It's all about Primes

By: Hussain Arif, Alireza Zandi, Keith De Souza, Muhammad Danish, Ameertharaj Nagesparan

Task 1

RSA Cryptosystem

RSA Cryptosystem

- Published in 1978 by Ron **Rivest**, Adi **Shamir**, and Len **Adleman** at MIT.
- Involves the use of **prime numbers** to form **public key** and **private key**.
- RSA is used for secure communication, digital signatures, secure file transfer, password encryption and more.
- Most widely accepted and implemented asymmetric-key encryption.
- **E.g.:** Alice wants to send a message to Bob. Bob has a private decryption key while they both have the public encryption key. Alice encrypts the message using the public key and sends it over to Bob. It is then decrypted by Bob using his private key.

RSA Key Generation

Example:

1. Pick 2, ideally large, prime numbers **p and q**
   ~ p and q are kept secret

   p=3, q=11

2. Calculate **$n=p\cdot q$**
   ~ N is part of the public key

   n = 33

3. Calculate **$\phi(n)=(p-1)\cdot(q-1)$**
   ~ $\phi(n)$ is kept secret

   $\boldsymbol{\phi(n)} = (3-1)\cdot(11-1) = \mathbf{20}$

4. Pick e such that:
   $\gcd(\phi(n),e) = 1$ and $2 < e < \phi(n)$
   ~ e is part of the public key

   $\gcd(20,e) = 1$
   Pick e = 3

5. Determine d using the Extended Euclidean Algorithm:
   $d\cdot e \equiv 1 \bmod(\phi(n))$
   ~ d is the private key

   $d\cdot 3 \equiv 1 \bmod (20)$
   d=7

RSA Encryption and Decryption

Public key is $\mathbf{K_{Pub}= (n,e)}$ & Private key is $\mathbf{K_{Prv}= (d)}$.
To encrypt a message M:

	Encryption		**Decryption**	
Plaintext		**Ciphertext**		**Plaintext**
M →	$\mathbf{M^e \bmod n = C}$	→ **C** →	$\mathbf{C^d \bmod n = C}$	→ **M**

Example:

The Public key is $\mathbf{K_{Pub}= (33,3)}$. The Private key is $\mathbf{K_{Prv} = (7)}$:
To encrypt a message M (equivalent to 4_{10}):

	Encryption		**Decryption**	
Plaintext		**Ciphertext**		**Plaintext**
4 →	$\mathbf{4^3 \bmod 33 = 31}$	→ **31** →	$\mathbf{31^7 \bmod 33 = 4}$	→ **4**

Importance of Prime Numbers in RSA

- **Uniqueness of Prime Numbers**
  - Prime numbers only have two factors, 1 and itself, making them unique.

- **Security Properties**
  - High difficulty of factoring large integers
  - No known method for large integer factorization in a feasible time frame.

- **Euler's Totient function**

  RSA uses Euler's totient function(**ɸ**) to compute the public and private keys.
  - counts the number of positive integers coprime to a given number

- **Introduced Public Key Ciphers**

  The use of this prime numbers based encryption method allows for encryption keys to be discussed and decided upon without a secure channel, as it is public.

  ~This was however not the case in the past when symmetric key ciphers were used.

Task 2

Two prime number-related functions

2.1: Code a function to check if a number is prime

```
1 function [result] = isNumberPrime(n)
2 if(n==2 || n==3)
3     result = true;
4     return
5 end
6
7 if(mod(n,2)==0)
8     result = false;
9     return
10 end
11
12     for i=2:sqrt(n)
13         if mod(n, i) == 0
14             result = false;
15             return
16         end
17     end
18     result = true;
19 end
```

A breakdown of this code:

Lines 3 to 11: We check if the numbers 2, 3 and all even numbers are prime.
We are doing this for optimization purposes. This way, our program doesn’t enter the loop when an input is presented.

Lines 13-19: Start from 2 and to the square root of the input to check if the number is divisible by any number in this range.
If this condition is not met, we have found a prime.

Measuring performance

Command Window

```
>> tic; isprime(1000); toc;
Elapsed time is 0.000827 seconds.
>> tic; isNumberPrime(1000); toc;
Elapsed time is 0.000248 seconds.
>> tic; isprime(15104393); toc;
Elapsed time is 0.003423 seconds.
>> tic; isNumberPrime(15104393); toc;
Elapsed time is 0.000592 seconds.
>> tic; isprime(15104393987); toc;
Elapsed time is 0.000882 seconds.
>> tic; isNumberPrime(15104393987); toc;
Elapsed time is 0.000475 seconds.
>>
```

```
>> isprime(15104393)

ans =

  logical

   0

>> isNumberPrime(15104393)

ans =

  logical

   0
```

```
>> isprime(1510439387)

ans =

  logical

   1

>> isNumberPrime(1510439387)

ans =

  logical

   1

>>
```

- In many cases, our function is just as fast MATLAB's inbuilt ***'isprime'*** function with identical accuracy.
- However, it is important to remember that the ***isprime*** function uses Miller-Rabin's algorithm and other software tricks, which means that it is faster on larger inputs.
- This will be useful for us in upcoming tasks, where performance is important.

