Question 1

Part a

Here I am to test whether $n = 99512831474559074447420587631965683245067975322122894835681$ is prime. I used the Strong Primality Test algorithm, which analyses a series of remainders generated by dividing various powers of a given base a by n. My implementation is below. I wrote it as a function in case I needed to reuse this algorithm in the future, and to make my intent clearer by calling a well-named function.

```
function isProbablePrime = passesStrongTest(remainders, n)
    % Runs the Strong Primality Test on the already-calculated strong test remainders.
    % Returns 1 if the remainders pass the strong test, and 0 if they fail

    isProbablePrime = 0; % assume not prime. trying to prove otherwise
    if remainders(end) == 1
```

If the above condition is met, I then find the position of the first 1 in the array, since both of the following conditions depend on this index.

```
        firstOneIdx = -1;
        for i = 1 : length(remainders)
            if remainders(i) == 1
                firstOneIdx = i;
                break;
            end
        end
```

Notice how the above algorithm will have `firstOneIdx` as -1 if it could not find a 1, since the `if` statement's condition was never met. I check for this possibility first, then proceed to check whether `firstOneIdx` passes the Strong Primality Test. This is why I also had to pass n to the function – the algorithm may need to verify that the number before the first 1 is $n - 1$.

```
        if firstOneIdx ~= -1 % if there is a 1 at all
            if firstOneIdx == 1 || remainders(firstOneIdx - 1) == n - 1
                isProbablePrime = 1;
            end
        end
    end
    return
end
```

Notice that $n - 1$ will never be out of range for the second check. Because I used a short-circuit `or`, this condition will never be checked if `firstOneIdx` $= 1$.

As you can see, the above function takes an array of already-calculated strong test remainders as an argument. I made a separate function for calculating these remainders, because I only need the remainders (not the full test) in Part b. My implementation of this function is below:

```
function remainders = strongTestRemainders(n, a)
    % Returns the remainders used in the Strong Primality Test algorithm
```

```
    % Uses a as the base to run the test
```

I first check whether n and a meet the requirements of the Strong Primality Test. If they don't – either because they are not coprime, or if n is odd – I throw an appropriate error and the function ends.

```
    if gcd(n, a) ~= 1
        error('n and a should be coprime.');
    elseif mod(n, 2) ~= 1
        error('n should be odd.');
    end
```

I then proceed to find r and s. I loop through every possible value of r, and check if an odd integer s exists to satisfy $2^r s = n - 1$. This is made easier by the fact that r and s are unique for any given n, so I do not have to worry about in what way I check r and s. Since s is not less than 1, I know that r cannot be greater than $\log_2(n)$, so my maximum value of r is the integer part of this logarithm.

```
    for rTest = 0 : floor(log2(n - 1))
        sTest = (n - 1) / 2^rTest;
        if sTest == floor(sTest) && ... % s is an integer
        sTest / 2 ~= floor(sTest / 2)   % s cannot be divided by 2 - i.e. s is odd
            r = rTest;
            s = sTest;
            break;
        end
    end

    % r and s found. now finding remainders
    remainders = zeros(1, double(r) + 1); % preallocating for efficiency
    for i = 0 : r
        pow = sym(s) * 2^i;
        remainders(i + 1) = sym(feval(symengine, 'powermod', a, pow, n));
    end
    return
end
```

Since the above two functions do most of the work for me, my code for testing n is very simple:

```
n = sym('99512831474559074447420587631965683245067975322122894835681');
a = sym(3);

remainders = strongTestRemainders(n, a);
if passesStrongTest(remainders, n)
    fprintf('n could be prime.\n');
else
    fprintf('n is certainly composite.\n');
end
```

This outputted `n is certainly composite.`

Part b

I started by working out two arrays: one for the the GCD of n and each remainder $+1$, and one for the GCD of n and each remainder -1. I preallocated each array for efficiency, then calculated their values by going through each remainder in a `for` loop:

```
gcdsOfIncrementedRems = sym(zeros(1, length(remainders)));
gcdsOfDecrementedRems = sym(zeros(1, length(remainders)));

for i = 1 : length(remainders)
    gcdsOfIncrementedRems(i) = sym(gcd(remainders(i) + 1, n));
    gcdsOfDecrementedRems(i) = sym(gcd(remainders(i) - 1, n));
end
```

I used these two arrays to work out the prime factors of n by taking the primes from these two lists. I did this by going over an array composed of both of the gcd arrays, and adding each element to the array of prime factors if the element is prime and not already in the array. It is necessary to check that the element is not already in the array since `gcdsOfIncrementedRems` and `gcdsOfDecrementedRems` may have duplicate elements.

```
primeFactors = sym(zeros(1, 0));
for g = [gcdsOfIncrementedRems gcdsOfDecrementedRems]
    if ~ismember(g, primeFactors) && isprime(g)
        primeFactors = [primeFactors g];
    end
end

primeFactors
```

This output `[ 51004200000053561761, 76506300000080342641, 25502100000026780881]`, meaning that those three numbers are the prime factors of n.

