The RSA Algorithm

Q1i):

We can speed up the process of determining whether n is prime or not by reducing the number of iterations the for loop must perform. We simply edit the code to check up to the rounded-up value of $\sqrt{n}$ (IE, $\lceil \sqrt{n} \rceil$). This follows since "Fibonacci observed that to determine if a number n is prime, it suffices to divide by the integers $\leq \sqrt{n}$ " [1] .

Q1ii):

Take two similar sized numbers for n, say $n_1 = 989765438$ and $n_2 = 989765439$. Using the MATLAB code for 1c), factoring n_2 into its prime factors will take longer than for n_1. We find:

$$n_1 = 989765438 = 2 \cdot 151 \cdot 191 \cdot 17159, \text{ and } n_2 = 989765439 = 3 \cdot 13 \cdot 25378601.$$

Note above that n_2 contains a large prime factor, and hence the factoring will take longer. Generalizing, we can say that given two numbers, n_1, n_2, the one containing at least one reasonably large prime factor will take longer to fully factorize, as the code will have to run that many more times. Note that this also indicates that the further the factors of a given n are apart, the longer the factorization will take.

Q1iii):

As stated in Q1ii), if n contains a big (prime) factor, the time to break it into its factors will become longer. Consider $n = a \cdot z$, where $a < z$. We initially must check values $i = 2 : n$, however after $a - 1$ steps, when $i = a$ is reached, we pull out the prime factor and divide that range by a so that it is reduced to $i = a : z$ so that the loop will have to run $z - a$ more times until z is found. In the code for 1c) there are roughly 8 operations within the loop, noting that the mod-function counts as 4 operations, so that the number of operations is about $8 \cdot ((a - 1) + (z - a)) = 8 \cdot (z - 1)$. Note that if z is large enough, the number of calculations being performed increases and as a result it takes longer to break n into its primes, so that the equation above acts as a sort of upper bound.
An efficient code would have reduced the number of operations to about $8 \cdot \lceil \sqrt{n} \rceil + (a - 1)$. Here it would check if the new range $n = \frac{n}{i}$ was prime using the code from 1b) and consequently, would only have to go through $\sqrt{n}$ iterations of the loop. This would have stopped the loop from checking all values from a to z, since we would know if z was prime or not. After code was implemented (refer to code in appendix) the loop continued to check values up to z, since it took roughly the same time as without this edit, so there must have been an error.

Q1iv):

The code for 1b) requires that the remainder of n divided by i be non-zero if n is to be prime. Now if we let i range from 1 to $(n - 1)$, we find that the code will say that any number n is not prime since $n \,(mod\, 1) = 0$ and so our code will fail at its task. Even by the common definition of a prime number p, p must have 2 distinct factors, namely 1 and itself. For 1 we find that these are both equal (not distinct).

Q2i):
The code needs to consider that the first entry in the vector r is the larger of a and b. This means that we need to ensure that the first two entries in the vector 4 are defined properly so that the code runs correctly.

Q2ii):
For group F, we solve the following:

$$\begin{aligned} HCF(853669,\, 806541) &= 1 = 853669 \cdot 101776 + 806541 \cdot (-107723), \\ HCF(853669,\, 868948) &= 1 = 868948 \cdot (-335791) + 853669 \cdot 341801, \\ HCF(853669,\, 898025) &= 1 = 898025 \cdot (-196019) + 853669 \cdot 206204. \end{aligned}$$

Q2iii): With the help of code a) in part b), we can decide if solutions to the linear congruence problem exist or not. Set $V = Q2aHCF(a,\, n)$, where V is equal to a vector containing the values of HCF, u and v. These can be extracted individually to be used in the process of determining if solutions exist. We simply need to find if $c\,(mod\, V(1)) \equiv 0$, where c is from the input (referring to $ax = c\,(mod\, n)$), and $V(1)$ is the HCF – the first element of the output in code a)). Essentially, for solutions to exists, $V(1)$ must divide into c.

Q2iv):

$$\begin{aligned} 74946x &\equiv 5184\,(mod\, 330389) \;-\; \text{has one solution} : x = [190309], \\ 74946x &\equiv 5184\,(mod\, 655678) \;-\; \text{has two solutions} : x = [252513,\, 580352], \\ 74946x &\equiv 5184\,(mod\, 983517) \;-\; \text{has three solutions} : x = [252513,\, 580352,\, 908191], \\ 74946x &\equiv 5184\,(mod\, 162383) \;-\; \text{has no solutions} : x = [\,]. \end{aligned}$$

No solutions exist to the last equation, referring to the argument in Q2iii), since $HCF(74946, 162383)$ $= 12491$ and this does not divide into 5184.

Q3i)

$$\begin{aligned}(999797,\ 949951) &\Rightarrow (999797,\ \mathbf{127}),\\ (898993,\ 198097) &\Rightarrow (898993,\ \mathbf{31985}),\\ (813691,\ 534233) &\Rightarrow (813691,\ \mathbf{446297}),\\ (145157,\ 3649) &\Rightarrow (145157,\ \mathbf{277}).\end{aligned}$$

Note that the above decryption keys, d, are given in bold.