2.2: Code a function to output the first *n* prime numbers

```
1 function [result] = ReturnFirstNPrimes(n)
2     result = [];
3     counter = 3;
4     primeNumber= 5;
5     result(1)=2;
6     result(2)=3;
7     while size(result)<n
8         if isNumberPrime(primeNumber)
9             result(counter) = primeNumber;
10            counter= counter+1;
11        end
12        primeNumber= primeNumber+2;
13    end
14 end
```

A breakdown of this code:

Lines 7 to 11: Until we have *n* elements in the ***result*** array, keep on appending prime numbers into *result*. Notice that here, we are reusing the ***isNumberPrime*** function. This allows for modular and cleaner code.

Line 12: Increment the counter by 2 and then check if this number is a prime. This is because, according to our findings, it was more efficient to skip the counter by 2 intervals. This will result in faster, and more efficient code.

Measuring performance

```
>> ReturnFirstNPrimes(10)

ans =

     2     3     5     7    11    13    17    19    23    29

>> tic; ReturnFirstNPrimes(10^2); toc;
Elapsed time is 0.000676 seconds.
>> tic; ReturnFirstNPrimes(10^4); toc;
Elapsed time is 0.155513 seconds.
>> tic; ReturnFirstNPrimes(10^6); toc;
Elapsed time is 191.062131 seconds.
```

For small inputs, we get our outputs under one second. However, as the input gets larger, the required time grows exponentially.

Task 3

List of prime numbers

3.1: Showing the nth Prime Number

```
function nthPrime = task3a(n)
    ......
    % The vector is initialised
    % pVector(1)==2 & pVector(2)==3
    ......
    k=1;
    count=3;
    while count<=n
        possiblePrime=k*6-1; % as every prime is in this form %

        if(is_prime3(possiblePrime,pVector))
            pVector(count)=possiblePrime;
            count=count+1;
        end
        if count>n
            break;
        end
        possiblePrime=k*6+1;
        if(is_prime3(possiblePrime,pVector))
            pVector(count)=possiblePrime;
            count=count+1;
        end
        k=k+1;
    end
    nthPrime=pVector(n)
end
```

- The is_prime function is optimized. To check if a number n is prime:
    - n=6k+1 or n=6k-1 *(Unless n=2 or n=3)*, k ∈ N
    - n is not composed of other primes ∴ check if n is a not a multiple of all prime numbers less than √n with the help of pVector that is passed as a parameter.
- If prime, add n to the pVector.
- Return the last element in the array to get the nth prime.

```
function prime = is_prime3(n,pVector)
    prime = true;
    count = 3;

    while pVector(count)<=sqrt(n) && pVector(count)~=0
        if(mod(n,pVector(count))==0)
            prime = false;
            return
        end
        count = count+1;
    end
end
```

3.1: Showing the nth prime number

i) The 99th prime number is 523

```
>> task3a(99)
tic;task3a(99);toc

ans =

   523

Elapsed time is 0.001143 seconds.
```

ii) The 9999th prime number is 104,723

```
>> task3a(9999)
tic;task3a(9999);toc

ans =

      104723

Elapsed time is 0.028684 seconds.
```

3.2: Finding the sum of all prime numbers

```
function sum = task3b(n)
    format long
    pVector=[];
    pVector(1)=2; % if n = 1 %
    if n==1
        return
    end
    pVector(2)=3; % if n = 2 %
    if n==2
        return
    end
    pVector(3)=0;
    k=1;count=3;

    sum=5;
    possiblePrime=k*6-1;
    while possiblePrime<=n
        if(is_prime3(possiblePrime,pVector))
            pVector(count)=possiblePrime;
            sum=sum+possiblePrime;count=count+1;
        end
        possiblePrime=k*6+1;
        if possiblePrime>n
            return;
        end
        if(is_prime3(possiblePrime,pVector))
            pVector(count)=possiblePrime;
            sum=sum+possiblePrime;count=count+1;
        end
        k=k+1;
        possiblePrime=k*6-1;

    end
end
```

- In this function, all prime numbers before the upper bound are stored in a vector *pVector.*
- With the help of while loop, we update the value of sum in each iteration.