Part c

This algorithm will fail if n has a duplicate factor, i.e. if $n = p^m k$, where p is prime, $m > 1$ and k is an arbitrary positive integer. This is because we have to check that a prime p is not already in the array before we add it, so in this case p will appear once, not m times.

Question 2

Part a

Here I calculated the private key for the RSA algorithm. I had to start by finding two primes, p and q, that satisfied certain constraints. p had to be the smallest prime greater than $698754312 \times S^6$, where S was my student ID, 10364803. Similarly, q was the smallest prime greater than $6 \times S^7$. Further, $\frac{p-1}{2}$ and $\frac{q-1}{2}$ had to be prime. Noticing that a very similar process would be required to find both p and q, I wrote a function to solve this problem in the general case. In it I refer to such primes as p and q as special primes.

```
function prime = smallestSpecialPrimeGreaterThan(coeff, S, pow)
    % This is the general solution for 2(a).
    % A special prime p is one where (p - 1) / 2 is also prime

    p = coeff * S^pow + 1;
```

The first possible prime, p, is one greater than my starting number, coeff $\times$ S^{pow}. I then use `nextprime` to find the next prime greater than or equal to p. If this number is not a special prime, I make p the next potential prime – the next odd number after what `nextprime` returned – and check this number. This continues until p is a special prime.

```
    while 1
        prime = nextprime(p);
        if isprime((prime - 1) / 2)
            return
        else
            p = prime + 2;
        end
    end
end
```

With this function defined, solving Part a was simple:

```
S = sym(10364803);
p = smallestSpecialPrimeGreaterThan(sym(698754312), S, sym(6));
q = smallestSpecialPrimeGreaterThan(sym(6), S, sym(7));

fprintf(p = fprintf(q = %s\n, string(q));
```

This function returned

```
p = 866344880029504985589518345000808681374788854757259
q = 771043028202260599129142534220475779172533162137 99
```

I then used p, q and $e = 65537$ to find d, my private key. Since $de \equiv 1 \mod (p-1)(q-1)$, solving for d required solving a multiplicative congruence equation. I wrote a function that solved for x in the general case where $ax \equiv b \mod m$, making use of the extended `gcd` function. I also refer to this function twice in Question 4, so having a function that solves this problem makes my code neater.

```
function [x] = solveMultCongruence(a, b, m)
    if gcd(a, m) ~= 1
```

```
        error('gcd(a, m) is not 1');     % no multiplicative inverse exists
    else
        % finding the multiplicative inverse of a:
        [~, u, ~] = gcd(a, m);       % using ~ to denote values that I will not need
        aInv = mod(u, m);
        x = mod(aInv * b, m);
        return
    end
end
```

I then used this function to solve for d as follows:

```
modulus = (p - 1) * (q - 1);
d = solveMultCongruence(65537, 1, modulus);
fprintf(The private key is
```

This outputted

```
The private key is 52635349851639267463777425410833454187638
489077483382501184188110966034762834994144460029534060125085
```

Part b

Here I decrypted the given encrypted message according to my student number. I decrypted the given encrypted message by finding the remainder of the message raised to the power of d, my private key, modulo pq. This was easy to solve with `feval`.

```
encryted = sym('230600964530163734473010252418317318384737256195733
30907412480584688663591666231773072745953968896421');
decrypted = feval(symengine, 'powermod', encryted, d, p * q);
```

However, I then had to convert the decrypted message to text so that I could read it. I used the following method to break down `decrypted`'s digits into an array of two digit pairs. I would first need to find the amount of digits, then the size of the array needed to store them in two-digit groups. I had to use `ceil` for the size of the array because an odd amount of digits would require a bigger array than one with an even amount of digits, so the size would have to be rounded up.

```
amountOfDigits = ceil(log10(decrypted));
N = ceil(amountOfDigits / 2); % size of array
arrayOfTwoDigits = zeros(1, double(sizeOfArray));
```

My method was to convert `decrypted` to an array of characters, then to take the next pair of characters and convert those back to an integer.

```
decryptedAsCharArray = char(decrypted);
decryptedAsString = ;
for i = 1 : N
    currentChars = decryptedAsCharArray(2*i - 1 : 2*i);
    currentDigitPair = str2num(currentChars);
```

