Chapter one

Background and literature review

1.1: Background:

In this section we will explain briefly how the new data type had been defined, the using of double linked list, the operators that had been overloaded and the implementation of this new data type.

1.1.1: Defining new data type:

Our research concerned with defining new data type, we called this type of number a *bigint* (for big integer). The *bigint* class (we will use the C++ concept of a class which is a structure that can hold data and functions that operate on that data) implements integers with unlimited size.

It is conceptually simplest to implement our new data type in decimal and represent each integer as an integer of base 1000.

1.1.2: Double linked list:

Double linked list is our storage structure, it is a linked list, in which each node contains a pointer to a next node and a pointer to a previous element in the list.

We used double linked list because in a standard linked list, you can only traverse the list in one direction. It may be useful however, to be able to go both forwards and backwards when traversing a linked list. In order to allow this, we need each node to contain not one, but two references to nodes one to the next node in the list and one to the previous node in the list.

Each number will be represented in a separate list, our base is 1000 so, their will be three digits per node in the list.

1.1.3: Overloading operators and defining functions: -

C++ allows the user to overload most of the operators to work effectively in a specific application. C++ does not allow the user to create new operators. Most of the existing operators can be overloaded to manipulate class objects.

1.1.3.1: Stream operators (Input/Output):

The istream operator >> and the ostream operator << are overloaded.

1.1.3.2: Arithmetical Operators:

Arithmetic on large integers is often necessary in general and in cryptography especially. So, *bigint* support the classical algorithms of overloading the addition, subtraction, multiplication and division giving a quotient and remainder operators and they can be used in exactly the same way as for machine types in C++ (e.g. int).

1.1.3.3: Relational Operators:

The binary operators ==, !=, >=, <=, >, < and the unary operator ! (Comparison with zero) are overloaded and can be used in exactly the same way as for machine types in C (e.g. int).

Functions:

Various numbers of functions were declared which are:

-Functions related to number theory such like the Extended Euclidean algorithm and an algorithm for modular arithmetic.

-Functions related to RSA algorithm such like random function to generate large integer numbers, RSMA Function, encrypt and decrypt functions.

1.1.4: Case study:

After implementing *bigint* data type, arithmetic operators, relational operators, functions related to number theory and others functions, it is important to test our new data type.

We find that RSA represent an excellent case study to use *bigint* class. Therefore there are too many functions concerned to RSA function must be declare, just like random functions, RSMA function and encryption and decryption functions.

1.2: Literature Review:

Integers are whole numbers, and never have a decimal point. They are broken down into several types, so let's see BIGINT data type in several languages. [20]

Remember that a computer operates on *binary* data. a 1 or a 0 is stored in a *bit*, and eight bits is a *byte*. That means that one byte can only hold a finite number of integer values. If you want a bigger integer, you need another byte. Therefore, a MySQL TINYINT has a storage size of 1 byte, meaning that it can store values from -128 to 127. One of the bits is used to store the sign (positive or negative). If you declare a TINYINT as UNSIGNED, it means that you want all eight bits to be used for storing numbers. That, in turn, means that an UNSIGNED TINYINT can hold values from 0 to 255. [20]

1.2.1: SQL:

The *bigint* data type is supported where integer values are supported. However, *bigint* is intended for special cases where the integer values may exceed the range supported by the int data type. The int data type remains the primary integer data type in SQL Server. [18]

BIGINT provides 8 bytes of storage for integer values. Integer (whole number) data from -2^^{63}(-9,223,372,036,854,775,808) through 2^$^{63-1}$(9,223,372,036,854,775,807). Storage size is 8 bytes.

Functions will return ***bigint*** only if the parameter expression is a ***bigint*** data type. SQL Server will not automatically promote other integer data types (**tinyint**, **smallint**, and **int**) to ***bigint***

***1.2.2:* JAVA:[11]**

Class BigInt

```
Java.lang.Object
   |
   +----Java.lang.Number
           |
             +----BigInt
```

Final class **BigInt**
extends Number
Multiple precision integer arithmetic.

The BigInt class implements integers with unbounded precision. BigInts support the Classical Algorithms of addition, subtraction, multiplication and division giving a quotient and remainder.

BigInts are a subclass of java.lang.Number so they support the conversion operations intValue(), doubleValue etc., just like the other forms of “boxed” number (classes Integer, Float, Long and Double). It is a pity that “Integer” was used as the name for a boxed int. “Int” would have been a more regular name, and it would have left the name `Integer' open for multiple precision integers which better model the mathematical notion of integer.

JAVA has its data type and the following table explain the basic data type available in JAVA.

Table 2: JAVA basic data type:[35]

Type	**Identifier**	**Bits**	**Values**
Character	char	16	Unicode 2.0
8-bit signed integer	byte	8	-128 to 127
Short signed integer	short	16 (2 bytes)	-32768 to 32767
Signed integer	int	32 (4 bytes)	-2,147,483,648 to +2,147,483,647
Signed long integer	long	64 (8 bytes)	maximum of over 10^{18}
Real number (single precision)	float	32 (4 bytes)	Maximum of over 10^{38} (IEEE 754-1985)
Real number (double precision)	double	64 (8 bytes)	Maximum of over 10^{308} (IEEE 754-1985)
Boolean	Boolean		true or false

1.2.3: Fortran:[50]

Table 3: Represent FORTRAN Data Types

Fortran Data Type	**Format**	**Range**
INTEGER	2's complement integer	-2^{31} to $2^{31}-1$
INTEGER*2	2's complement integer	-32768 to 32767
INTEGER*4	same as INTEGER	

INTEGER*8	same as INTEGER	-2^{63} to $2^{63}-1$
LOGICAL	same as INTEGER	true or false
LOGICAL*1	8 bit value	true or false
LOGICAL*2	16 bit value	true or false
LOGICAL*4	same as INTEGER	true or false
LOGICAL*8	same as INTEGER	true or false
BYTE	2's complement	-128 to 127
REAL	Single-precision floating point	10^{-37} to 10^{38} (1)
REAL*4	Single-precision floating point	10^{-37} to 10^{38} (1)
REAL*8	Double-precision floating point	10^{-307} to 10^{308} (1)
DOUBLE PRECISION	Double-precision floating point	10^{-307} to 10^{308} (1)

COMPLEX	See REAL	See REAL
DOUBLE COMPLEX	See DOUBLE PRECISION	See DOUBLE PRECISION
COMPLEX*16	Same as above	Same as above
CHARACTER*n	Sequence of n bytes	

(1) Approximate value.

From this table we notice that Fortran doesn’t support *bigint* data type.

1.2.4: C/C++:[51]

Table 4: *C/C++* Data Types

Data Type	Size (bytes)	Format	Range
unsigned char	1	ordinal	0 to 255
[signed] char	1	two's-complement integer	-128 to 127
unsigned short	2	ordinal	0 to 65535

[signed] short	2	two's-complement integer	-32768 to 32767
unsigned int	4	ordinal	0 to 2^{32} -1
[signed] int [signed] long int	4	two's-complement integer	-2^{31} to 2^{31}-1
unsigned long int	4	ordinal	0 to 2^{32}-1
[signed] long long [int]	8	two's-complement integer	-2^{63} to 2^{63}-1
unsigned long long [int]	8	ordinal	0 to 2^{64}-1
Float	4	IEEE single-precision floating-point	10^{-37} to 10^{38} (1)
double	8	IEEE double-precision floating-point	10^{-307} to 10^{308} (1)
long double	8	IEEE double-precision floating-point	10^{-307} to 10^{308} (1)
bit field(2)	1 to 32	ordinal	0 to 2^{size}-1, where size is the number

(unsigned value)	bits		of bits in the bit field
bit field(2) (signed value)	1 to 32 bits	two's complement integer	-2^{size-1} to $2^{size-1}-1$, where size is the number of bits in the bit field
pointer	4	address	0 to $2^{32}-1$
Enum	4	two's complement integer	-2^{31} to $2^{31}-1$

(1) Approximate value.
(2) Bit fields occupy as many bits as you assign them, up to 4 bytes, and their length need not be a multiple of 8 bits (1 byte).

From this table we notice that C/C++ doesn't support *bigint* data type.

1.2.5: Pascal:[40]

Table 5: integer data types available in Pascal:

Type	Minimum	Maximum	Format
Integer	–2147483648	2147483647	signed 32-bit
Cardinal	0	4294967295	unsigned 32-bit
Shortint	–128	127	signed 8-bit
Smallint	–32768	32767	signed 16-bit
Longint	–2147483648	2147483647	signed 32-bit
Int64	–2^63	2^63–1	signed 64-bit
Byte	0	255	unsigned 8-bit
Word	0	65535	unsigned 16-bit

Longword	0	4294967295	unsigned 32-bit

1.2.6: Delphi:[37]

Delphi provides many different data types for storing numbers. Your choice depends on the data you want to handle. Our attention is on integer data type, table below demonstrate the integer data type available in Delphi language.

Table 6: Integer data type in Delphi:

Byte	0 to 255
ShortInt	-127 to 127
Word	0 to 65,535
SmallInt	-32,768 to 32,767
LongWord	0 to 4,294,967,295
Cardinal	0 to 4,294,967,295
LongInt	-2,147,483,648 to 2,147,483,647
Integer	-2,147,483,648 to 2,147,483,647
Int64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

1.2.7: Visual Basic 6 and Visual Basic .NET:

Integer variables are stored as signed 32-bit (4-byte) integers ranging in value from -2,147,483,648 through 2,147,483,647.[19]

The Integer data type provides optimal performance on a 32-bit processor, as the smaller integral types are slower to load and store from and to memory.[19]

You can convert the Integer data type to Long, Single, Double, or Decimal without encountering a System. Overflow Exception error.[19]

Visual Basic 6 dose not support *bigint* data type

The numeric data types supplied by Visual Basic 6 are: [22]

- Byte
- Integer
- Long
- Single
- Double
- Currency
- Decimal

The numeric variables supported by VB .NET are: [22]

- Byte (System.Int)
- Short (System.Int16)
- Integer (System.Int32)
- Long (System.Int64)
- Single (System.Single)
- Double (System.Double)
- Decimal (System.Decimal)

1.2.8: C#:[36]

Data Types and Variables are the core part of C# programming language. It represents how to express numbers, characters, strings and other values in real code.

Different Types of Data Types

The .NET Framework provides lot of data types which you can use for developing applications. Since these data types are provided by the Common Language Runtime (CLR), all .NET languages such as C#, Visual Basic .NET can take advantage of them. A complete list of all data types and its range values are given in Table 1.

Table 7: List of Data Types

Data Type Prefix	**.NET Data Type**	**Min Value**	**Max Value**
Sbyte	System.Sbyte	-128	127
Byte	System.Byte	0	255
Short	System.Int16	-32,768	32,767
Ushort	System.UInt16	0	65,535
Int	System.Int32	-2,147,483,648	2,147,483,647
Uint	System.UInt32	0	4,294,967,295
Long	System.Int64	-9,223372,036,854,775,808	9,223372,036,854,775,808
Ulong	System.UInt64	0	18,446,744,073,709,551,615
Char	System.Char	0	65,535
Float	System.Single	1.5 x 10-45	3.4 x 10 38
Double	System.Double	5.0 x 10-324	1.7 x 1010308
Bool	System.Boolean	False (0)	True (1)
Decimal	System.Decimal	1.0 x 10-28	7.9 x 1028

1.2.9: MATLAB:

The most common numeric data types in MATLAB include 8, 16, 32, and 64 bits in length, signed and unsigned integers, and single- and double numbers. Enables you to manipulate integer quantities in a memory efficient manner. [14]

Chapter Two

Designing *bigint* class and overloading operators

Introduction:

In this chapter we will describe the need for the new data type (*bigint*), the storage structure we used to represent *bigint*, defining big integer numbers, the designing of *bigint* class and the overloading and dealing with the basic arithmetic and relational operations for multiple-precision integers.

