rfloor < r = 2 + \beta &\iff \beta > \lfloor \sqrt{n_0} \rfloor - 2 > \sqrt{n_0} - 3 \\ n \leq 2^{-\alpha} n_0 < r_0 = 2 &\iff \alpha > \log_2(n_0) + 1 \end{aligned}$$

This yields the following upper bounds for a and b, which are the lowest values of α and β satisfying termination:

$$\begin{aligned} b &\leq \sqrt{n_0} - 2 \\ a \leq \log_2(n_0) + 2 = 2\log_2(\sqrt{n_0}) + 2 &\leq 2\sqrt{n_0} + 2 \end{aligned}$$

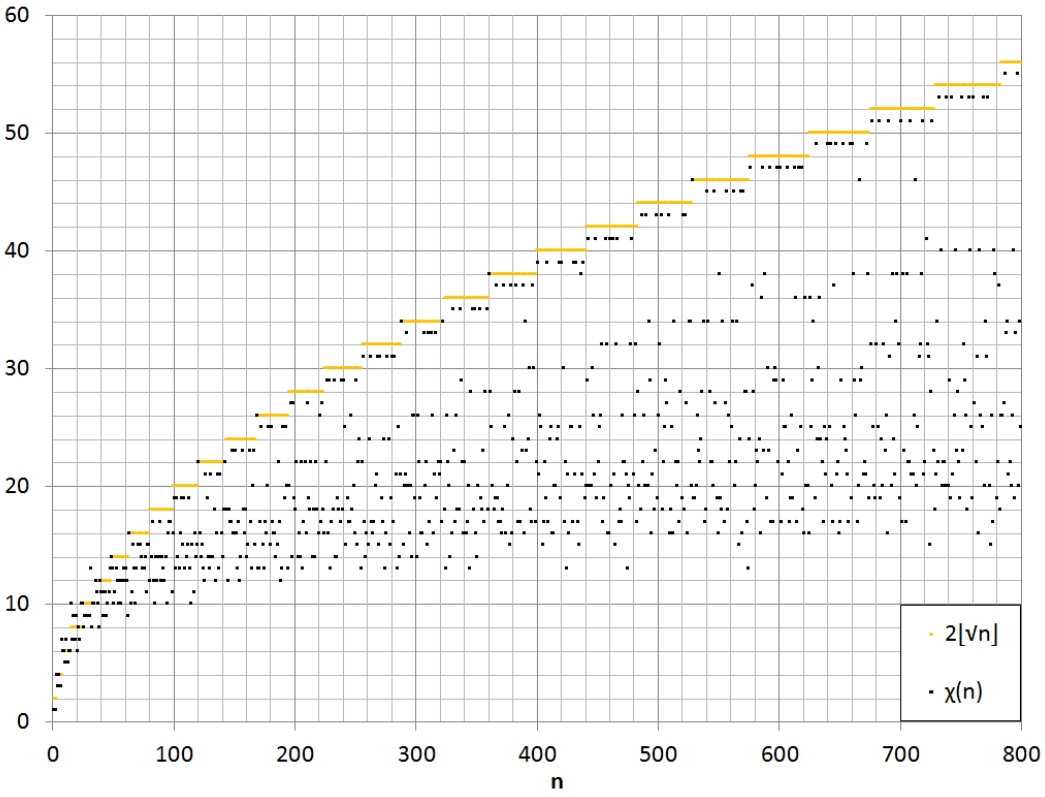


Figure 2: $\chi(n)$ compared to the putative upper bound $2\lfloor\sqrt{n}\rfloor$.

Therefore, $\forall n \in \mathbb{N}$,

$$\chi(n) = 1 + 3a + 2b \leq 8\sqrt{n} + 3$$

A **lower bound** can be attained by noting that

$$\forall p \in \mathbb{P} \quad \chi(p) = 2\lfloor\sqrt{p}\rfloor - 1$$

This is because the factorisation of a prime number consists of $m - 1$ failures and nothing else (noting that $m = \lfloor\sqrt{n}\rfloor$). There are infinitely many primes, so, in this case, the algorithm has complexity $\Theta(\sqrt{n})$. This matches the upper bound, so the worst-case complexity is $\Theta(\sqrt{n})$.