I could then convert each of the two digit numbers to a character to build up the decrypted message as a string. Exploiting the fact that the number to letter mapping was ordered in the same way as the ASCII code, I used `char` to convert from a numeric ASCII code to its corresponding character. However, the letter 'A' has ASCII value 64, but its preimage is 1 in the mapping. I had to add 63 to each number to compensate for this. Here I chose to represent spaces manually, since their representation of 0 in the code does not correspond to a suitable ASCII value.

```
    if currentDigitPair == 0
        decryptedAsString = strcat(decryptedAsString,  );
    else
        decryptedAsString = strcat(decryptedAsString, char(currentDigitPair + 64));
    end
fprintf('The decoded message is %s.\n', stringValue);
```

This resulted in the output `The decoded message is MATHEMATICS IS THE DOOR AND KEY TO THE SCIENCES`.

Question 3

Part a

Here I used the built-in functionality to find g, the smallest primitive root modulo p, where p has the same definition as in Question 2:

```
p = sym('86634488002950498558951834500080868137478885475725 9');
g = feval(symengine, 'numlib::primroot', p);
```

Letting $x = S^2$, where S has the same definition as in Question 2, we obtain the following value for y, part of the public key in the ElGamal public key cryptosystem:

```
y = feval(symengine, 'powermod', g, x, p);
fprintf(y is
```

This outputs `y is 20596846858527957253617618652547486464472062620134 2`.

Part b

We now decrypt the message given by the following values of c_1 and c_2 in the ElGamal cryptosystem:

```
c1 = sym('80747105427795637527117551095217035870940213081069 7');
c2 = sym('71072452985306839955023737672929935112378223380303');
```

To decrypt this, I would have to calculate the remainder of $c_2\, c_1^{p-1-x}$ modulo p. I know how to efficiently calculate the remainder of c_1^{p-1-x}, but not with the product c_2. Instead I will calculate the remainders of c_2 modulo p and the exponent of c_1 modulo p separately, then multiply the answers together modulo p at the end.

```
c1DecryptedPart = feval(symengine, 'powermod', c1, p − 1 − x, p);
c2DecryptedPart = mod(c2, p);
decrypted = mod(c1DecryptedPart ∗ c2DecryptedPart, p);
fprintf('decrypted message is %s\n', string(decrypted));
```

This outputs `decrypted message is 31415926535897932384626433832795028842`.

Question 4

Part a

Here I calculate the digital signiature given $p = 99145399569186411432799$, $q = 314873624753$ and $g_0 = 2$. I start by finding g, the non-unity remainder of $g_0^{\frac{p-1}{q}}$ when divided by p. This is an integer since $q \mid p-1$. To ensure that $g \neq 1$, I continuously choose a random value for g_0 until this condition is met. To make this easiest, I define a function `rem(g0)` that works out g given g_0.

```
g_0 = 2;
rem = @(g0) feval(symengine, 'powermod', g0, (p - 1)/q, p);

while rem(g_0) == 1
    g_0 = sym(randi(p - 1)); % choosing a new random value of g_0
end
g = rem(g_0); % now we are sure that this will not be 1
```

I then calculate the last part of the digital signiature, y, given x.

```
x = sym('125001057271');
y = feval(symengine, 'powermod', g, x, p);
fprintf(y =
```

This results in a `y = 15180966959797319916674` as the output.

Part b

Here I determine whether a given signiature (r, s) is valid, given a value of H. They are given the following values.

```
H = sym('259989565575');
r = sym('89602878941');
s = sym('115571347129');
```

I first check if r and s are not less than p and q respectively:

```
if r >= p || s >= q
    fprintf(r or s are not in range);
```

If not, I proceed to use p, q and g to check a further condition. This condition depends on the integers u_1 and u_2. To solve for them I simply reuse the function `solveMultCongruence` that I defined earlier:

```
else
    u1 = solveMultCongruence(s, H, q);
    u2 = solveMultCongruence(s, r, q);
```

I now find the remainder of $g^{u_1} y^{u_2}$ on division by p. I work out these two exponents separately modulo p, then multiply the results modulo p, so that I can make use of MATLAB's efficient `feval` function.

```
    % by modular arithmetic, we can work out the remainders separately
    gRemainder = feval(symengine, 'powermod', g, u1, p);
```

```
    yRemainder = feval(symengine, 'powermod', y, u2, p);
    remainder = mod(gRemainder * yRemainder, p);
```

I finally check whether this remainder is congruent to r modulo q:

```
    if mod(remainder, q) == r
        fprintf('Signiature is valid!\n');
    else
        fprintf('Signiature is invalid, although r and s are in range.\n');
    end
end % end of initial if statement
```

The output is `Signiature is valid!`