2.1: The need of large integer:

The integer basic data type provided by the C/C++ language to represent integers has many limitations: It is limited in the smallest and largest integer that it can represent, as well as we know that computers are powerful, but they are not infinitely precise in their calculations. For example, the computers cannot represent the integer numbers with more than about 20 digits in them. This is an other limitation of the computer hardware, so if we want to do a simple arithmetic operation, say multiplication of two numbers each of them of 30 digits, we are in trouble. [55]

Actually, the need to represent big numbers exactly (and big integers, in particular) also arises in computer security. Modern techniques of cryptography rely on the ability to determine very large integers (integers with hundreds of digits), and approximations just won't work in this case. Here, we need arbitrary-precision arithmetic.

This is just a fancy term meaning exact arithmetic for numbers of any size. For this project, we had created a C++ class that represents just such arbitrarily sized numbers. We'll call this class or this type of number a *bigint* (for big integer).

The new data type allows a programmer to declare and use integers of arbitrary length. We could support the basic arithmetic operations and relational operations of this data type. We could also be able to read and write *bigint* variables from and to stdin and stdout respectively.

A programmer should be able to use *bigint* in a C++ program just like the type int as we will see.

The data type *bigint* is important to do a multiple precision arithmetic routines written in C++ to carry out the usual large natural number calculations required in cryptography calculations.

2.2: Linked List: [9]

A linked list is an algorithm for storing a list of items. It is made of any number of pieces of memory (nodes) and each node contains whatever data you are storing along with a pointer (a link) to another node. By locating the node referenced by that pointer and then doing the same with the pointer in that new node and so on. You can traverse the entire list.

Because a linked list stores a list of items, it has some similarities to an array. But the two are implemented quite differently. Any array is a single piece of memory while a linked list contains as many pieces of memory as there are items in the list. Obviously, if your links get messed up, you not only lose part of the list, but you will lose any reference to those items no longer included in the list (unless you store another pointer to those items somewhere).

Some advantages that a linked list has over an array are that you can quickly insert and delete items in a linked list. Inserting and deleting items in an array requires you to either make room for new items or fill the "hole" left by deleting an item. With a linked list, you simply rearrange those pointers that are affected by the change. Linked lists also allow you to have different –sized nodes in the list. Some disadvantages to linked lists include that they are quite difficult to store. Also, you cannot immediately locate, say, the hundredth element in a linked list the way you can in an array. Instead, you must traverse the list until you've found the hundredth element.

A linked list like I've describe above, where each item has a pointer to the next item in the list, is called singly linked list. To implement this list, you would also want to store a pointer to the first item in the list (the head), which you would use to access the other items. However, some operations are awkward with a singly linked list. For example, to remove an item, you may need to traverse the entire list to locate the item that came before the item you are removing in order to modify its NEXT pointer. For this reason, many linked lists are implemented as a doubly linked list. In a doubly linked list, each item contains a pointer to both the next and the previous item in the list. Because you may want to traverse the list in reverse order, you would probably want to store the last item in the list (the tail) in addition to the first item.

2.3: Defining numbers:

We used double-linked lists to implement integers of unlimited size. We would implement integers, and the arithmetic operators we used to implement our big integers are addition, subtraction, multiplication, division and modulation we could also implement the relational operator >, <, <=, <=, ==, != . The program stored the integer three digits per node.

When you enter your number you should tack in your account that it is of base 1000 that is mean there are three digits at each node and if you entered one digit or two digits then the remainder digits will be of zeros.

The blocks of your number should be separated by space, for example, if you want to read the number 123456789, you should follow the following steps:

1- Enter 123
2- Enter a space
3- Enter 456
4- Enter space
5- Enter 789
6- Enter a space
7- At the end of your number you must enter a negative number to make sure it is the end of the number.

The number will appear: 123 456 789

And if you want to enter 123004056 then:

1- Enter 123
2- Enter space
3- Enter 4
4- Enter space
5- Enter 56
6- Enter space
7- Enter a negative number

We notice that if we enter 4 or 004 it is the same because it will push zeros to fill the three digits.

Radix representation: [23]

Positive integers can be represented in various ways, the most common being *base* 10 (*decimal representation*). For example, a = 123 base 10 means $a = 1*10^2+2*10^1+3*10^0$.

For machine computations, *base* 2 (*binary representation*) is preferable. If a = 1111011 base 2, then a = $2^6 + 2^5 + 2^4 + 2^3 + 0 * 2^2 + 2^1 + 2^0$.

In general, if *b* is the base (radix), we write a number in the numeral system of base (radix) *b* by expressing it in the form
$a_1b^k + a_2b^{k-1} + a_3b^{k-2} + ... + a_{k+1}b^0$ and writing the digits ($a_1a_2a_3$ ... a_{k+1}) in order. The digits are natural numbers between *0* and *b-1*, inclusive.

The new data type used radix *b*, where *b* is equal to 1000 if a=123 456 789, then a=$123*1000^2 + 456 * 1000^1 + 789 * 1000^0$

2.4: Designing A *bigint* Class:

Here are some concepts we should know them when designing our class.

2.4.1: Class Fundamentals: [41]

A *class* defines a new data type. Or rather, redefines an old data type (the struct). It is a set of variables and functions that can be accessed by the other variables and functions within that class. Within a class declaration, there are usually *private* members and *public* members. Members of a class declaration are private by default, but the keyword public can be used to make members available for use by the program outside of the class in which they are declared. Private members are those that can only be accessed by other members of that class.

A class declaration is similar to the declaration of a struct. It creates a new data type. This data type can then be used to instantiate objects. Each instance of an object in a class-created data type has its own copy of the variables and functions declared by that class.

2.4.2: Friend Functions: [41]

There may come a time when you have a single function that must work with two different classes of data. When this occurs, that function can be made a friend of each of those two classes. A friend function is not a member of a class, but it has access to the private parts of the class to which it has been declared a friend. A function is declared a friend of a class in the public part of the class declaration and is preceded by friend. A friend of a class can access the members of that class directly.

2.4.3: Steps to designing *bigint* class:

The first step in designing a class for computing with large integers is to select a storage structure to represent these integers.

A linked list seems appropriate because the number of digits in these integers may vary considerably. And because it is necessary to traverse this list in both directions, we would probably use a doubly linked list.

Each integer to be processed will be stored in a separate list, with each node storing a block of consecutive digits in the number.

Since the standard list type provides iterates and a wealth of list operations that are much easier to use than rudimentary pointer operations, we will use it instead, so we will use a list <short int > data member in *bigint* to store its digits.

2.5: Overloading Operators and algorithms describe the basic arithmetic and relational operations:

In this section we will describe overloading operators we used in our new class, the stream operators, the relational operations and the basic arithmetic operations.

The stream operators mean input and output numbers. The relational operations include: less than, greater than, less than or equal, greater than or equal, equal to and not equal.

The arithmetic operations include: addition, subtraction, multiplication, division and modulation.

2.5.1: Overloading Operators:

Operator overloading is the ability for a language to redefine the way its operators behave for certain objects. It allows the programmer to extend the language and give it new abilities. Some languages such as C++, Algol, python and ruby allow operator overloading, and others such as Java deliberately leave it out. Operator overloading is a controversial subject for some thing that you can do with operator overloading, can also be accomplished by using appropriate functions and method calls. On the other hand, it may make your code easier to read and comprehend. It also enables the STL library to work elegantly. [16]

As it happens, C++ is a language that has a lot of operators. In the following lines, we will examine how to overload different operator types. As we will see later on, it is not necessary to overload all operators for a class, just the ones that we think should be overloaded. Also, C++ has some code in the standard library that reduces the amount of code that we need to write. [16]

In order to overload an operator we must write functions. The name of the function that overloads an operator is the reserved word **operator** followed by the operator to be overloaded. Operators are defined as either member functions or friend functions.

Here are the operators we needed in our class.

2.5.1.1: Stream Operators:

2.5.1.1.1: Input operator (>>):

It is the first operation we need it, suppose for convenience that a long integer will be entered in three digit blocks separated by blank. For a *bigint* variable number, the input operation must read these blocks and attach a node containing the value of each of these blocks to number. [13]

Suppose that we want to input a number which consist of four blocks and the first three blocks had been read and stored, when the fourth block read a new node must be created for it and attached to the end of the list by setting the back ward link in this new node to point to the last node in the list, its forward link to point to the head node and then setting the forward link in the last node and the backward link in the last head node to point to this new node.[13]

This exactly what the push_back() operation for a list will do. So, operator>>() need only repeatedly read blocks and push them onto the end of number. [13]

To enter our large integer number we should enter it in three digits separated by space and enter a negative integer in the last block.

Here is the algorithm:

1- Enter the first block (The first block determine wither the number is negative number or positive number).
2- Start an infinite loop.
3- Enter block.
4- If block (unless the fist block because it may a negative integer) less than zero it will return the blocks that had been entered.
5- If block greater than 999, it will reject this block and display an error message.
6- If block greater than zero and less than 999, pushed at the end of the list that's represent the number.

This will be repeated until the user enters a negative number.

2.5.1.1.2: Output operator (<<):

An output operator is simply traverse the list from left to right, displaying the block of digits stored in each node, this can easily be done using a list<short int> iterator, its dereferencing and increment operations. [13]

The algorithm is:

1- Move inside a list that represents the number using an iterator from left to right.
2- Display three digit blocks that's stored in each node using setw(3).
3- Separate each block with space.
4- If the block reaches the end of line then, it will enter in a new line.

2.5.1.2: Relational Operators:

There are various kinds of relational operators should be overloaded to make comparison between numbers.

2.5.1.2.1: Equal operators:

There are two kinds which differ in parameters types:

1- The first assignment operator: consist of two parameters, first parameter of type *bigint* and the second of type integer. When the two parameter passed, it must traverse the list that represent first parameter from left to right, compare with the second parameter, if it is equal it will return true else it will return false.

2- The second assignment operator: consist of two parameters of type *bigint*, it must traverse the lists that representing these two numbers from left to right, compare the two three digit integers with the corresponding nodes, if it is equal move to the next node by using the increment operation and repeat this comparison until reach to the end of lists and if all are equal then return true, else return false.

   This will be done if the size of two numbers is the same, if the size of number1 differs from the size of number2 it will return false.

The explanation of an algorithm:

I.I- if the first parameter of type *bigint* and the second of type integer:

1- Traverse the list that representing first parameter from left to right using while loop.
2- Compare each node with second parameter.

3- If it is equal return true.

4- If it is not equal return false.

5- Move to the next node using an increment operator.

I.II- If both parameters are of type *bigint*:

1- If the size of number1 equal the size of number2 return false.

2- If the size of two numbers is equal then it will traverse the lists that's represent two number from left to right.

3- If node in number1 = corresponding node in number2, move to the next node using an increment operator.

4- If all corresponding nodes are equal, return true else return false.

2.5.1.2.2: Not equal operator:

It is the same as equal operator but instead of equal it could be not equal.

2.5.1.2.3: Greater Than and Less Than operators:

They are the same as equal operator, they have two kinds of parameters differ in its type.

1- The first kind: the type of the first parameter is *bigint* and the second one is integer, it should move on the list that represent first parameter (of type *bigint*) from left to right and if all nodes in number1 is greater than number2 (which is a second parameter) it will return true, else it will return false.

2- The second kind: the type of each parameter is *bigint*, if the size of number1 is greater (less) than the size of number2 it will return true else it will return false. If the two number has the same size , it must traverse the lists representing these two number from left to right, if the first node in number1 is greater (less) than corresponding node in number2 it will return true and stop loop, and if node in number1 is equal to corresponding node in number2 it will move to the next node by using the increment operator and continue the comparison, if node in number1 greater (less) than corresponding node in number2 it will return true and stop loop and if they are equal it will continue the comparison until reach to the end of list ,else it will return false.

Algorithm of Greater (less) than operators:

They are also have two kinds differ in it's type:

II.I- The type of first parameter is *bigint* and the second of type integer.

1- Move on the list of the first parameter from left to right.
2- If nodes in the first parameter greater than the second one, return true.
3- If not, return false.

II.II- The type of each parameter is *bigint*:

1- If the size of both parameter are equal, traverse two list from left to right.
2- If the first node in number1 greater (less) than corresponding node in number2 ,return true and stop loop.
3- If the previous step return false, check if node in number1 equal to the corresponding node in number2.
4- If true then move to next nodes in each number and go to step 2.
5- If not then return false.
6- If size of number1 greater (less) than size of number2, return true else return false.