More precisely, the worst-case complexity appears to be asymptotic to $2\sqrt{n}$. A stricter lower bound of $2\lfloor\sqrt{n}\rfloor$ arises from considering numbers of the form $n = p^2$, $p \in \mathbb{P}$ (noting that there are $m - 2$ failures and 1 success; these numbers also occur infinitely often). Above $n > 48$, however, checking as far as $n = 5000$, $\chi(n)$ is bounded above by $2\lfloor\sqrt{n}\rfloor$ (see figure 2).[2]

The worst-case complexity of both this algorithm and the trial division algorithm (counting as set out above) is thus $\Theta(\sqrt{n})$. Trial division requires $2\lfloor\sqrt{p}\rfloor - 1$ operations for $p \in \mathbb{P}$, like the former, and less for $p \notin \mathbb{P}$. Factorisation continues after it has found the first factor, unlike the latter, but on account of reducing n greatly by division, its time-complexity is still $\Theta(\sqrt{n})$. Indeed, heuristically, the primes are virtually the worst case for the factorisation algorithm.

[2]This deduction is made from the raw data output by running `FactorTrace(n)` on $n \in [1, 5000]$. I include figure 2 purely for interest.

3 Euclid's Algorithm

Q3. The highest common factor of each pair of numbers is the LHS of the following equations.

```
>>> EuclidLC(2077945721,2659982993)
'8609 = 41467(2659982993) + -53082(2077945721)'
>>> EuclidLC(462094817,1547231131)
'3407 = -22943(1547231131) + 76820(462094817)'
>>> EuclidLC(112176517033,57479712010)
'19397 = -283711(112176517033) + 553686(57479712010)'
>>> EuclidLC(919805092492,543225077822)
'2 = 40786982269(919805092492) + -69061749043(543225077822)'
```

Q4. This section explains the method used by `Cong(a,b,n)` to solve $ax \equiv_n b$ for $x \in \mathbb{Z}$, with $n \in \mathbb{N}$, $a, b \in \mathbb{Z}$.[3]

Let $h = \text{hcf}(a, n)$. Then by Bezout's lemma,

$$(\exists x \in \mathbb{Z} : ax \equiv_n b) \iff (\exists x, y \in \mathbb{Z} : ax + ny = b) \iff h|b$$

Thus, there is a solution iff $h|b$.

If so, let $\alpha = \frac{a}{h}$, $\beta = \frac{b}{h}$, $\nu = \frac{n}{h}$ so that $\alpha, \beta \in \mathbb{N}_0$, $\nu \in \mathbb{N}$ (note $h > 0$ as $n \neq 0$). Then $\forall x \in \mathbb{Z} \quad ax \equiv_n b \iff \alpha x \equiv_\nu \beta$. $\text{hcf}(\alpha, \nu) = 1$, so α is invertible modulo ν. By Euclid's algorithm, we can find $s, t \in \mathbb{Z} : h = sa + tn$,[4] which implies $1 = s\alpha + t\nu$, whence $1 \equiv_\nu s\alpha$, so $\alpha^{-1} \equiv_\nu s$. Therefore, $\forall x \in \mathbb{Z}$

$$ax \equiv_n b \iff \alpha x \equiv_\nu \beta \iff x \equiv_\nu \alpha^{-1}\beta = \frac{sb}{h}$$

Q5. There are no solutions iff $\text{hcf}(a, n) \not| b$.

```
>>> Congo(718141,20559,932191)
'x = 559615 (mod 932191)'
>>> Congo(718141,20559,968280)
'No solution.'
>>> Congo(718141,20559,999915)
'x = 2814 (mod 11235)'
```

[3] This method works for all $a \in \mathbb{Z}$, but `Cong(a,b,n)` sets a to its residue modulo n (which yields an equivalent problem) before implementing it, to simplify the programming.

[4] Running Euclid's algorithm on a, n rather than α, ν saves the program from having to run the algorithm twice.

4 RSA

Q6.

Complexity of finding p and q. p and q are determined by applying trial division to n. This is done using `FactorOnce(n)`; as n is a product of two primes, the output is its prime factorisation.

The algorithm requires exactly $2\min(p,q) - 2 \leq 2\sqrt{n}$ arithmetic operations[5], and thus is asymptotically dominated by $2\sqrt{n}$. The worst-case complexity is in fact asymptotic to $2\sqrt{n}$; this follows, for instance, from the fact that there exists $k \in \mathbb{N}$ such that there are infinitely-many pairs of primes differing by k.[6] Choosing successively further prime pairs, $\frac{\min(p,q)}{\sqrt{n}} \to 0$, so $\frac{2\min(p,q)-2}{2\sqrt{n}} \to 0$.