3.2: Finding the sum of all prime numbers

i) The sum of all the prime numbers below 1,500 is 165,040

```
>> task3b(1500)
tic;task3b(1500);toc

ans =

      165040

Elapsed time is 0.001756 seconds.
```

ii) The sum of all the prime numbers below 1,500,000 is 82,074,443,256

```
>> task3b(1500000)
tic;task3b(1500000);toc

ans =

     8.207444325600000e+10

Elapsed time is 0.924969 seconds.
```

Task 4

Prime Number Theorem

Prime Number Counting Function

```
function primeCountVector = primeCountFunction(n)

    primeCountVector=[];
    primeCountVector(1,n)=0;

    count=0;
    for k=1:n
        if isprime(k)
            count=count+1;
        end
        primeCountVector(k)=count;
    end
end
```

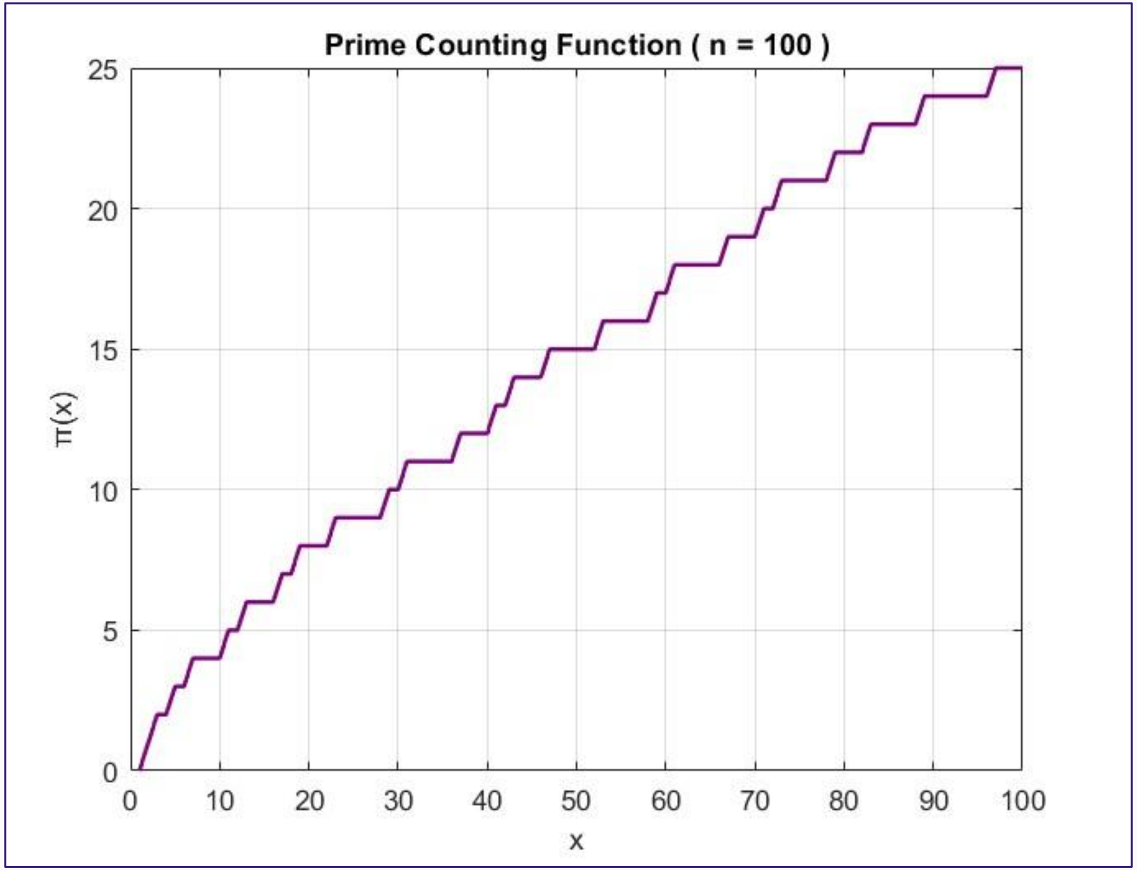
- The code initializes an entry vector called ‘primeCountVector’ and iterates from 1 to n.
- The primality of a number k is checked by using the function defined in task 2.
- The count is incremented and stores in ‘primeCounterVector’ if a number is prime.