2.5.1.3: Arithmetic operators:

We will describe the algorithms for performing the basic arithmetic operations: addition, subtraction, multiplication, division and modulation of two n-digit integers $a=(a_{n-1}a_{n-2}\ldots a_1a_0)_r$ and $b=(b_{n-1}b_{n-2}\ldots b_1b_0)_r$ with integers in base r notation where r=1000.

The initial digits of zeros are added in both addition and subtraction operations to make both expansions the same length .We will traverse lists of two number from right to left in addition, subtraction and multiplication operations.

Here is the discussion of the basic five arithmetic operations and its algorithm.

2.5.1.3.1: Addition:

The first arithmetic operation is the addition of two long integers. It should take two extended precision integers as input and return a third extended precision integer that is the sum of the two.

It must traverse the lists representing these two numbers from right to left, adding the two three digit integers in corresponding nodes and the carry digit from the preceding nodes to obtain a three digit sum and a carry digit. A node is created to store this three digit sum and is attached at the front of the list representing the sum of the two numbers.

In the definition of operator+(), we have two numbers, number1 and number2 stored in a separate list, they are traverse from right to left using list<short int> reverse iterators that are moved through the lists synchronously by using the increment operator. The carry digit and the blocks at each position of the iterator are added, the new carry digit and sum block are calculated and the sum block inserted at the front of the list in sum using list insert() operation.

Note:

We mean by the term size, the number of node or blocks in the list,
Each node contain three digit and if our number consist of 9 digits, then the size of this number equal 3(nodes or block) and if the number consist of 11 digits then its size is 4 (the third digit will be filled by zero).

If the size of the two numbers that we apply an arithmetic operation on them is varying then, it must push zeros into the begging of the number which has small size. The result is two numbers with same size.

Algorithm: [12]

When we add a and b we obtain the sum:

$$a + b = \sum_{j=0}^{n-1} a_j r^j + \sum_{j=0}^{n-1} b_j r^j = \sum_{j=0}^{n-1} (a_j+b_j) r^j$$

To find the base r expansion of the a+b first note that there are integers C_0 and S_0 such that:

$$a_0 + b_0 = C_0r + S_0 \quad , \quad 0<=S_0<r$$

Because a_0 and b_0 are positive integers not exceeding r.

C_0 is the carry to the next place .Next we find that there are integers C_1 and S_1 such that:

$$a_1 + b_1 + C_0 = C_1r + S_1 \quad , \quad 0<=S_1<r$$

Proceeding inductively, we find integers C_i and S_i for 1<=i<=ni1 by:

$$a_i + b_i + C_{i-1} = C_ir + S_i \quad , \quad 0<=S_i<r$$

Finally, we let $S_n = C_{n-1}$, since the sum of two integers with n digits has n+1 digits when there is a carry in the place.

We conclude that the base r expansion for the sum is $a + b =(S_{n-1}S_{n-2}\ldots S_1S_0)_r$

2.5.1.3.2: Subtraction:

The second operation is the subtraction operation. It should take two nonnegative digit extended precision integers as input and return a third extended precision integer that is the difference of the two. You should check if the first integer is greater than or equal to the second before subtracting, if number1 less than number2, the resulting of subtraction is equal zero.

Subtraction operation is same as addition operation, it must traverse the lists representing these numbers from right to left, subtract the three digits in number2 from the corresponding three digit in number1, obtain a three digit sub or less.

It will create a node to store this three digit sub and attached at the front of the list representing the sub of two numbers.

If the three digit block in number1 is less than three digits block in number2, it will subtract one (which is equal 1000) from the succedent block and add it to the previous block then subtract from the corresponding node.

This is precisely what happens in the definition of operator-(), the lists in number1 and number2 are traverse from right to left using list<short int> reverse iterators that are moved through the lists synchronously by using the increment operator. The new sub block are obtained and inserted at the front of the list in sub using list's insert() operation .

Algorithm: [12]

Now we will turn our attention to subtraction. We consider:

$$a - b = \sum_{j=0}^{n-1} a_j r^j - \sum_{j=0}^{n-1} b_j r^j = \sum_{j=0}^{n-1} (a_j - b_j) r^j$$

When a > b, there are integers B_0 and d_0 such that:

$$a_0 - b_0 = B_0 r + d_0 \quad , \quad 0<=d_0<r$$

When $a_0 - b_0 >= 0$, we have $B_0 = 0$. Otherwise, when $a_0 - b_0 < 0$, we have $B_0= -1$ B_0 is the borrow from the next place of the base r expansion of a. again to find B_1 and d_1 such that:

$$a_1 - b_1 + B_0 = B_1 r + d_1 \quad , \quad 0<=d_1<r$$

From this equation, we see that the borrow $B_1 = 0$ as long as $a_1 - b_1 + B_0 >= 0$ and $B_1 = -1$ otherwise. We proceed inductively to find integers B_i and d_i, such that:

$$a_i - b_i + B_{i-1} = B_i r + d_i \quad , \quad 0<=d_i<r$$

We see that $B_{n-1} = 0$ since a > b.

We can conclude that: $a - b = (d_{n-1}d_{n-2}\ldots d_1 d_0)_r$

2.5.1.3.3: Multiplication:

It should take two extended precision integers as input and return a third extended precision integer that is the product of the two. Observe that if number1 and number2 are N-digit integers, the product will have at most 2N digits.

Multiplication is one of the most exciting operators, to multiply two long numbers, it must traverse the lists that represent these two numbers from right to left. So, it will need two loops to traverse the lists, the external loop represent the second number and the internal loop represent the first number.

It will takes the first block in the second number and multiply by each block in the first number and stored the resulting in a separate row in sum (which is two dimensional array), after the internal loop finish we will add all rows and the summation will be stored in sum_num array.

We should take in account that when we multiply a blocks in number1 by blocks in number2, it must reserve the position of each block by zero if it is not in the corresponding block. These operations will be repeated until the external loop finish.

Now it has sum_num, it is an array of two dimension, any row in sum_num represent the resulting which is the sum of multiply each block in number2 by all blocks in number1. All elements (cells) in an array contain almost three digits.

It will add the corresponding columns in each rows and the carry digit from the preceding cells to obtain three digits and a carry digit. It will create node to store this three digits and attached at the front of the list representing the multiplication of the two long numbers.

Algorithm:

When multiplying two integers a and b with base **r** expansions, we first multiply each digits of a by all digits of b, shifting each time by the appropriate number of places until reach to the final digit of **a** such that:

$$a * b = a \left(\sum_{j=0}^{n-1} b_j r^j \right) = \sum_{j=0}^{n-1} (a * b_j)\, r^j$$

1- Move on two lists of two numbers using for loop.
2- The external loop represent a.
3- The internal loop represents b.
4- Take first digit in b.
5- Multiply by digit in a.
6- Store the result in a row in an array (sum) of two dimension {recall that each element in the row consist of three digits}.
7- Move to next digit in a.

8- Reserve the position by zero for each digit not in the corresponding digit.
9- Go to step 5 until the internal loop finish.
10- Add all rows in sum and stored in sum_num array.
11- Go to step 4 until the external loop finish.
12- Add all rows in the same column.
13- Obtain three digits and the carry digit.
14- Store three digits in the front of list mult representing the multiplication of two numbers.

2.5.1.3.4: Division:

This is the forth operation, it is the most complicated of the arithmetic operations. The first number must be greater than the second, if it is not then the result is zero. There are many state her:

1- If number1 equal zero and number2 greater than zero then the result is zero.
2- If number1 greater than zero and number2 equal zero then the division is by zero.
3- If number1 equal number2 equal zero then the result is unknown number.
4- If number1 equal number2 then the result equal 1.
5- If number1 less than number2 then the result equal zero.
6- If number1 greater than number2 then the result will be obtain the result as the following algorithm bellow:

Algorithm: [12]

Here is the algorithm we use it to compute $q = a / b$

We wish to find the quotient (**q**) where a=bq+R,   0<=R<=b

a, b : two long integer numbers, **R**: remainders,

r: radix or base (which is equal to 1000 in our project),

n: the size of number a (number of the node in the list a),

i=0

1) If the base (radix) expansion of q is q = $(q_{n-1}q_{n-2}\ldots q_1q_0)_r$, then we have:

$$a = b \left(\sum_{j=0}^{n-1} q_j r^j \right) + R , \qquad 0 <= R <= b$$

2) Determine the first digit q_{n-1} of q

$$a - bq_{n-1} r^{n-1} = b \left(\sum_{j=0}^{n-2} q_j r^j \right) + R$$

The right hand side of this equation is positive and less than br^{n-1}

Therefore $0 <= a- bq_{n-1} r^{n-1} < br^{n-1}$

Thereupon $q_{n-1} = a / (br^{n-1})$

We can obtain q_{n-1} by successively subtracting br^{n-1} from a until
a negative result is obtained .

3) Find other digits of q by defining the sequence of partial remainders R by
$R_0 = a$ and $R_i = R_{i-1} - bq_{n-i} r^{n-i}$

Consequently, since $R_i = R_{i-1} - bq_{n-i} r^{n-i}$ and $0 <= R_i < r^{n-1}b$, we see that the digit q_{n-i} is given by $[R_{i-1} / (br^{n-i})]$ and can be obtained by successively subtracting br^{n-i} from R_{i-1} until a negative result is obtained , and then q_{n-i} is one less than the number of subtractions. This is how we find the digits of q.

Know we will discuss how we will obtain the digits of the quotient (q) which it is in fact a subtraction operation, firstly it will check if the size of number1(R_{i-1}) greater than the size of number2 (br^{n-i}), if yes it will fill the difference with zeros in the begging of number2 to obtain two number with same size.

After that the subtraction operation will start using while loop and a counter (c). While loop here checks if number1 is greater than number2, if yes then the counter (c) will be increase until number1 became less than number2 and stop loop.

Our program dealing with large numbers. So, the counter will be large as well. Inside the loop when the counter greater than (999) it will store 999 in the div list and make c equal to zero, whereas any node in div represent (999).

It will find how many (999) in div list by using size() function. If the size is greater than zero that's mean the div list contain nodes in it, and while the size is greater than zero it must attached three digit block to div1 list until the size become zero.

Div2 will consist of one node contain (999). If the counter greater than (999) div3 will be calculated by multiply div1 by div2 (that is mean it will call multiplication operation here) and if the counter contain value less than (999) it will be pushed in div4 list then add it to div3 obtain div5 which represent the resulting of division two number2.

If the size of div equal zero, it is mean there is no node inside it, so, the counter (c) will had value less than (999) and it will be stored in div5 represent the division of two numbers (R_{i-1} and (br^{n-i})). In which to represent the digits in q.

After obtained the first digit it will be pushed in the q1 list and repeat this operation until i < n, finally q1 represent the division of the two numbers.

2.5.1.3.5: Modulation:

It is the last operation, it seems to be like division operation, there is also much state we will illustrate below: -

1- If number1 equals zero or number2 equal 1 or number1 equal number2 then the modulation equal zero.

2- If number2 equal zero and number1 greater than zero, then division is by zero.

3- If number1 equal number2 equal zero then the division is unknown value.

4- If number1 less than number2 then modulation equal number1.

5- If number1 greater that number2 then modulation will be obtained as follow: -

   It is the same as division algorithm, instead of returned q it will return R which is a remainder of two large integer numbers.

Chapter Three
Number Theory

Introduction:

This chapter is concerns with Number Theory, we will give an introduction to number theory and in the rest of this chapter we will explain modular arithmetic, modular exponentiation, how to obtain the greatest common divisor of two positive integers by using Extend Euclidean Algorithm, the prime number and the inverses modular.

3.1: Introduction to Number Theory:

Number theory is one of the oldest and largest branches of pure mathematics. It is that branch of pure mathematics concerned with the properties of integers. It contains many results and open problems that are easily understood, even by non-mathematicians. More generally, the field has come to be concerned with wider classes of problems that have arisen naturally from the study of integers. Number theory may be subdivided into several fields, according to the methods used. [29]

Our research focuses on some basic Number Theory such like obtaining the GCD (Greatest Common Divisor) by using the Extended Euclid's Algorithm, modular arithmetic, modular exponentiation, inverses modulo, check if the number is prime or not.