Complexity of finding d. d is returned by `Key(n,e)`. Once $\phi(n)$ has been determined, the algorithm requires $\max(3\eta - 1, 1)$ arithmetic operations, where $\eta = \texttt{len(r)} - 2$ (in the notation of the program) is the number of divisions required by Euclid's algorithm to find $\mathrm{hcf}(e, \phi(n))$.[7] Hence, determining its complexity is tantamount to determining η. In accordance with the convention of the program, the remainder list is denoted $(r_i)_{i=0}^{i=\eta+1}$, so that $r_\eta = \mathrm{hcf}(e, \phi(n)) = 1$ and $r_{\eta+1} = 0$.

Upper bound: Suppose $e \geq 1$ so that $\eta \geq 1$.[8] Let $k \in [0, \eta - 1]$. $r_k \geq r_{k+1} \geq r_{k+2}$, so $q_k \geq 1$, so $r_k = q_k r_{k+1} + r_{k+2} \geq r_{k+1} + r_{k+2} \geq 2r_{k+2}$. Hence, by induction, $\eta \in 2\mathbb{N} \implies 2^{\frac{\eta}{2}-1} r_\eta \leq r_2 \leq r_1$ and $\eta \in 2\mathbb{N} - 1 \implies 2^{\frac{\eta-1}{2}} r_\eta \leq r_1$. Therefore, $\eta \leq 2\log_2\left(\frac{r_1}{r_\eta}\right) + 2 = 2\log_2(e) + 2$.

Lower bound: For the worst case, consider the *Fibonacci numbers* $\{F_r\}_{r \in \mathbb{N}_0}$.[9]. These satisfy two important properties:

- $\forall k \in \mathbb{N}_0 \quad 2\log_3(F_k) \leq k$. This is because $\forall t \in \mathbb{N}_0 \quad F_{t+2} = 2F_t + F_{t-1} \leq 3F_t$, whence inductively $k \in 2\mathbb{N}_0 \implies F_k \leq 3^{\frac{r}{2}} F_0 = 3^{\frac{r}{2}}$ and $k \in 2\mathbb{N} - 1 \implies F_k \leq 3^{\frac{r-1}{2}} F_1 = 3^{\frac{r-1}{2}}$. Thus, $F_k \leq 3^{\frac{r}{2}}$, whence $2\log_3(F_k) \leq k$, giving the result.
- $\forall k \in \mathbb{N}_0$ The number of divisions performed by Euclid's algorithm on (F_k, F_{k+1}) is k. This is because $\forall t \in \mathbb{N}_0 \quad 2F_t \geq F_{t+1}$, so every division returns a quotient of 1, and the preceding Fibonacci number as the remainder (until the last step, which outputs a remainder of 0, the penultimate step having given 1).

Now, set $k = \max\{t \in \mathbb{N} : t \leq 2\log_3(e)\}$ (with $e \geq 2$)[8]. Then $F_k \leq 3^{\frac{k}{2}} \leq e \leq 3^{\frac{k+1}{2}}$. $F_k \leq e$ (with F_{k+1}) thus yields a case of k divisions, so the worst-case complexity is greater than or equal to $k \geq 2\log_3(e) - 1$.

The Fibonacci numbers diverge to infinity, and successive Fibonacci numbers are coprime, but it is unknown[10] if there are infinitely many Fibonacci numbers of the form $(p-1)(q-1)$, $p, q \in \mathbb{P}$. If so, these could be taken as $\phi(n)$ and the preceding Fibonacci number as e, to yield a case with the required worst-case complexity. However, this analysis does prove, noting that $\log(x) \equiv \log(2)\log_2(x) \equiv \log(3)\log_3(x)$ on $(0, \infty)$, that the worst-case complexity of Euclid's algorithm in general (counting in divisions) is $\Theta(\log(n))$, where n is the lower of the inputs, and the complexity of `Key(n,e)` (counting in arithmetic operations) is $O(\log(n))$.

[5] $n - 2$ failures and 1 success, each costing 2 operations including the test itself.

[6] This result was proven by Yitang Zhang in 2013. The upper bound on k, initially 70,000,000, has now been reduced to 246 through an ongoing Polymath project.