Q3ii)
We shall cover the task of finding d in three parts, firstly finding the two primes of which n comprises of, calculating $\varphi(n)$, and lastly solving the linear congruence problem to obtain d. We make the following assumption : $n = p \cdot q$ where $p > q$.

Factoring n back into p and q will require about $8 \cdot (p - 1)$ operations, recalling a similar result in Q1iii). The calculation for $\varphi(n)$ only requires 3 operations and to lastly obtain d, we need to find $HCF(e,\ \varphi(n))$. Now, it can be shown that "if $u > v > 0$, then the number of division steps $E(u,\ v)$ performed by the Euclidean algorithm is always less than 5 times the number of decimal digits in v"[1]. We find that this part requires roughly $5 \cdot decDig(e) + 38$ operations.

Counting the operations in the code, we have a total $8 \cdot (p - 1) + 3 + 5 \cdot decDig(e) + 38$ calculations, where $e = 1 - b \cdot \varphi(n)(mod\, a)$ and $a,\ b \in \mathbb{Z}$ solutions to $HCF(e,\ \varphi(n))$. This roughly acts as the upper bound on operations a computer will need to perform
The main part for concern is factoring n back into p and q. If $p \gg q$ then the time taken to factor p out increases as will the number of operations. For relatively small values of $n = p \cdot q$, as seen in Q3vi) ($p = 941$ and $q = 967$), it does not cause trouble. However, it becomes cause for major inefficiency and computational expense when using real world values of p and q in the RSA algorithm, which are hundreds of digits long.

Q3iii)
The code for 3b) uses the method of repeated exponentiation to calculate $c^d\ (mod\, n)$. If d is a huge number the code will still work, but it will take longer for the calculation to be performed.

Q3vi)
Decrypted message: SEND PEOPLE.

Q4)
Consider the two following key dependent methods of encoding – RSA (named after the founders, Ron Rivest, Adi Shamir and Leonard Adleman) and the Advanced Encryption Standard (AES). Whilst the RSA method uses asymmetrical keys, meaning that the encryption and decryption keys differ, the AES uses a single symmetric key for both these procedures. There are certain advantages when using one or the other of the practices which will be discussed below.

The AES allows for 16 bytes (contained within in a 4x4 matrix) of data to be encoded at a time sequentially, so that, essentially, there is no limit, besides storage, on how much one can encrypt. AES provides a very good encryption system against brute force attacks; however, AES applies a single key for both the encoding and decoding of data. This makes it susceptible to man-in-the-middle attacks, where a third-party intercepts the key. This person would then be able to reverse the cipher text using this key and gain access to the information within, which is a security issue. Compare this to RSA, which uses asymmetrical keys for encoding and decoding . Here the encryption key, e, is public and the decryption key, d, is kept private. Someone who intercepts e will not be able to decrypt the message as was the case for symmetric encryption methods, such as AES. However, there are a few things to consider when using RSA. Alongside the encryption number e, there is another number n, also publicly available, which will serve as the modulus in the mathematical calculation. This number is strongly advised to be the product of 2 huge primes. If they are chosen to be relatively small, for example, if $n = p \cdot q$ where $p = 941$ and $q = 967$ as in Q3vi), it is straightforward to calculate d, given n and e by using a normal computer, I.E. no supercomputer is required for this task.

By selecting a product of two huge primes, the brute force calculation for d becomes increasingly time inefficient, referring to Q1ii), where we observed that a product of large primes will take longer to factor those of small primes. The process of calculating n is easy, whereas the reverse, finding the factors that make up n, is not. This is the underlying idea that RSA's security relies on.

The strength of RSA may become its vulnerability within the next few decades as technology progresses, however. Consider the factoring estimate using the Ekerå-Håstad's algorithm. "[...], a single correct run of this quantum algorithm suffices for the RSA integer to be factored with at least 99 % success probability in the classical post-processing." [2]

Although there are and will be numerous issues with post-quantum cryptography in the future, a report has discussed future possible methods of encryption, resistant to quantum computers, such as: Lattice-based cryptography and Code-based cryptography [3].

Another issue with RSA is that it can only encrypt messages of smaller or equal size to the key used. If a 4096-bit key is chosen, the maximal size of the message being encoded must be 4096 bits or smaller. This clearly is a very inefficient way of transmitting large files, such as images. RSA can be used in conjunction with AES as follows to make data transmission even more secure. We make use of the limited data that RSA can encode : the typical bit size for RSA ranges from 1024, 2048 to up to 4096 bits, allowing enough space for a 256-bit key (from AES) to be encrypted. This enables the symmetric key to be sent much more securely.

Overall, when using appropriate sized values in the algorithm, RSA provides a sufficient method of encryption to date. Despite this, future factoring methods may be able to efficiently crack RSA, AES and many more encryption methods.