Many public-key encryption and digital signature schemes, and some hash functions require computations in Z_m The set of integers {…., -3, -2, -1, 0, 1, 2, 3, …. } is denoted by the symbol Z, the integers modulo m (m is a large positive integer which may or may not be a prime). For example, the RSA, Rabin, and ElGamal schemes require efficient methods for performing multiplication and exponentiation in Z_m. The efficiency of any cryptographic scheme depend on a number of factors, such as parameter size, time-memory tradeoffs, processing power available, software and/or hardware optimization, and mathematical algorithms. [17]

We know that there are a number of techniques for doing modular multiplication and exponentiation. Efficiency can be measured in numerous ways; thus, it is difficult to definitively state which algorithms the best. An algorithm may be efficient in the time it takes to perform a certain algebraic operation, but quite inefficient in the amount of storage it requires. One algorithm may require more code space than another. [17]
Depending on the environment in which computations are to be performed, one algorithm may be preferable over another. [17]

3.2: Modular arithmetic:

It is an important field in Number Theory, this section gives the definition of modular arithmetic, its properties and applications and the modular exponentiation.

3.2.1: Definition: [44]

Modular arithmetic is interesting as an abstract topic in number theory, but it also plays a daily role in real life. It is the basis for public key cryptography and check digits associated with error detection. Here we describe the idea of modular arithmetic and can help us to better understand primes and composite numbers.

Suppose that a, b, and n are integers, n > 0. We say that a and b are congruent modulo n if and only if n|(a - b). We denote this relationship as:

$$a=b \pmod{n}$$

and read these symbols as “a is congruent to b (mod n).”

3.2.2: Modular arithmetic properties: [33]

Modular arithmetic is remainder arithmetic. So for instance, 7 mod 5=2 because 5 goes into 7 once with a remainder of 2. Congruent modulo is an interesting concept and important to cryptography. a = b mod(n) says that a and b are congruent modulo n if a mod(n) = b mod(n).

Furthermore, additional modulo arithmetic properties are:

Reducibility

1. (a + b) mod(n) = [(a mod(n)) + (b mod(n))] mod(n)
2. (a - b) mod(n) = [(a mod(n)) - (b mod(n))] mod(n)
3. (a * b) mod(n) = [(a mod(n)) * (b mod(n))] mod(n)

Distributivity

1. a * (b + c) mod(n) = [(a * b) + (a * c)] mod(n)

Identities

1. a + 0 mod(n) = a + 0 mod(n) = a
2. a * 1 mod(n) = 1 * a mod(n)

Inverses

1. a + (-a) mod n = 0
2. a * a-1 mod(n) = 1 if a not equal 0

Additive Inverse

The additive inverse modulo n to an integer x is y such that x+y=0 mod n.

3.2.3: Application Of Modular Arithmetic:

Much of modern number theory and many practical problems (including problems in cryptography and computer science) are concerned with modular arithmetic. [1]

In arithmetic modulo N, we are concerned with arithmetic on the integers, where we identify all numbers which differ by an exact multiple of N.[1]

That is, x = y mod N if x = y + m*N for some integer m. This identification divides all the integers into N equivalence classes. We usually denote these by their “simplest” members, that is, the numbers 0, 1, . . . , N −1. (Usually In the case of clock arithmetic (modulo 12), we use 1, . . . , 12 instead). [1]

Most ordinary arithmetic operations extend to modular arithmetic straightforwardly. [1]

$$x + y \rightarrow x + y \bmod N,$$
$$xy \rightarrow xy \bmod N,$$
$$x^y \rightarrow x^y \bmod N.$$

This does lead to some things, which are strange based on the intuition from ordinary integer arithmetic. For instance, all numbers have additive inverses, but these are now represented by positive numbers:

$$(-x) = N - x,$$

so the additive inverse of 3 modulo 7 is 4.

And unlike ordinary arithmetic, it is possible for a non-zero integer to have a multiplicative inverse, as well: $3 * 5 = 15 = 1 \bmod 7$. In fact, for prime N, all numbers $0, \ldots, N - 1$ have multiplicative inverses. [1]

If N is composite, then all numbers which have no common factor with N have multiplicative inverses. [1]

Generally, modular exponentiation problems take the form where given base b, exponent e, and modulus m, one wishes to calculate c such that: **$c=b^e(\bmod m)$**. [28]

If b, e, and m are non-negative and $b < m$, then a unique solution c exists and has the property $0 \le c < m$. For example, given $b = 5$, $e = 3$, and $m = 13$, the solution c works out to be 8. [28]

Modular exponentiation problems similar to the one described above are considered easy to do, even if the numbers involved are enormous. On the contrary, computing the discrete logarithm (finding b given c, e, and m) is believed to be difficult. This one way function behavior makes modular exponentiation a good candidate for use in cryptographic algorithms. [28]

Modular Arithmetic is the basis of many important operations in cryptographic applications. The RSA algorithm relies on modular exponentiation to provide secure public key encryption and decryption. [4]

This modular exponentiation is achieved through a combination of modular multiplications and squaring of the inputs. [4]

By definition, modular multiplication reduces the product of two numbers through division by the modulus, resulting in a bounded remainder. [4]

Modular inversion is used in applications such as the generation of public/private key pairs in the RSA system, the Diffie-Hellman key exchange algorithm and more recently in elliptic curve cryptography (ECC). Modular inversion can also be used to accelerate modular exponentiation in conjunction with addition-subtraction chains, where canonical recoding is used to reduce the average number of non-zero multiplications. [4]

3.2.4: Computing Modular Exponentiation(x^e mod n):

One of the most important arithmetic operations for public-key cryptography is exponentiation. [17]

The RSA scheme requires exponentiation in Z_m (Z is set of integer numbers) for some positive integer m, whereas Diffie-Hellman key agreement and the ElGamal encryption scheme use exponentiation in Z_p for some large prime p. ElGamal encryption can be generalized to any finite cyclic group. This section discusses methods for computing the exponential g^e, where the base g is an element of a finite group G and the exponent e is a non-negative integer. A reader uncomfortable with the setting of a general group may consider G to be Z^*_m; that is, read g^e as g^e mod m. [17]

An efficient method for multiplying two elements in the group G is essential to performing efficient exponentiation. The most naive way to compute g^e is to do e - 1 multiplications in the group G. For cryptographic applications, the order of the group G typically exceeds 2^{160} elements, and may exceed 2^{1024}. Most choices of e are large enough that it would be infeasible to compute g^e using e - 1 successive multiplications by g.

There are two ways to reduce the time required to do exponentiation. One way is to decrease the time to multiply two elements in the group; the other is to reduce the number of multiplications used to compute g^e. Ideally, one would do both. [17]

[38]The following lines consider RSMA algorithm, this is known as Repeated Square-and-Multiply Algorithm. Here is the RSMA algorithm:

Assume the binary representation of e is b_k, ……., b_0. Note that

$x^e = \prod^k_{i=0} {}_x b_i{*}2^i$.

The algorithm is:

```
y=$x^{b0}$;
t=x;
for i =1 to k do:
        { $t=x^{2i-1}$; $y=\prod^{i-1}_{j=0} {}_x bj{*}2^j$ }
        $t=t^2$ mod n; if $b_i$ then y=t*y mod n
end
return y;
```

Example. We want to find (34^{29} mod 137).

So, x = 34, e = 29, and n = 137. Thus, (b_0, b_1, b_2, b_3, b_4) = (1, 0, 1, 1, 1). We start with y = t = 34. We then have:

1. t = 34^2 mod 137 = 60.
2. t= 60^2 mod 137 = 38; y = 38 * 34 mod 137 = 59.
3. t = 38^2 mod 137 = 74; y = 59 * 74 mod 137 = 119.
4. t = 374^2 mod 137 = 133; y = 119 * 133 mod 137 = 72.

The idea is to decrease the number of multiplications used to compute g^e, the naive algorithm would compute g*g*…*g, e times, but instead it is more effective to compute $g^1{*}g^2{*}g^4$ …because the values g^2 = g*g and $g^4 = g^2{*}g^2$ can be computed using fewer multiplications. An example, compute g^{29}, this would require 29 multiplications to compute in a naive way, but observe that 29 = $10111_2$and that means that $g^{29} = g^{1011112} = g^{29} = g^{14}{*}g^2{*}g^1$which require a lot fewer calculations.

3.3: Greatest Common Divisor (GCD):

Many situations in cryptography require the computation of the Greatest Common Divisor (GCD) of two positive integers. There are Algorithms describe the classical Euclidean algorithm for this computation. [17]

[54] Firstly, we will know what do we mean by GCD? When we divide one integer by another (nonzero) integer we get an integer *quotient* (the "answer") plus a *remainder* (generally a rational number). For instance,

13/5 = 2 ("the quotient") + 3/5 ("the remainder").

We can rephrase this division, totally in terms of integers, without reference to the division operation:

$$13 = 2(5) + 3.$$

Note that this expression is obtained from the one above it by multiplying through by the divisor 5.

We refer to this way of writing a division of integers as the **Division Algorithm for Integers**. More formally stated:

If a and b are positive integers, there exist integers unique non-negative integers q and r so that

$$a = q*b + r \text{ , where } 0<=r < b.$$

q is called the *quotient* and r the *remainder*.

The *greatest common divisor* of integers a and b, denoted by *gcd(a, b)*, is the largest integer that divides (without remainder) both a and b. So, for example:

gcd(15, 5) = 5,   gcd(7, 9) = 1,   gcd(12, 9) = 3,   gcd(81, 57) = 3.

The gcd of two integers can be found by repeated application of the division algorithm, this is known as the ***Euclidean Algorithm***. You repeatedly divide the divisor by the remainder until the remainder is 0. The gcd is the last non-zero remainder in this algorithm. The following example shows the algorithm.

Finding the gcd of 81 and 57 by the Euclidean Algorithm:

$$81 = 1(57) + 24$$
$$57 = 2(24) + 9$$
$$24 = 2(9) + 6$$
$$9 = 1(6) + \mathbf{3}$$
$$6 = 2(3) + 0.$$

It is well known that if the gcd(a, b) = r then there exist integers p and s so that:

$$p(a) + s(b) = r. \quad [54]$$

There are variant methods for computing the gcd which are more efficient.

Let a, b be integers not both zero. A positive integer d is the greatest common divisor of a and b if

(i) d/a and d/b;

(ii) c/a and c/b $\rightarrow$ c/d

We write d=(a, b) if (a, b) =1, we say that a and b are relatively prime.[17]

The greatest common divisor of two integers a and b can be computed via various algorithm.

However, computing a gcd by first obtaining prime-power factorizations does not result in an efficient algorithm, as the problem of factoring integers appears to be relatively difficult. The Euclidean algorithm is an efficient algorithm for computing the greatest common divisor of two integers that does not require the factorization of the integers. It is based on the following simple fact.

If a and b are positive integers with a > b, then gcd(a, b) = gcd(b, a mod b). 17]

3.3.1: Euclidean algorithm for computing the greatest common divisor of two integers:[17]

```
INPUT: two non-negative integers a and b with a >= b.
OUTPUT: the greatest common divisor of a and b.
1. While b != 0 do the following:
    1.1 Set r = a mod b
    1.2 a = b
    1.3 b = r
2. Return(a).
```

Example (Euclidean algorithm) The following are the division steps of the pervious Algorithm:

For computing gcd(4864, 3458) = 38:

$$4864 = 1*3458 + 1406$$

$$3458 = 2*1406 + 646$$

$$1406 = 2*646 + 114$$

$$646 = 5*114 + 76$$

$$114 = 1*76 + 38$$

$$76 = 2*38 + 0$$

The Euclidean algorithm can be extended so that it not only yields the greatest common divisor d of two integers a and b, but also integers x and y satisfying ax + by = d.

3.3.2: Extended Euclidean algorithm:[17]

INPUT: two non-negative integers a and b with a>=b.