[7] The individual operations are labelled in the program listings.

[8] These assumptions eliminate the need for special cases and are without loss of generality, as asymptotic behaviour for large e, n is being considered.

[9] Definition: $F_0 = F_1 = 1$; $\forall r \in \mathbb{N}_0 \quad F_{r+2} = F_{r+1} + F_r$

[10] To me, at least.

Q7. The decryption keys are as follows:

```
>>> Key(1792393783,99833)
1653136833
>>> Key(1837601609,50512913)
1062003017
>>> Key
   (9996158063509,123456715)
9478063366735
>>> Key(1682749591,166907411)
78553691
>>> Key(6718172889047,901)
141670569661
>>> Key(1617097231,34577)
338486129
```

Q8. The function `Crypt(c,n,d)` computes $c^d \pmod{n}$ by iteratively multiplying c, taking the residue modulo n at each step to minimise the computation time of each multiplication. As it is simply a modular exponentiation function (also usable to encrypt), setting $\mathtt{d} = e$ and $\mathtt{c} = c^d$ yields m; this can be used to check the solution. For example, with $n = 77$, $e = 17 \perp \phi(n) = 60$, $c = 4$:

```
>>> Key(77,17)
53
>>> Crypt(4,77,53)
9
>>> Crypt(9,77,17)
4
```

The entire algorithm tree is built using integer arithmetic, so there is no possibility of an inaccurate result, regardless of the size of n.[11] The computer may, of course, fail to process the calculation in an affordable time frame, if n is too large.

Q9. This is the output of the script `Q9.py`:

```
>>>
Key: 82393
Message: what did james ellis clifford cocks and malcolm
   williamson do first: invent public key cryptography
```

[11] Thou shalt not encourage floating-point approximations to integer computations: `https://dolphin-emu.org/blog/2014/03/15/pixel-processing-problems/`

A Programs

The programs take the form of three modules, providing functions for the three sections, and one script for Q9. These modules are loaded into the shell via an additional initialisation module `Init.py` (which loads all of each module's functions into the global symbol table), and commands are then typed into the shell in an interactive session (with output recorded).

A.1 Documentation

This section describes the purpose of the project's functions, and the inputs for which they are designed to give valid output. The order they appear in corresponds to the listings.

`Prime(n)`: $n \in \mathbb{N}$. Checks if n is prime.

`Factor(n)`: $n \in \mathbb{N}$. Returns the prime factorisation of n as a descending list with repeats.

`FactorTrace(n)`: $n \in \mathbb{N}$. Returns the number of arithmetic operations required to run `Factor(n)`.

`FactorOnce(n)`: $n \in \mathbb{N}$. Returns `[r,s]`, where `r` is the smallest non-1 factor of n and `s` is its complement.

`EuclidA(a,b)`: $a, b \in \mathbb{N}_0$. Returns `(r,q)`, where `r` is the list of remainders (preceded by $\max(a,b)$, $\min(a,b)$) and `q` is the list of quotients obtained from the divisions executed by Euclid's algorithm running on a, b.

`EuclidB(a,b)`: $a, b \in \mathbb{N}_0$. Returns `[c,y,x]`, where $c = y\max(a,b) + x\min(a,b)$, as obtained by back-substitution using Euclid's algorithm on a, b.

`EuclidHcf(a,b)`: $a, b \in \mathbb{N}_0$. Returns $\text{hcf}(a,b)$,[12] from the remainder list of Euclid's algorithm.

`EuclidLC(a,b)`: $a, b \in \mathbb{N}_0$. Returns a string representation of `EuclidB(a,b)`.

`Cong(a,b,n)`: $n \in \mathbb{N}$, $a, b \in \mathbb{Z}$. Returns `[r,m]`, where $x \equiv_m r$ is the solution to the congruence $ax \equiv_n b$, where the solution exists.

`Congo(a,b,n)`: $n \in \mathbb{N}$, $a, b \in \mathbb{Z}$. Returns a string representation of `Cong(a,b,n)`.

`Tot(n)`: $n \in \mathbb{N}$. Returns $\phi(n)$ if n is a product of two distinct prime numbers.

`Key(n,e)`: $n \in \mathbb{N}$, $e \in \{x \in [0, n-1] : x \perp e\}$. Returns `Cong(e, 1, Tot(n))[0]`. It is a rewrite of this function, optimised to be used on the restricted set of inputs it allows.