References

[1] Jeffrey Shallit, *LaTeX: Origins of the Analysis of the Euclidean Algorithm, HISTORIA MATHEMATICA.* ScienceDirect, Volume 21, Issue 4, November 1994, Pages 401-419, [online] Available at https://doi.org/10.1006/hmat.1994.1031.

[2] Craig Gidney and Martin Ekerå, *LaTeX: How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits.* Quantum Journal, Volume 5, 2021, Pages 433, [online] Available at https://doi.org/10.22331/q-2021-04-15-433.

[3] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, Daniel Smith-Tone, *LaTeX: Report on Post-Quantum Cryptography.* [online] Available at http://dx.doi.org/10.6028/NIST.IR.8105.

Matlab Code

Code for Question 1a)

```
function  f=Q1aTestPrime(n)
%The code below will test a natural number n>1 if it is prime or not
%(0=false&1=true)
   if n == 1
       f=0;
```

```
  end
 for i = 2: (n-1)          % check all nbrs up to n-1
     if mod(n,i) == 0   % if remainder=0 implies it is not a prime
        f=0;
        break           % break when a divisor is found
     else
        f=1;
     end
 end
end
```

Code for Question 1b)

```
function  f=Q1bTestPrime(n)
%This code is the improved version of code for Q1a
if n == 1
   f=0;
end
for i = 2 : ceil(n^(0.5))
      if mod(n,i) == 0
          f=0;
          break                   % break when a divisor is found
      else
          f=1;
      end
    end
end
```

Code for Question 1c)

```
function f = Q1cPrimeFactors(n)
%The code below will list all the prime factors of a natural number n > 1.
v=1;
for i=2:n                         %start from 2 since 1 is not prime
  while mod(n,i)==0
        f(v)= i;                  %pull out factor
        v=v+1;                    %increase vector by 1 unit
        n=n/i;                    %new n, in n=pq, new n is n/q=p
        if n==1
            break
        elseif Q1bTestPrime(n)==1
            break
        end
  end
end
end
```

Code for Question 2a)

```
function f=Q2aHCF(a,b)
% This function determines the highest common factor of two natural numbers
% and finds solutions to HCF(a,b)= au + bv for u,v integers.
i=1;
r=[];
if a<b
     r(i)=b;
     r(i+1)=a;
else
     r(i)=a;
     r(i+1)=b;
end
u=[1 0];
v=[0 1];
q=[0 0];
while mod(r(i),r(i+1))~=0
    k=mod(r(i),r(i+1));
    r(i+2)=k;
    q(i)=(r(i)-k)/(r(i+1));
    u(i+2)=u(i)-q(i)*u(i+1);
    v(i+2)=v(i)-q(i)*v(i+1);
    i=i+1;
end
f=[r(i+1) u(i+1) v(i+1)]; %f(2)=u, f(3)=v s.t. au+bv=HCF, where a>b
end
```

Code for Question 2b)

```
function f=Q2bLcEDITED(a,c,n)
%This function solve the linear congruence problem : ax=c(mod n)
%and gives solutions in the range x in [0,...,n-1]
%reduce to a,c (mod n)
a = mod(a,n);
c = mod(c,n);
V=Q2aHCF(a,n);  %let V be the vector where 1st entry is HCF,
                %2nd is u and 3rd is v (a>b for HCF(a,n)=au+nv)
                %V(1)=HCF(a,n),V(2)=u,V(3)=v,
                %where HCF(a,n)=au+nv.
%Check if solutions exist:
   if mod(c,V(1))~=0
      f=[];
   end
      if mod(c,V(1))==0
          if V(1)==1
             f=mod(c*V(3)/V(1),n);
          end
        %solution to ax=HCF(a,n), where a<n, is x=V(3)=v
        x=V(3)*(c/V(1));
           for i=1:V(1)-1
                 if x<0
                     while x<0
                         x=x+mod(n/V(1),n);
```

```
                        end
                    end
                f(1) = x;
                f(i+1) = x + mod(n*i/V(1),n);
            end
        end
end
```

Code for Question 3a)

```
function f = Q3aDecryptKey(n,e)
%This function find the inverse, d, of e s.t. e*d=1(mod(phi(n)))
%The output is a vector containing n and d.
%First find phi(n);
%Then solve for e*d=1(mod(phi(n)))
F=Q1cPrimeFactors(n);
phi=(F(1)-1)*(F(2)-1);
M=Q2bLC(e,1,phi);
f=[n, M(1)];
end
```

Code for Question 3b)

```
function f = Q3bConvertEncryptNbr(c,n,d)
%The code below decrypts the code c
%*Improve efficiency using sum of squares...*
%below method uses repeated exponentiation
for i=1:d-1
    k(1)=mod(c,n);
    k(i+1)=mod(c.*k(i),n);
    %c=k(i+1); not needed...
end
f=k(d);
end
```