OUTPUT: d = gcd(a; b) and integers x, y satisfying ax + by = d.

```
1. If $b = 0$ then:
    set $d = a$
        $x = 1$
        $y = 0$
      return(d,x,y).
2. Set $x_2 = 1$
        $x_1 = 0$
        $y_2 = 0$
        $y_1 = 1$
3. While $b > 0$ do the following:
    3.1 $q = [a/b]$
        $r = a - q*b$
        $x = x_2 - q*x_1$
        $y = y_2 - q*y_1$
    3.2 $a=b$
        $b=r$
        $x_2=x_1$
        $x_1=x$
        $y_2=y_1$
        $y_1=y$
4. Set $d = a$
        $x = x_2$
        $y = y_2$
      return(d,x,y).
```

Table 8 shows the steps of the previous Algorithm with inputs a = 4864 and b = 3458.

Hence gcd(4864, 3458) = 38 and (4864)(32) +(3458)(-45) = 38.

Table 8: Example to get the GCD by using the Extend Euclidean Algorithm:

Q	r	X	Y	a	B	X_2	X_1	Y_1	Y_2
-	-	-	-	4864	3458	1	0	0	1
1	1406	1	-1	3458	1406	0	1	1	-1
2	646	-2	3	1406	646	1	-2	-1	3
2	114	5	-7	646	114	-2	5	3	-7
5	76	-27	38	114	76	5	-27	-7	38
1	38	32	-45	76	38	-27	32	38	-45
2	0	-91	128	38	0	32	-91	-45	128

4.4: Prime Number:

A **prime** number is a number which has no divisors other than 1 and the number itself.

Thus, 13 is a **prime** but 12 is not, it has divisors 2, 3, and 6. A number which is not prime is said to be **composite**.

In modern cryptography, we normally use special mathematical tools for generating prime numbers - as we will see in the next chapter -. Naturally, we want to convince ourselves that the generated number is prime. [34]

Did you ever stop to wonder if 473155932665450549 was prime?

Proving this is somewhat difficult, so we turn the question up-side down, and convince ourselves probabilistically that we are dealing with a prime. If the test succeeds you are dealing with a prime (with extremely high probability). If the test fails, you can be certain that you are not dealing with a prime. [34]

3.4.1: Relatively prime: [21]

Two integers are **relatively prime** if there is no integer greater than one that divides them both (that is, their greatest common divisor is one). For example, 12 and 13 are relatively prime, but 12 and 14 are not.

A list of integers is (mutually) **relatively prime** if there is no integer that divides them all. For example, the integers 30, 42, 70, and 105 are mutually relatively prime (but not pair wise relatively prime).

3.4.2: Pseudoprime:[30]

A pseudoprime is a composite number that passes a test or sequence of tests that fail for most composite numbers. Unfortunately, some authors drop the "composite" requirement, calling any number that passes the specified tests a pseudoprime even if it is prime.

A **pseudoprime** is a probable prime (an integer which shares a property common to all prime numbers) which is not actually prime. Pseudoprimes can be classified according to which property they satisfy.

The most important class of pseudoprimes come from Fermat's little theorem and hence are called **Fermat pseudoprimes**. This theorem states that if p is prime and a is coprime to p, then $a^{p-1} - 1$ is divisible by p. If a number x is not prime, a is coprime to x and x divides $a^{x-1} - 1$, then x is called a *pseudoprime to base a*. A number x that is a pseudoprime for all values of a that are coprime to x is called a Carmichael number.

3.4.3: Coprime: [5]

In mathematics, the integers a and b are said to be **coprime** or **relatively prime** if and only if they have no common factor other than 1 and −1, or equivalently, if their greatest common divisor is 1.

For example, 6 and 35 are coprime, but 6 and 27 are not because they are both divisible by 3. The number 1 is coprime to every integer; 0 is coprime only to 1 and −1.

A fast way to determine whether two numbers are coprime is given by the Euclidean algorithm.

Properties of coprime:

The numbers a and b are coprime if and only if there exist integers x and y such that $ax + by = 1$. Equivalently, b has a multiplicative inverse modulo a: there exists an integer y such that $by \equiv 1 \pmod{a}$.

As a consequence, if a and b are coprime and $br \equiv bs \pmod{a}$, then $r \equiv s \pmod{a}$ [because we may "divide by b" when working modulo a]. Furthermore, if a and b_1 are coprime, and a and b_2 are coprime, then a and b_1b_2 are also coprime [because the product of units is a unit].

If a and b are coprime and a divides a product bc, then a divides c.

The probability that two randomly chosen integers are coprime is $6/\pi^2$.

3.4.4: Primality Tests:

A primality test is an algorithm for determining whether an input number is prime or not.

A primality test is a criterion for a number **n** not to be prime, thus if n "fails" the test that’s mean n is definitely composite, if n "passes" the test that's mean n may be prime, and if n "passes" many tests this mean n is very likely prime. [45]

There are too many methods to test if our number is prime or not we will explain a few methods and apply one of them in our project.

3.4.4.1: Method 1: Naive Method: [24]

This is a simplest primality test, it is as follows: Given an input number n, we see if any integer k from 2 to n-1 divides n. If n is divisible by any k then n is composite, otherwise it is prime.

A slightly better method is to see if n is divisible by any integer k from 2 to $\sqrt{n}$, inclusive. If n is composite then it can be factored into two values, at least one of which is less than or equal to $\sqrt{n}$.

Briefly, This method test for some/all m <= $\sqrt{n}$ whether m|n.

3.4.4.2:Method 2: Fermat Test:[26]

The **Fermat primality test** is a probabalistic test to determine if a number is composite or *probably* prime.

Fermat's little theorem states that if *n* is prime and $1 \le b \le n$, then

$$b^{n-1} \equiv 1 (\text{mod } n).$$

If we want to test if *n* is prime, then we can pick random *b*'s in the interval and see if the equality holds. If the equality does not hold for a value of *b*, then *n* is composite. If the equality does hold for many values of *b*, then we can say that *n* is probably prime, or a pseudoprime.

Carmichael numbers:

A Carmichael number is an odd composite number *n* which satisfies Fermat's little theorem

$$a^{n-1} - 1 = 0 \ (mod\ n)$$

For every choice of *a* satisfying $(a, n)=1$(i.e., *a* and *n* are relatively prime) with $1 < a < n$. A Carmichael number is therefore a pseudoprime to any base. [6]

We know that if a number **n** successfully passes the Fermat Theorem test:

$$b^{n-1}=1 \bmod n$$

For all coprime bases 0< b < n, this does not necessary mean that **n** is a prime number. [43]

Example. Run the above test for **n=225593397919** and random bases **b**. I would be very surprised if **n** has ever failed the test. However, **n** is not a prime number! The explanation of the remarkable behavior of the number **n=225593397919** is that the above condition holds for every coprime base **b**. Of course, it does not hold if the base **b** has with **n** a common prime factor. But, since prime factors of **n** are rather big, namely, the prime factorization of **n** is [43]

$$\mathbf{225593397919=2207 * 6619 * 15443,}$$

a randomly chosen base is almost always coprime to **n**. This is why **n** passes the test again and again.

Odd composite numbers **n** with the property that they are psudoprime to any coprime base are called **Carmichael numbers**. You can try the Fermat Theorem Test on the nastiest Carmichael number in the range **< 10^{16}**:

9585921133193329

However it is relatively easy to recognize a Carmichael number.

Theorem: If n is a product of distinct prime numbers,

$$\mathbf{n=p_1* p_2 * ...* p_s,}$$

and

$$\mathbf{p_i -1 \mid n-1}$$

for every prime factor **p_i , i=1, ..., s** , then **n** is a Carmichael number.

The converse is also true: every Carmichael number **n** has at least three prime factors, all its prime factors **p** are different and satisfy

$$\mathbf{p-1 \mid n-1} .$$

3.4.4.3: Method 3: Euler Test (Solovay-Strassen Method): [32]

The Solovay-Strassen primality test is a probabilistic test to determine if a number is composite or probably prime.

Euler (**Leonhard Euler** is Swiss mathematician noted both for his work in analysis and algebra, including complex numbers and logarithms, and his introduction of much of the basic notation in mathematics) proved that for a prime number n,

For b < n with gcd(b, n) =1 whether **n** satisfies

$$b^{(n-1)/2} = (\tfrac{b}{n})(\mathrm{mod}\ n)$$

Thus, if we have a value **n** and want to determine if it is prime, we can check many random values of **b** and make sure the above equality holds. If it does not hold for some a, we know that p must not be prime.

Therefore, n is an Euler pseudoprime to the base b. when n passes K tests this mean the probability of n to be composite $<=1/2^k$.

3.4.4.4: Method 4: Miller-Rabin Test: [27]

The **Miller-Rabin primality test** is a primality test: an algorithm which determines whether a given number is prime, similar to the Fermat primality test and the Solovay-Strassen primality test. Its original version, due to G. L. Miller, is deterministic, but it relies on the unproven generalized Riemann hypothesis; M. O. Rabin modified it to obtain an unconditional probabilistic algorithm.

Just like with the Fermat and Solovay-Strassen tests, with the Miller-Rabin test we will rely on an equality or set of equalities that hold true for prime values, and then see whether or not they hold for a number that we want to test for primality.

Let n be an odd prime, then we can write $n - 1$ as $2^s \times d$, where s is an integer and d is odd, this is the same as factoring out 2 from n repeatedly. Then, one of the following must be true for some $a \in (\mathbb{Z}/n\mathbb{Z})^*$:

$$a^d \equiv 1 (\text{mod } n)$$

or

$$a^{2^r \cdot d} \equiv -1 \pmod{n} \text{ for some } 0 \leq r \leq s - 1$$

To show that one of these must be true, recall Fermat's little theorem:

$$a^{n-1} \equiv 1 (\text{mod } n)$$

So, if we keep taking square roots of a^{n-1}, we will either get 1 or −1. If we get −1 then the second equality holds and we are done.

In the case when we've taken out every power of 2 and the second equality never held, we are left with the first equality which also must be equal to 1 or −1, as it too is a square root. However, if the second equality never held, then it couldn't have held for $r = 0$, meaning that

$$a^{2^0 \cdot d} = a^d \not\equiv -1 \pmod{n}$$

Thus in the case the second equality doesn't hold, the first equality must.

The Miller-Rabin primality test is based on the above equalities. If we want to test to see if *n* is prime, then if

$$a^d \not\equiv 1 \pmod{n}$$

and

$$a^{2^r d} \not\equiv -1 \pmod{n} \text{ for all } 0 \le r \le s-1$$

then *a* is called a *strong witness* for the compositeness of *n*. Otherwise *a* is called a *strong liar* and *n* is a *probable prime*.

3.4.5: Algorithm for Miller-Rabin primality test[8]

The input value is n to test for primality and the output is composite or prime. The algorithm can be written as follows:

1- Select $x \in R \{1, \ldots, n-1\}$
2- Compute gcd (x, n) = d , if $d \neq 1$ then return "composite" , (observe that when gcd (x, n) =1 , $x \in R\ Z^{+}_{n}$)
3- Write $n-1 = m * 2^k$ where m is odd
4- If $x^m = 1 \pmod{n}$ then return "prime"
5- For i = 0 To k-1 Do

   If $x^{2^k . m} = -1 \pmod{n}$ then return "prime" else return "composite" .

If n passes K tests, then the probability of n to be composite $<=1/4^k$

3.4.6: Proving Primality:

In fact, verifying that a large number *is* prime can be difficult. There are many methods as shown above that ensure it is *highly probable* the *n* is prime, but do not provide a guarantee.

In our project, we would generate random large integers and try to prove they are prime by using Miller-Rabin primality test.

3.5: Inverse Modulo:

Modular inversion is used in many applications such as the generation of public/private key pairs in the RSA system (it will be discuss in the next chapter), the Diffe-Hellman key exchange algorithm and more recently in elliptic curve cryptography (ECC). Modular inversion can also be used to accelerate modular exponentiation in conjunction with addition-subtraction chains, where canonical recoding is used to reduce the average number of non-zero multiplications.[41]

A modular inverse of an integer *b* (modulo *m*) is the integer b^{-1} such that

$$b^{-1} = 1 \text{ (mod m)}$$

Every nonzero integer *b* has an inverse (modulo *p*) for *p* a prime and *b* not a multiple of *p*. For example, the modular inverses of 1, 2, 3, and 4 (mod 5) are 1, 3, 2, and 4.