`Crypt(c,d,n)`: $c, d \in \mathbb{N}_0$, $n \in \mathbb{N}$. Returns the residue of c^d modulo n.[12]

[12] Convenient convention: $\text{hcf}(0,0) = 0$; $0^0 = 1$

A.2 PrimeFactor.py

```
import math

def Prime(n):
    if n == 1:
        return 'non-prime'
    m = math.floor(math.sqrt(n))
    for r in range(2, m+1):
        if n % r == 0:
            return 'non-prime'
    return 'prime'

def Factor(n):
    x = []
    r = 2
    m = math.floor(math.sqrt(n))
    while r <= m:
        if n % r == 0:
            x.append(r)
            n = n // r
            m = math.floor(math.sqrt(n))
        else:
            r += 1
    if n != 1:
        x.append(n)
    return x

def FactorTrace(n):
    r = 2
    m = math.floor(math.sqrt(n))
    y = 1
    while r <= m:
        if n % r == 0:
            n = n // r
            m = math.floor(math.sqrt(n))
            y += 3
        else:
            r += 1
            y += 2
    return y

def FactorOnce(n):
    for r in range(2, n+1):
        if n % r == 0:
            return [r, n//r]
    return []
```

A.3 Euclid.py

```
def EuclidA(a,b):
    r = [max(a,b),min(a,b)] # the remainder list
    q = []                  # the quotient list
    while r[-1] != 0:
        t = divmod(r[-2],r[-1])
        r.append(t[1])
        q.append(t[0])
    return (r,q)        # len(r)-2 iterations, 1 operation each

def EuclidB(a,b):
    (a,b) = (max(a,b),min(a,b))
    (r,q) = EuclidA(a,b)
    (x,y) = (1,0)
    # Now, r[-2] == x*r[-2] + y*r[-3]
    for i in range(len(r)-4,-1,-1):
        (x,y) = (y - x*q[i],x)
        # Now, r[-2] == x*r[i+1] + y*r[i]
    return (r[-2],y,x)  # max(len(r)-3,0) iterations, 2
       operations each

def EuclidHcf(a,b):
    return EuclidA(a,b)[0][-2]

def EuclidLC(a,b):
    (c,y,x) = EuclidB(a,b)
    (c,y,x) = (str(c),str(y),str(x))
    (a,b) = (str(max(a,b)),str(min(a,b)))
    return c + ' = ' + y + '(' + a + ') + ' + x + '(' + b + ')'

def Cong(a,b,n):
    a = a % n
    e = EuclidB(a,n)
    h = e[0]
    if b % h != 0:
        return 'No solution.'
    b = b // h
    n = n // h
    return [(e[2] * b) % n, n]

def Congo(a,b,n):
    c = Cong(a,b,n)
    if c == 'No solution.':
        return c
    return 'x = ' + str(c[0]) + ' (mod ' + str(c[1]) + ')'
```

A.4 RSA.py

```
from PrimeFactor import *
from Euclid import *

def Tot(n):
    x = Factor(n)
    if len(x) != 2 or x[0] == x[1]:
        return 'Input must be a product of two distinct primes.'
    return (x[0] - 1) * (x[1] - 1)

def Key(n,e):
    t = Tot(n)
    return (EuclidB(e,t)[2]) %t # 1 + [EuclidB(e,t)] operations

def Crypt(c,n,d):
    r = 1
    for i in range(1,d+1):
        r = (r * c) % n
    return r
```

A.5 Q9.py

```
from RSA import *
x = [569010, 157904, 679003, 511858, 64330, 227775, 798880,
   345152, 334332, 594524, 917269, 866647, 92778, 834013, 372172,
    558357, 210768, 528931, 818047, 587250, 357542, 437704,
   968899, 546508, 538213, 130764, 589138, 331077, 305125,
   255352, 545397, 311491, 725411]
n = 998191
e = 123457
d = Key(n,e)
y = []
for i in range(len(x)):
    y.append(Crypt(x[i],n,d))
s = ''
for i in range(len(y)):
    q = str(y[i])
    s = s + ('0'*(6-len(q))) + str(y[i])
t = ''
for i in range(len(s)//2):
    (a,b)=(s[2*i],s[(2*i)+1])
    t = t + " abcdefghijklmnopqrstuvwxyz.:'"[int(a + b)]
print('Key: ' + str(d))
print('Message: ' + t)
```