4.1: Plotting the Prime Counting Function

```
function task4a(n)
   x = 1:1:n;
   y = primeCountFunction(n);

   h=plot(x,y);

   h.LineWidth = 1.5;

   h.Color = [128/255 0/255 128/255];

   grid on
   xlabel('x'); ylabel('π(x)');

   title('Prime Counting Function ( n = 100 )')
end
```

- In this function, the prime counting function $\pi(x)$ is plotted for $1 \leq x \leq n$
- An x-axis array from 1 to n is created and the corresponding y-axis values are assigned.
- The plot is further customized with a specific line width, colour, grid lines and, and axis labels.

4.1: Plotting the Prime Counting Function

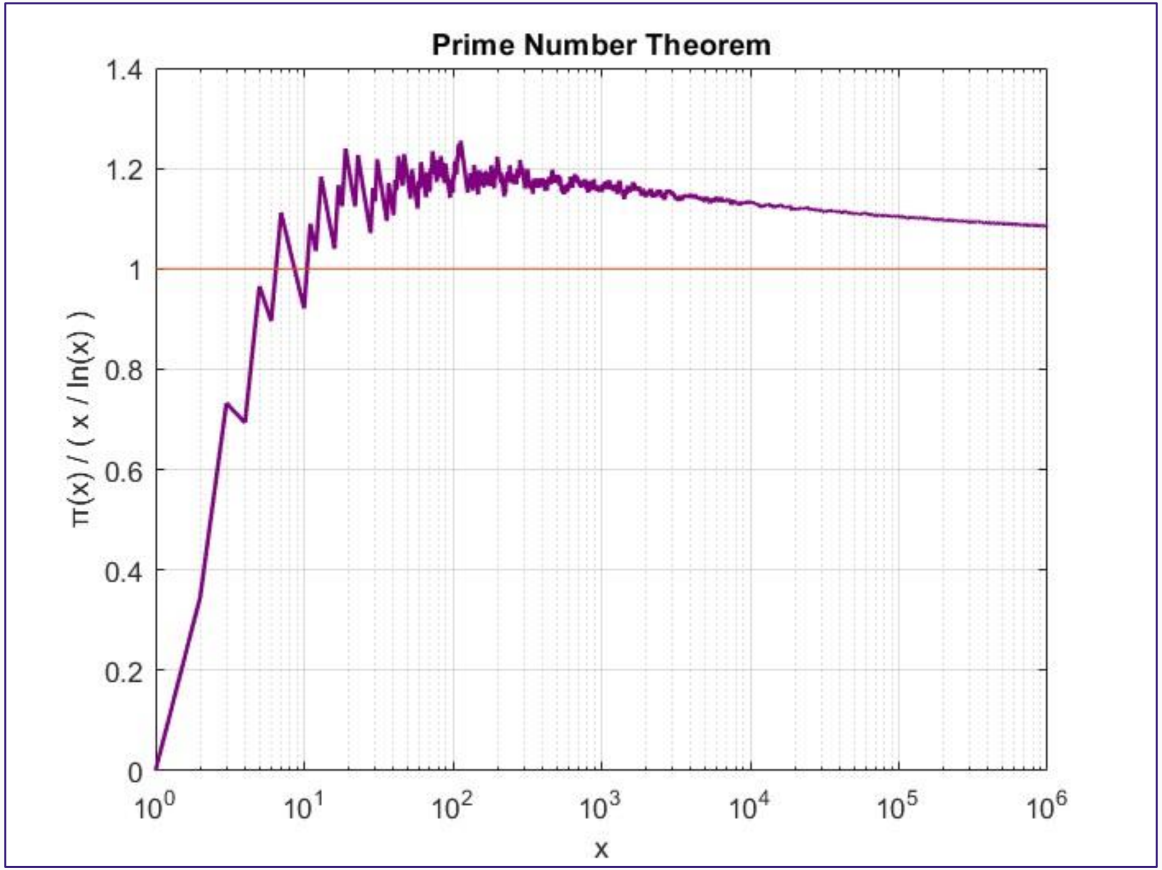


4.2: Prime Number Theorem

```
function task4b()

    n=1000000;

    x=1:1:n;
    pi=primeCountFunction(n);
    lnx=log(x);

    h=semilogx(x,(pi)./(x./lnx));
    hold on
    semilogx(x,ones(1,n))

    h.LineWidth = 1.5;

    h.Color = [128/255 0/255 128/255];
    xlabel('x'); ylabel('π(x) / ( x / ln(x) )');

    title('Prime Number Theorem')
    hold off
    grid on


end
```

- A graph is generated by plotting the ratio of pi(x) to x/ln(x) against x.
- The x-axis has a logarithmic scaling
- Customization is carried out like in the last graph.
- ‘hold on’ and ‘hold off’ commands are used to retain the existing graph and adding additional elements to it.