If *m* is not prime, then not every nonzero integer *b* has a modular inverse. For example,$1^{-1} = 1$ (mod 4) and $3^{-1} = 3$ (mod 4), but 2 does not have a modular inverse. [7]

The Modular inverse of an integer a $\in$ [1,M-1] modulo a prime M is defined as the integer x, such that:

$$a * x = 1 \text{ (mod M)}$$

which can be written:

$$x := \text{ModInv (a)} = a^{-1} \text{ (mod M)}$$

Note that the multiplicative inverse of an integer a (mod M) exists if, and only if, a and M are relatively prime. [4]

Example: [52]

Let: $Z_n = \{0, 1, 2, \dots, n-1\}$

Let: $Z^*_n = \{x \in Zn \mid \gcd(x, n) = 1\}$

$Z_{15} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$

$Z^*_{15} = \{1, 2, 4, 7, 8, 11, 13, 14\}$

$Z_6 = \{0, 1, 2, 3, 4, 5\}$

$Z^*_6 = \{1, 5\}$

Fact: Z^*_n is the set of all elements in Z_n that have a multiplicative inverse.

Proof:

If gcd(x, n) = 1, then there are a, b :

$ax + bn = 1$

$\rightarrow$ $ax = 1 \pmod n$

$\rightarrow$ a is a multiplicative inverse of x

Modular multiplication of **a** and **b** may be accomplished by simply multiplying a and b as integers, and then taking the remainder of the result after division by n. Inverses in Z_n can be computed using the extended Euclidean algorithm as next described. [17]

Algorithm Computing multiplicative inverses in Z_n

INPUT: $a \in Z_n$.

OUTPUT: a^{-1} mod n, provided that it exists.

1. Use the Extended Euclidean algorithm to find integers x and y such that: $ax + ny = d$, where $d = \gcd(a, n)$.
2. If $d > 1$, then a^{-1} mod n does not exist. Otherwise, return(x). [17]

3.5.1: The Extended Euclidean Algorithm & Modular Inverses: [49]

The Extended Euclidean algorithm not only computes gcd(n, m), but also returns the numbers a and b such that $\gcd(n, m)=a*n + b*m$. If $\gcd(n, m)=1$ this solves the problem of computing modular inverses.

Assume that we want to compute $n^{(-1)} \pmod m$ and further assume that $\gcd(n, m)=1$. Run the Extended Euclidean algorithm to get a and b such that

$a*n + b*m=1$. Rearranging this result, we see that $a*n=1-b*m$, or $a*n=1(\text{mod } m)$. This solves the problem of finding the modular inverse of n, as this shows that $n^{(-1)}=a(\text{mod } m)$.

The Extended Euclidean algorithm is nothing more than the usual Euclidean algorithm, with side computations to keep careful track of what combination of the original numbers n and m have been added at each step. The algorithm runs generally like this:

1. Write down n, m, and the two vectors (1,0) and (0,1)
2. Divide the larger of the two numbers by the smaller call this quotient q
3. Subtract q times the smaller from the larger (i.e. reduce the larger modulo the smaller)
4. Subtract q times the vector corresponding to the smaller from the vector corresponding to the larger
5. Repeat steps 2 through 4 until the result is zero
6. Publish the preceding result as gcd(n, m)

An example of this algorithm is the following computation of $30^{(-1)}$(mod 53):

53	30	(1,0)	(0,1)
53-1*30=23	30	(1,0)-1*(0,1)=(1,-1)	(0,1)
23	30-1*23=7	(1,-1)	(0,1)-1*(1,-1)=(1,2)
23-3*7=2	7	(1,-1)-3*(1,2)=(-2,-7)	(1,2)
2	7-3*2=1	(-2,-7)	(1,2)-3*(-2,7)=(7,23)
2-2*1=0	1	(-2,-7)-2*(7,23)=(-16,-53)	(7,23)

From this we see that gcd(30,53)=1 and, rearranging terms, we see that 1=7*53+23*30, so we conclude that $30^{(-1)}=23$(mod 53). (This can be confirmed by checking that 23*30 = 1(mod 53)).

3.5.2: Extended Euclidean Algorithm [53]

The Extended Euclidean Algorithm determines, if present, the inverse of b modulo n:

```
1. $b_0 = b$
2. $n_0 = n$
3. $t_0 = 0$
4. $t = 1$
5. $Q = n_0 / b_0$
6. $r = n_0 - q * b_0$
7. while $r > 0$ do
8.         $temp = t_0 - q * t$
9.         if $temp >= 0$ then $temp = temp \bmod n$
10.        if $temp < 0$ then $temp = n - ((-temp) \bmod n)$
11.        $t_0 = t$
12.        $t = temp$
13.        $n_0 = b_0$
14.        $b_0 = r$
15.        $q = n_0 / b_0$
16.        $r = n_0 - q * b_0$
17. if $b_0 != 1$ then
           $b$ has no inverse modulo $n$
    else
           $b^{-1} = t \bmod n$
```

Chapter four

RSA Public Key Cryptosystem

Introduction:

This chapter explains our implementation which is the RSA public-key cryptosystem.

We will apply our new data type (*bigint*) in RSA public key cryptosystem and ensure that it is act as any data type define in any computer language and given a high level of security by generating a very large number.

This chapter consists of the following subjects an explanation of a Cryptology, Cryptography, Cryptanalysis and other terminology related to Cryptology, types of cryptosystems, public key cryptosystem and the RSA public key cryptosystem.

4.1: Cryptology:

Here an explanation of various terminologies related to cryptology.

4.1.1: Cryptography:

The story begins: When Julius Caesar sent messages to his trusted acquaintances, he didn't trust the messengers. So he replaced every A by a D, every B by a E, and so on through the alphabet. Only someone who knew the "shift by 3" rule could decipher his messages. [3]

Cryptography refers to the creation and use of systems for disguising messages so that only the intended party can read the message contents. Of fundamental importance in cryptography is the idea of algorithmic complexity, that is, how hard it is to perform certain operations where "hard" is directly related to how long it takes to solve the problem. All successful cryptosystems rely on it being hard to decrypt the encrypted message without knowledge of some secret piece of information - that is, it must take longer than anyone is willing to wait in order to 'break' the system and guess the secret needed to recover the original message. [3]

Historically, data encryption has been used primarily to protect diplomatic and military secrets from foreign governments. It is also now used increasingly by the financial industry to protect money transfers, by merchants to protect credit-card information in electronic commerce, and by corporations to secure sensitive communications of proprietary information.

All modern cryptography is based on the use of algorithms to scramble (encrypt) the original message, called plaintext, into unintelligible babble, called ciphertext. The operation of the algorithm requires the use of a key. Until 1976 the algorithms were symmetric, that is, the key used to encrypt the plaintext was the same as the key used to decrypt the cipher text. In 1977 the asymmetric or public key algorithm was introduced by the American mathematicians W. Diffie and M. E. Hellman. This algorithm requires two keys, an unguarded public key used to encrypt the plaintext and a guarded private key used for decryption of the cipher text; the two keys are mathematically related but cannot be deduced from one another. The advantages of asymmetric algorithms are that compromising one of the keys is not sufficient for breaking the cipher and fewer unique keys must be generated.[25]

In 1977 the Data Encryption Standard (DES), a symmetric algorithm, was adopted in the United States as a federal standard. DES and the International Data Encryption Algorithm (IDEA) are the two most commonly used symmetric techniques. The most common asymmetric technique is the RSA algorithm, named after Ronald Rivest, Adi Shami, and Len Adleman, who invented it while at the Massachusetts Institute of Technology in 1977. Other commonly used encryption algorithms include Pretty Good Privacy

(PGP), Secure Sockets Layer (SSL), and Secure Hypertext Transfer Protocol (S-HTTP). The National Institute of Standards and Technology (NIST) is working with industry and the cryptographic community to develop the Advanced Encryption Standard (AES), a mutually acceptable algorithm that will protect sensitive government information and will be used by industry on a voluntary basis. [25]

Cryptography (from Greek kryptós, "hidden", and gráphein, "to write") is, traditionally, the study of means of converting information from its normal, comprehensible form into an incomprehensible format, rendering it unreadable without secret knowledge — the art of encryption. In the past, cryptography helped ensure secrecy in important communications, such as those of spies, military leaders, and diplomats. In recent decades, the field of cryptography has expanded its remit in two ways. Firstly, it provides mechanisms for more than just keeping secrets: schemes like digital signatures and digital cash, for example. Secondly, cryptography has come to be in widespread use by many civilians who do not have extraordinary needs for secrecy, although typically it is transparently built into the infrastructure for computing and telecommunications, and users are not aware of it. [25]

The study of how to circumvent the use of cryptography is called cryptanalysis, or code breaking. Cryptography and cryptanalysis are sometimes grouped together under the umbrella term cryptology, encompassing the entire subject. In practice, "cryptography" is also often used to refer to the field as a whole; crypto is an informal abbreviation. [25]

Cryptography is an interdisciplinary subject, drawing from several fields. Before the time of computers, it was closely related to linguistics. Nowadays the emphasis has shifted, and cryptography makes extensive use of technical areas of mathematics, especially those areas collectively known as discrete mathematics. This includes topics from number theory, information theory, computational complexity, statistics and combinatorics. It is also a branch of engineering, but an unusual one as it must deal with active, intelligent and malevolent opposition. [25]

4.1.2: Terminology: [25]

The original information which is to be protected by cryptography is called the ***plaintext***. ***Encryption*** is the process of converting plaintext into an unreadable form, termed ***ciphertext***, or, occasionally, a ***cryptogram***. ***Decryption*** is the reverse process, recovering the plaintext back from the ciphertext. *Enciphering* and *deciphering* are alternative terms. A ***cipher*** is an **algorithm** for encryption and decryption. The exact operation of ciphers is normally controlled by a **key** — some secret piece of information that customizes how the ciphertext is produced. ***Protocols*** specify the details of how ciphers (and other cryptographic primitives) are to be used to achieve specific tasks. A suite of protocols, ciphers, key management, user-prescribed actions implemented together as a system constitute a ***cryptosystem***; this is what an end-user interacts with, e.g. **PGP** or **GPG**.

In ordinary parlance, a (secret) "**code**" is often used synonymously with "cipher". In cryptography, however, the term has a specialized technical meaning: **codes** are a method for classical cryptography, substituting larger *units of text*, typically words or phrases (e.g., "apple pie" replaces "attack at dawn"). In contrast, classical ciphers usually substitute or rearrange individual *letters* (e.g., "attack at dawn" becomes "buubdl bu ebxo" by substitution.).

4.1.3: Cryptanalysis:

A cryptanalyst might appear to be the natural adversary of a cryptographer. However, it is possible, in fact preferable, to interpret the two roles as complementary: a thorough understanding of cryptanalysis is necessary to create secure cryptography. [25]

There are a wide variety of cryptanalytic attacks, and it is convenient to classify them. One distinction concerns what an attacker can know and do in order to learn secret information, e.g. does the cryptanalyst have access only to the ciphertext? Does he also know or can he guess some corresponding plaintexts? Or even: Can he *choose* arbitrary plaintexts to be encrypted? [25]

Cryptanalysis [18]

The study of methods of defeating cryptosystems, including:

- The extraction of private information from an encrypted message by unauthorized means.
- The unauthorized alteration of encrypted data. or
- The impersonation of a participant in the information exchange.

Cryptology = **Cryptography** + **Cryptanalysis**

4.1.4: Secure communications [25]

Cryptography is commonly used for securing communications. Four desirable properties are:

1. **Confidentiality**, also known as **secrecy**: only an authorized recipient should be able to extract the contents of the message from its encrypted form. Otherwise, it should not be possible to obtain any significant information about the message contents.
2. **Integrity**: the recipient should be able to determine if the message has been altered during transmission.
3. **Authentication**: the recipient should be able to identify the sender, and verify that the purported sender actually did send the message.
4. **Non-repudiation**: the sender should not be able to deny sending the message.

Cryptography can provide mechanisms to help achieve all of the above. However, some goals aren't always necessary, practical or even desirable in some contexts. For example, the sender of a message may wish to remain anonymous; clearly non-repudiation would be inappropriate in that case.

