Problem Solving by Computer MATH36032
Coursework 1

Question 1: Write a function `AppEM` such that `[p,q] = AppEM(N)` returns the best pair of integers p and q, such that $|\frac{p}{q} - \gamma|$ is smallest amongst all positive integers and $p + q$ is less than or equal to N. Record your result for $N = 2021$.

To write this function, I wanted to find a way of seaching through integers m and n in an efficient way to find the best pair p and q. To do this I assign a temporary best pair `[p, q]` which I compare the current pair of integers on each loop, `[m, n]`, against. If $|\frac{m}{n} - \gamma| < |\frac{p}{q} - \gamma|$ then I update the temporary best pair to `[p, q] = [m, n]`. If two best pairs are found, I want to take the pair with the lowest $p + q$. Notice that the values of m and n are strictly increasing on each loop. So if another best pair `[m, n]` is found, it will always be true that $m + n > p + q$, therefore we will not need to update our temporary best pair (ie. we do not need to write any extra code).

```
function [p, q] = AppEM(N)
% Approximates the Euler - Mascheroni constant by the ...
   rational number
% p /q, among all pairs of positive integers (p, q) such ...
   that p + q <= N.

emconst = 0.577215664901533;
p = 0;
q = 1; % initial values chosen so that p/q = 0

for n = 1:N-1
    for m = 1:N-n   % p + q <= N so p <= N - q
        if(abs((m/n)-emconst) < abs((p/q)-emconst))
            p = m;
            q = n;
        end
    end
end
```

```
>> [p,q] = AppEM(2021)
p =
   228
q =
   395
```

Question 2: Write a function `MyLuckynum` such that `MyLuckynum(N)` returns the smallest MyLucky number greater than or equal to N. Record your output for $N = 2021$.

For this question, I wrote a function that scans through a list of numbers, and performs a series of checks on each to decide whether or not it is a MyLucky number. I found three checks I wanted to perform and decided to write each as a seperate function which returns `true` or `false`, as I found it easier to express the final function `MyLuckynum` as an implementation of these components.

The first thing I checked was whether a given n has only odd prime factors. Observe that 2 is the only even prime, so we can rephrase the problem as: Does n have 2 as a prime factor? If not, it must have only odd prime factors. The MATLAB function `factor(n)` works in our favour here as the vector of prime factors it outputs is written in ascending order, so if 2 is a factor of n we will find it in the first entry of that vector.

```
function c = check1(n)
% checks if n has any even prime factors

p = factor(n);
c = false;

if(p(1)==2)
    c = true;
end
```

The second thing I checked was whether a given n has distinct prime factors. Verifying this was a simple case of scanning through the vector of prime factors of n generated by `factor(n)` and comparing each entry with the one after it to find any repeats.

```
function c = check2(n)
% checks if the factors of n are distinct

p = factor(n);
c = true;

for k=1:length(p)-1
    if(p(k)==p(k+1))
        c = false;
        break
    end
end
```

The third thing I checked was whether for all prime factors p of n, $p-1 \mid n-1$. To answer this, I needed to describe the notion of 'divides' in MATLAB. I wrote a function `divides(a,b)` to tell whether or not $a \mid b$, where we know $a, b \in \mathbb{Z}$ for our uses. It works on the idea that if $a \nmid b$, then $10 \times \frac{b}{a}$ will have non-zero residue mod 10. I used the MATLAB function `mod(x,y)` to find this residue.

```
function n = divides(a,b)
% Tests whether a divides b
n = true;
if(mod(10*b/a,10)~=0)
    n = false;
end
```

Now we can write third check by looking at each element p of the vector of prime factors of n and finding `divides(p-1,n-1)`.

```
function c = check3(n)
% checks if p-1 divides n-1 for each prime factor p of n

p = factor(n);
c = true;

for k=1:length(p)
    if(divides(p(k)-1,n-1)==false)
        c = false;
        break
    end
end
```