4.2: Prime Number Theorem

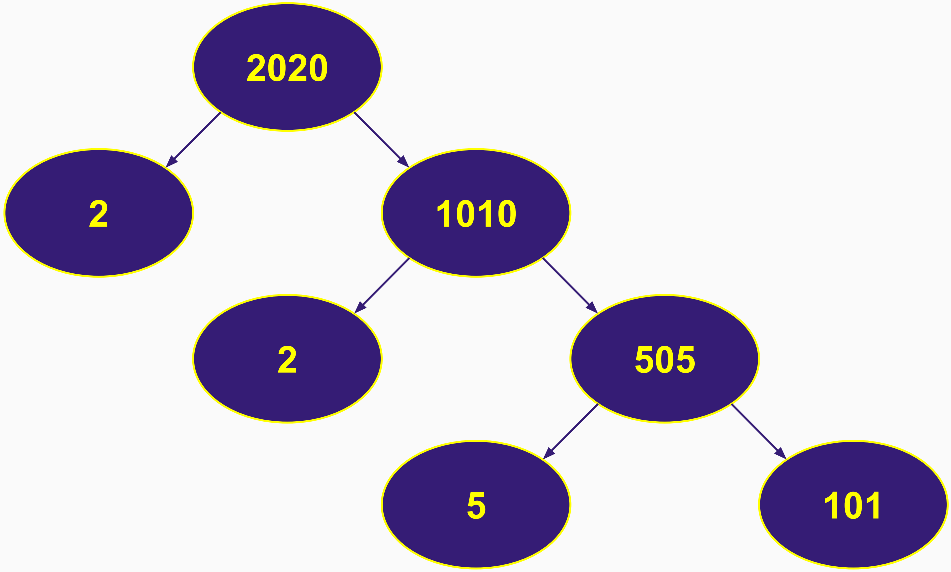


Task 5

Largest prime factor

How to get prime factors?

- Make a prime factor tree!



- Factorize root node and split into two nodes, where the left node is a prime number.
- If the right node is not a prime number, it can be factorized again (repeat first step using recursion)
- If the right node is a prime number, it is the largest prime factor.

```
function pMultiple=task5(n,m)

    pMultiple=n;
    for k=m:sqrt(n)
        ....
        % k should be prime, else continue %

        if mod(n,k)==0

            pMultiple=n/k;
            % k is left node           %
            % pMultiple is right node %

            if(is_prime5(pMultiple))
                % returns largest prime factor %
                return
            end

            pMultiple=task5(pMultiple,k);

            return
        end
    end
end
```

```
function prime = is_prime5(n)
    % Miller Rabin Primality Test %
    .....% If even return false %

    prime=true;
    d=n-1;
    m=0;

    while mod(d,2)==0
        d=d/2;
        m=m+1;
    end
    % 2^m + d = n-1 %

    a=[2,3,5,7,11];
    % Enough to check n < 2,152,302,898,747 %

    for k=1:5
        for s=0:m-1
            % If a(k)^((2^s)*d) mod n ~= 1  OR   %
            %    a(k)^((2^s)*d) mod n ~=-1        %
            % for ALL 0 <= s < m: n is composite %
            result=squareMultiply(a(k),(2^s)*d,n);

            if(result~=n-1 && result~=1)
                prime=false;
            else
                prime=true;
                break;
            end
        end

        if(~prime)
            return
        end
    end
end
```

```
function result = squareMultiply(base,exponent,modulus)
    % Difficult to do base^exponent, especially %
    %      when dealing with large numbers      %

    % Binary Exponentiation %
    % Allows to quickly compute large positive %
    %       integer powers of a number.        %

    bin = dec2bin(exponent);
    result = base;

    for k= 2:size(bin,2)

        %square
        result=mod(result^2,modulus);

        %multiply
        if bin(k) == '1'
            result = mod(result*base,modulus);
        end

    end
end
```

Witness Numbers (and the truthful 1,662,803) - Numberphile

Square & Multiply Algorithm - Computerphile

Largest Prime Factor

a) Largest Prime Factor of 2,287,946:

```
>> task5(2287946,2)
tic;task5(2287946,2);toc;

ans =

        2797

Elapsed time is 0.000496 seconds.
fx >>
```

b) Largest Prime Factor of 565,499,313,531:

```
>> task5(565499313531,2)
tic;task5(565499313531,2);toc;

ans =

     8038027

Elapsed time is 0.000714 seconds.
fx >>
```

Thank you.