4.2: Types of Cryptosystem: [47]

Cryptology is the hiding of information so that it is unintelligable to those we do not wish to read it and intelligable to those we do. In this section we present the fundamental or basic ideas needed to understand the science.

We start with the question, what information are we trying to hide? The answer can range from any number of things; however most importantly it is information that we want to keep private. For instance, bank account numbers and social security numbers are pieces of information that we don't want everyone to be able to obtain. For military purposes, anybody can see the need to keep plans of attack secret from the enemy, and cryptology offers a way to do this while still using standard lines of communication.

The most elementary idea in cryptology is the idea of a cryptosystem. This is a system in which information can be made unintelligible to all but the intended reader. The first component of a cryptosystem is the original set of information, called the **plaintext**. This may be the orders to attack, or the account number that we want to keep hidden from prying eyes. The next element of a cryptosystem is the algorithm, commonly known as the **cipher**. This is the process that makes the information unreadable to the common person. There are many ciphers and many kinds of ciphers; however for the most part all of them have the same purpose. The next part of a cryptosystem is the information that has been altered, which we call the **ciphertext**. This is the information that is not recognizable, and therefore can be sent out over public channels without fear of anybody understanding it. A good representation of a cryptosystem is as follows:

plaintext -> F -> ciphertext -> G -> plaintext

where F and G are functions or the ciphers.

The cipher can be very complex or very simple. Many common ciphers involve substituting one element of the information for another. For instance we have the message:

"THERE IS A WAR IN KASHMIR"

If we replace every 'A' with 'Z', we get

"THERE IS Z WZR IN KZSHMIR"

Thus, the text now doesn't make much sense (however it is easy to figure out what the message says). So usually substitutions are made for the entire alphabet.

There are also ciphers that use numbers to mask the information. This is done by first translating the alphabet into a number system, such as A = 1, B = 2, C = 3,...,Z = 26. For example, the message "A DOG" written numerically is

"1 4157"

It is very confusing and most would not even think to translate it into a series of letters. This truly is the idea behind cryptology. Currently, because of computers, most modern cryptosystems all involve mathematical ciphers. This means that usually ciphertext looks like nothing more than a set of 1's and 0's, which is very hard to decode. This is why there is such security in digital data, such as electronic bank accounts.

Within cryptology, there are two sources of study, cryptography, and cryptanalysis. **Cryptography** is the study of encryption, or the transformation of information into unintelligible code. Cryptographers are the people who invent the ciphers. These people are the codemakers. **Cryptanalysis** is the study of decryption, or the breaking of those codes. It is usually done without the key. Cryptanalysists are codebreakers.

Cryptology now plays very important roles in today's society. Traditionally, cryptology has been purely a military matter; however currently it is part of our everyday lives.

4.2.1: Symmetric key cryptography: [25]

Symmetric key ciphers use the same key for encryption and decryption, or a little more precisely, the key used for decryption is "easy" to calculate from the key used for encryption. Other terms include "private-key", "one-key" and "single-key" cryptography.

Symmetric key ciphers can be broadly grouped into **block ciphers** and **stream ciphers**. Stream ciphers encrypt one bit at a time, in contrast to a block cipher, which operates on a group of bits (a "block") of a certain length all in one go. Depending on the mode of operation, block ciphers can be implemented as self-synchronizing stream ciphers (CFB mode). Likewise, stream ciphers can be made to work on individual blocks of plaintext at a time. Thus, there is some duality between the two. The block ciphers **DES**, **IDEA** and **AES**, and the stream cipher **RC4**, are among the most well-known symmetric key ciphers.

Other cryptographic primitives are sometimes classified as symmetric cryptography:

- **Cryptographic hash functions** produce a **hash** of a message. While it should be easy to compute, it must be very difficult to invert (**one-way**), though other properties are usually needed as well. **MD5** and **SHA-1** are well-known hash functions.
- **Message authentication codes** (MACs), also known as *keyed-hash functions*, are similar to hash functions, except that a key is needed to compute the hash. As the name suggests, they are commonly used for message authentication. They are often constructed from other primitives, such as block ciphers, unkeyed-hash functions or stream ciphers.

4.2.2: Public key cryptography (Asymmetric key cryptography): [25]

Symmetric key encryption has a troublesome drawback — two people who wish to exchange confidential messages must share a secret key. The key must be exchanged in a secure way, and not by the means they would normally communicate. This is usually inconvenient, and public-key (or asymmetric) cryptography provides an alternative. In public key encryption there are two keys used, a *public* and a *private* key, with the public key for encryption and the private key for decryption. It must be "difficult" to derive the private key from the public key. This means that someone can freely send their public key out over an insecure channel and yet be sure that only they can decrypt messages encrypted with it.

Public key algorithms are usually based on hard mathematical problems. **RSA**, for example, relies on the (conjectured) difficulty of factorization. For efficiency reasons, *hybrid* encryption systems are used in practice; a key is exchanged using a public-key cipher, and the rest of the communication is encrypted using a symmetric-key algorithm (which is typically much faster). **Elliptic curve cryptography** is a type of public-key algorithm that may offer efficiency gains over other schemes.

Asymmetric cryptography also provides mechanisms for digital signatures, which are way to establish with high confidence (under the assumption that the relevant private key has not been compromised in any way) that the message received was sent by the claimed sender. Such signatures are often, in law or by implicit inference, seen as the digital equivalent of physical signatures on paper documents. In a technical sense, they are not as there is no physical contact nor connection between the 'signer' and the 'signed'. Properly used high quality designs and implementations are capable of a very high degree of assurance, likely exceeding any but the most careful physical signature. Examples of digital signature protocols include **DSA** and **ElGamal**. Digital signatures are central to the operation of **public key infrastructure** and many network security schemes (e.g., **Kerberos**, most **VPNs**, etc). Like encryption, *hybrid* algorithms are typically used in practice; rather than signing an entire document, a cryptographic hash of the document is signed instead.

4.3: Public Key Cryptosystem:

It is the second type of cryptosystem the following lines explain it.

4.3.1: Introducing Public Key Cryptosystems: [48]

An encryption key is a critical piece of information that defines the method in which the message is encoded. A decryption key defines how the message is to be decoded. For example, the Caesar shift uses a shift transformation of three; therefore, the encryption key is the number three and the decryption key is negative three. (Recall that -3 = 23 mod 26). In any affine transformation

$$C = f(P) = (aP + b) \bmod N,$$

in which a and N are relatively prime, the encryption key consists of the values a, b and N. Recall that in order to decode the message, the inverse function (also called the decoding function) must be calculated:

$$f^{-1} = (a^{-1} C - a^{-1} b) \bmod N.$$

The decryption key, then, consists of a^{-1}, b and N. As a last example, suppose that Alice sends Bob a message enciphered using a shift transformation. There are, of course, 25 different possible shifts she could use to encode her message. The shift she chooses is called the encryption key. To decode the message, Bob must use a shift of the same size in the opposite direction.

In private keys cryptosystems, security is compromised if, besides from the sender and the receiver of the message, a third party has access to the cipher key. So, in the last example above, as long as only Alice and Bob know this key, the message is considered secure. However, as we have seen, even encryption systems that are considered secure can be decoded by unintended recipients. A simple shift transformation can be easily broken by a third party, even if the key is unknown to that party. The third party simply checking at most 25 different shifts until a part or all of the message is decoded.

There is a weakness to this type of message encoding. Namely, there is no defense mechanism for invasions by third parties knowing the private encryption and decryption keys. As soon as one or both of the private keys are known to someone other than the sender-receiver team, there are no guarantees that the message will remain unreadable to outsiders.

This weakness is not inherent within public key cryptosystems. A public key cryptosystem is an encoding method in which the encryption key is publicized. In a private key cryptosystem, public knowledge of either or both keys can lead to the decoding of the message. In a public key cryptosystem, however, knowing the encryption key does not compromise the security, as determining the encryption key using only the knowledge of the encryption key is computationally feasible, but highly inefficient.

4.3.2: Public-key cryptography Technology: [31]

An encryption method that uses a two-part key: a public key and a private key. To send an encrypted message to someone, you use the recipient's public key, which can be sent to you via regular e-mail or made available on any public Web site. To decrypt the message, the recipient uses the private key, which he or she keeps secret. Contrast with "secret key cryptography," which uses the same key to encrypt and decrypt.

Public-key cryptography is a form of modern **cryptography** which allows users to communicate securely without previously agreeing on a shared secret key. For most of the history of cryptography, a key had to be kept absolutely secret and would be agreed upon beforehand using a secure, but non-cryptographic, method; for example, a face-to-face meeting or a trusted courier. There are a number of significant practical difficulties in this approach to distributing keys. Public-key cryptography was invented to address these drawbacks - with public-key cryptography, users can communicate securely over an insecure channel without having to agree upon a key beforehand.

Public-key algorithms typically use a pair of two related keys — one key is private and must be kept secret, while the other is made public and can be widely distributed; it should not be possible to deduce one key of a pair given the other. The terminology of "public-key cryptography" derives from the idea of making part of the key public information. The term **asymmetric-key cryptography** is also used because not all parties hold the same information. Some public-key algorithms operate a little differently, and use other methods to enable parties to agree on secret keys without having previously exchanged key material.

Public-key cryptography has two main applications. First, is **encryption** keeping the contents of messages secret. Second, **digital signatures** can be implemented using public key techniques. Typically, public-key techniques are much more computationally intensive than symmetric algorithms.

4.3.3: History: [31]

The first asymmetric key algorithm was invented, secretly, by **Clifford Cocks** (then a recent mathematics graduate and a new staff member at **GCHQ** in the UK) early in the 1970s, and reinvented by **Rivest**, **Shamir** and **Adleman** all then at **MIT**. Their work was published in 1976, and the algorithm was named **RSA** after the initials of their last names. Since then, several other asymmetric key algorithms have been developed, but the most widely known remains Cocks/RSA. It uses exponentiation modulo a product of two large primes to encrypt and decrypt. The public key exponent differs from the private key exponent, and determining one from the other is believed to be fundamentally hard without knowing the primes; these are in turn (if large enough) computationally infeasible to determine at the current state of the computer hardware and large integer factorization arts. Another algorithm is ElGamal (invented by Taher ElGamal then of **Netscape**) which relies on the (similar, and related) difficulty of the discrete logarithm problem. A third is a group of algorithms based on **elliptic curves**, first discovered by Neal Koblitz in the mid '80s.

4.3.4: Security: [31]

Regarding security, there is nothing special about asymmetric key algorithms. There are good ones, bad ones, insecure ones, etc; none have been proved secure in the absolute sense the one-time pad has, and some are known to be quite insecure. As with all cryptographic algorithms, these algorithms must be chosen and used with care.

4.3.5: Applications: [31]

The most obvious application is **confidentiality**; a message which a sender encrypts using the recipient's public key can only be decrypted by the recipient's paired private key.

In many cases, public-key algorithms can be used for sender **authentication**. For instance, a user can encrypt a message with her own private key and send it. That it can be decrypted using the corresponding public key provides assurance than that user (and no other) sent it.

Similarly, in the other direction, a user can be assured that a message using the proper key originates from a specific source.

4.3.6: Practical considerations: [31]

Note that so far, all these algorithms are very computationally costly, especially in comparison with many symmetric key algorithms of essentially equivalent security. This fact has important implications for their practical use. Most are used in hybrid cryptosystems for reasons of efficiency.

4.3.7: Examples: [31]

Examples of well-regarded **asymmetric key algorithms** include:

- **Diffie-Hellman**
- **RSA** encryption algorithm
- **ElGamal**
- **Elliptic curve cryptography**
- **Paillier cryptosystem**

Examples of not well regarded **asymmetric key algorithms** include:

- **Merkle-Hellman** the 'knapsack' algorithms

Examples of protocols using **asymmetric key algorithms** include:

- **DSS** (Digital Signature Standard), which incorporates the Digital Signature Algorithm
- **PGP**
- **GPG** an implementation of **OpenPGP**
- **ssh**
- **Secure Socket Layer** now implemented as an **IETF** standard **- TLS**

4.4: RSA Public Key Cryptosystem:

It is one of the most famous asymmetric key algorithms.