Finally, I wrote the `MyLuckynum(N)` function. I'm able to quickly verify whether each n is a MyLucky number using a simple `if` statement, and this was why I wrote the conditions for n to be MyLucky number as the three checks above. I've also included an `if` statement to discard any prime numbers from the search before going through all of the checks, to speed up the computation. Here is the function:

```
function n = MyLuckynum(N)
% Finds the smallest MyLucky number that is greater than
% or equal to N.

n = 2; % want to start at 3 since 1 and 2 are not MyLucky
ML = false; % want one of the conditions for the while loop ...
   to be that it runs until we find a MyLucky #

while(ML==false || n<N)
    % want the first n such that (ML == true & n >= N),
    % so we set the loop to run while the negation (ML == ...
       false | n < N) holds.

    n = n+1;
    ML = true;

    if(isprime(n)==true) % primes are not MyLucky
        ML = false;
        continue
    end

    if(check1(n) == true || check2(n) == false || check3(n) ...
       == false)
        ML=false;
        continue
    end
end
```

```
>> MyLuckynum(2021)
ans =
        2465
```

Question 3: Find the beautiful square number n, such that n^2 is closest to 360322021.

When I read this question I decided to first try and make a list of all beautiful square numbers. I needed to write a function that would search a list of integers and find any beautiful square numbers n. That is, any n such that n^2 consists of all nine non-zero digits exactly once. So how many numbers should we search? Simply, we want to search up to the smallest n such that $n^2 \geq 987654321$, the largest possible number consisting of all nine non-zero digits exactly once. To find this upper limit, I wrote a function `searchlimit(N)`:

```
function n = searchlimit(N)
% finds the smallest n such that n^2 >= N
n = 0;
while( n^2 < N)
    n = n+1;
end
```

```
>> searchlimit(987654321)
ans =
       31427
```

Next I wrote a function `listBeautisqnum()` to search through all the nine digit square numbers up to 31427^2 and find beautiful squares. To identify a beautiful square, I chose to convert each number n^2 into a vector of its digits then used the MATLAB function `sort(A)` to arrange the elements of the vector in ascending order. To convert each number into digits I used the `num2dig(n)` function written in the lab demonstrations which I have also included below (with some added comments to show how it works). Now, if n is a beautiful square number then the sorted vector of digits of n^2 will equal the vector $(1, 2, 3, 4, 5, 6, 7, 8, 9)$.

```
function [dig] = num2dig(N)
% converts a number into a vector of its digits
dig=[];
n=N; % so as not to overwrite the original variable N
while(n>0)
    dig=[rem(n,10) dig]; % eg. rem(1234,10)=4
    n = floor(n/10); % eg. floor(1234/10)=floor(123.4)=123
end
```

```
function n = listBeautisqnum()
% returns a vector of all beautiful square numbers.
n = [];
for k = 10000:31427 % 10000^2 is the smallest 9 digit number
    if(sort(num2dig(k^2))==[1:9])
        n = [k, n];
    end
end
```

```
>> listBeautisqnum
ans =
  Columns 1 through 9
       30384       29106       29034       27273       27129 ...
                   26733       26409       25941       25572
  Columns 10 through 18
       25059       24807       24441       24276       24237 ...
                   23439       23178       23019       22887
  Columns 19 through 27
       20316       19629       19569       19377       19023 ...
                   18072       15963       15681       14676
  Columns 28 through 30
       12543       12363       11826
```

Finally, I wrote the function `Beautisqnum(N)` to find the value of n such that n^2 is closest to N. What I have written searches through elements of a given vector, and updates the value of n each time it finds a new 'best' value. Now that we know all the beautiful square numbers, we know in particular that the smallest such number is 11826, which I used as the initial closest value. Additionally, we can narrow down the search in the final function to just 30 numbers, which is a huge increase

in efficiency! Note that in the code I have written below, I call `listBeautisqnum()` at the start, but I have only written this for brevity and in practice it would be faster (computationally) to assign to a variable the pre-calculated vector of beautiful squares rather than having to generate the list every time.

There are cases when one can find two nearest beautiful squares. For example, I found that 250355565 is equidistant to 15681 and 15963 after a brief search. To account for this, I added an `elseif` statement which will output the two numbers as a vector when this happens.

```
function n = Beautisqnum(N)
% Returns n, such that n^2 is a perfect square number
% with all nine distinct digits from 1 to 9,
% which is closest to the input N, another nine digits number
% (can be assumed to be between 10^8 and 10^9 - 1)

b = listBeautisqnum();
n = 11826;

for k = 1:length(b)
    if( abs(N-b(k)^2) < abs(N-n.^2) ) % compares the ...
        distance from N of b(k) to that of the current n
         n = b(k);
    elseif( abs(N-b(k)^2) == abs(N-n.^2) )
         n = [b(k) n];
    end
end
```

```
>> Beautisqnum(360322021)
ans =
       19023
```