4.4.1: Introduction to RSA: [48]

RSA stands for Rivest-Shamir-Adleman, the creators of this public key cryptosystem. Suppose Alice would like to send a message to Bob, and suppose that each letter m_i in the message has already been transformed into a positive integer. Bob, being the receiver, must choose two prime numbers, which are called p and q. The larger p and q are, the harder the code is to break. Typically, p and q are chosen to be numbers of more than 100 digits. Bob then computes the product $N = pq$. Then he computes the product $(p - 1)(q - 1)$. Then he finds a composite integer k (k is an integer which is not prime), that satisfies the equivalence

$$k = \text{mod } (p - 1)(q - 1).$$

As k is a composite, it has at least two factors other than k and 1. Bob picks two of these factors such that their product is k, and calls them e and d, for 'encoding' and 'decoding'.

For a plaintext letter P, the encoding function is

$$C = \mathrm{E}(P) = P^e \text{ mod } N.$$

For a ciphertext letter C, the decoding function is

$$P = \mathrm{D}(C) = C^d \text{ mod } N.$$

Let us try an example. Bob first chooses the primes $p = 2$ and $q = 11$. Bob then computes $N = 22$, and $(p - 1)(q - 1) = 10$. As

$$11 = 1 \text{ mod } 10,$$

Bob could choose *k* to be 11, but then remembers that *k* has to be a composite integer. So Bob tries again, and finds that

$$21 = 1 \bmod 10.$$

Because 21 is not a prime, Bob chooses *k* to be 21. As 21 = 3 * 7, Bob lets *e* = 7 and *d* = 3. Bob then makes public his encryption key, which is the pair *e* = 7 and *N* = 22, but keeps the decryption letter *d* private.

Now Alice prepares to encrypt her message using the public encryption key. The first letter in Alice's message is represented by the number 2. So the first letter is encoded as the number 18, because

$$18 = E(2) = 2^7 \bmod 22.$$

When Bob receives the message, he begins to decode the first letter:

$$2 = D(8) = 18^3 \bmod 22.$$

Looking back at our example above, suppose the letter to be encoded was represented by a larger number than 2, such as 14. Then to encode, we would not need to solve the equivalence

$$20 = 14^7 \bmod 22.$$

Clearly, the multiplication 14^7 is not efficiently calculated by hand, and the use of a calculator does not guarantee a correct solution when the values become too big. To ensure that such calculations can be efficiently and correctly computed, we present a series of shortcuts in the following lines.

4.4.2: Tools for RSA Public Key Cryptography:

As we had discuss **modular arithmetic** in the previous chapter, in the following lines we will first examine operations in **modular arithmetic** and **modular exponentiation** briefly, which underlie the RSA encryption procedure, second, an example of **RSA encryption** and **decryption,** third **creation of very large keys in the RSA system using random function,** and finally explain an algorithm. In this project we want to explain how to create very large keys in the RSA system, and discuss that our new data type makes the system difficult to break.

4.4.2.1: Modular Arithmetic and modular exponentiation:

Recall the reduction of a number a by an integer N is defined to be the remainder upon division of a by N. For example, the reduction of 32 by 26 is 6. We say that 32 is equivalent to 6 modulo 26, or write 32 = 6 mod 26. The number N is called the **modulus**. So far, all of our reductions have used $N = 26$ to model the 26 letters of the alphabet. However, modular arithmetic can be performed using any positive integer greater than 0. For instance, 18 = 6 mod 12. Observe that 18 is the representation of 6:00 PM using a 24-hour clock.[48]

Modular arithmetic refers to performing basic arithmetic operations under a certain modulus. For example, consider (10 + 8) mod 12. This is the same as 18 mod 12, which is equivalent to 6. Similarly, (2)(12) mod 11 is the same as 24 mod 11, which is equivalent to 2. In other words, we perform the operation first, and then find the remainder of this result upon division by N.[48]

There are shortcuts which simplify the process of performing modular arithmetic. The first is given below.

$$(a)(b) \bmod N \text{ is the same as } (a \bmod N)(b \bmod N)$$

As an example, the expression (729)(9) mod 7 can be rewritten as the expression (729 mod 7)(9 mod 7). We know that

$$729 \bmod 7 = 1$$

and

$$9 \bmod 7 = 2.$$

Therefore, we have

$$(729)(9) \bmod 7 = (1)(2) \bmod 7 = 2 \bmod 7.$$

A second shortcut is given as follows.

$$a^m \bmod N \text{ is the same as } (a \bmod N)^m.$$

For instance, to calculate 9^4 mod 7, we can either multiply out 9^4 and then reduce that value modulo 7, or we can reduce modulo 7 and then raise this value to the fourth power. Clearly, the second option is easier to calculate. So,

$$9^4 \bmod 7 = (2 \bmod 7)^4 = 16 \bmod 7 = 2 \bmod 7,$$

This is quite an easy calculation. [48]

Note that modular addition and multiplication inherit the rules of *commutatively*

$$a + b = b + a \bmod n,$$

$$ab = ba \bmod n$$

and *associatively*

$$(a + b) + c = a + (b + c) \bmod n$$

$$(ab)c = a(bc) \bmod n$$

From regular arithmetic. [46]

Modular exponentiation enters into the RSA cryptosystem in an essential way. First, recall that in regular arithmetic one computes, for example,

$3^3 = (3^2)3 = (9)(3) = 27$. However, if we want to compute 3^3 mod 8, for instance, then we will reduce every intermediate stage in the computation: [46]

$$3^3 = 3^2 3 \bmod 8 = (1)(3) \bmod 8 = 3 \bmod 8$$

For example,

If the modulus n is 8, then $2^4 = 0 \bmod 8$.
If the modulus n is 55, then $2^8 = 36 \bmod 55$. [46]

Faced with the prospect of computing $2^8 = (2)(2)(2)(2)(2)(2)(2)(2)$, it pays to try to exploit the rules of exponentiation. In fact, we are going to be taking numbers to fantastically large powers modulo n, so it pays to note two more facts about the procedure of modular exponentiation before we go farther. These rules are analogues of familiar rules from regular arithmetic. [46]

$$(ap)^q = a^{pq} \bmod n$$
$$a^p a^q = a^{p+q} \bmod n$$

So, if we want to compute 2^8 mod 55 efficiently, one way to do it is to exploit the first rule over and over, obtaining

$$2^8 = ((2^2)^2)^2 = 36 \bmod 55$$

Note that instead of doing seven multiplications, we only have to do three. If you want to compute 2^{17}, for example, it pays to realize that $17 = 2^4 + 2^0$, and then you may

$2^{17} = ((((2^2)^2)^2)^2)2 = 7 \bmod 55$. [46]

In this program on the RSA encryption procedure we implement a procedure which amounts to the following algorithm. If you want to compute x^p relatively rapidly, examine the binary expansion $p = a_k 2^k + a_{k-1} 2^{k-1} + ... + a_0 2^0$, where $a_k = 1$, and where a_i is either 0 or 1. First, write x, which you should think of as corresponding to $a_k = 1$, and interpret the sequence of a_i's read from left to right as follows. If a_i is zero, square what you have already written, and if a_i is one, square what you have written and multiply the result by x. For example, if we want to compute x^{11}, then write $11 = (1)2^3 + (1)2 + 1 = (1011)_2$. The procedure says that $(((x^2)^2)x)^2x = x^{11}$, which is easy to verify, using the rules of exponents.[46]

4.4.2.2: RSA encryption and decryption: [46]

Now we can demonstrate the procedures of encryption and decryption in the RSA scheme.

Let $n > 1$ be a natural number. Then $ø(n)$ is the number of elements in the set $\{a : 1 \leq a < n$ and the greatest common divisor of a and n is $1\}$.

For example, if $n = 8$, then the numbers between 1 and 8 prime to 8 are 1,3,5, and 7, so $ø(8) = 4$. It is easy to see that if $n = p$, a prime, then $ø(p) = p - 1$. Common implementations of the RSA cryptosystem use the fact that if $n = p*q$ is a product of distinct primes, then $ø(n) = (p - 1)(q - 1)$. There is a general formula for $ø(n)$, but its evaluation requires knowing the prime factorization of n.

Higher Level RSA [48]

When RSA is actually used to protect private information, the primes p and q are chosen to have 100 or more digits. Efficient computer algorithms using number theory can quickly locate such large primes. Therefore, the modulus $N = pq$ which is made public, will have at least 200 digits. Even with the most efficient computer algorithms, factoring a number this large could take up several billion years. The encoding number e gives no useful information for factoring N. Thus these ciphers are very secure, as the public key gives little or no information to help break the private decryption key and hence decode the message.

The example below shows a higher level of encoding and decoding, although realistically much larger values must be used.

We choose $p = 5$ and $q = 11$. Then $N = 55$ is the modulus. Finding an integer k such that

$$k = 1 \bmod (p - 1)(q - 1)$$
$$= 1 \bmod 40,$$

yields $k = 41, 81, 121, 161$, etc. Recall that k must be factored into two integers. Although any of these values besides 41 is factorable into two integers other than 1, we decide to let $k = 161$. Let $e = 23$ and $d = 7$ (note that $ed = 161$). The encoding function is now given as

$$C = \mathrm{E}(P) = P^e \bmod N,$$

that is,

$$C = P^{23} \bmod 55.$$

To encode the message I AM HERE, we first convert the letters into the numerical string 8 0 12 7 4 17 4, and apply the function E to each of these values.

$$17 = 8^{23} \bmod 55$$

$$= (2^3)^{23} \bmod 55$$

$$= 2^{69} \bmod 55$$

$$= (2^6)^{11}(2^3 \bmod 55)$$

$$= (64 \bmod 55)^{11}(8 \bmod 55)$$

$$= (9^{11} \bmod 55)(8 \bmod 55)$$

$$= (9^2)^5 9 \bmod 55)(8 \bmod 55)$$

$$= (81 \bmod 55)^5(72 \bmod 55)$$

$$= (26^5)(17) \bmod 55$$

$$= ((26^2)^2 26(17)) \bmod 55$$

$$= ((676 \bmod 55)^2(442)) \bmod 55$$

$$= (16^2)(2) \bmod 55$$

$$= 512 \bmod 55$$

$$= 17 \bmod 55$$

This means that the number 8 is encoded as 17. To shorten the length of the computation, we present a shortcut. First, we find the prime factorization of each number to be encoded. That is,

$$8 = 2^3$$

$$12 = 2^2(3)$$

$$7 = (1)(7)$$

$$4 = 2^2$$

$$17 = (1)(17).$$

Letting $8 = 2^3$ yields $8^{23} = (2^3)^{23}$.

$$2^{23} \bmod 25$$

$$= (2^6)^3 2^5 \bmod 55$$

$$= ((64 \bmod 55)^3 \bmod 55)(2^5 \bmod 55)$$

$$= (9^3 \bmod 55)(32 \bmod 55)$$

$$= (729 \bmod 55)(32 \bmod 55)$$

$$= (14)(32) \bmod 55$$
$$= 448 \bmod 55$$

$$= 8 \bmod 55.$$

Thus, $2^{23} = 8 \bmod 55$ and each 2^{23} is replaced by 8. Now

$$8^{23} \bmod 55$$

$$= (2^{23})^3 \bmod 55$$

$$= 8^3 \bmod 55$$

$$= 512 \bmod 55$$

$$= 17 \bmod 55$$

We also use the value 2^{23} to encode the numbers 12 and 4. Namely,

$$4^{23} \bmod 55$$

$$= (2^2)^{23} \bmod 55$$

$$= (2^{23})^2 \bmod 55$$

$$= 8^2 \bmod 55$$

$$= 64 \bmod 55$$

$$= 9 \bmod 55.$$

Therefore, the number 4 is encoded as the number 9. To finish encoding the message above, we also need to find 3^{23} mod 55, 77^{23} mod 55, and 17^{23} mod 55.

Observe that the modulus in the encoding function

$$C = P^{23} \bmod 55$$

is larger than 26. If we reduce the numbers {26,...,54} modulo 26, we will end up with duplicate associations. This means that the encoded string of numbers cannot be converted back into an alphabetical text. This is typical in applications of RSA. We consider the encoded string of numbers as the ciphertext.

From these computations above, we readily conclude that the difficulty of encoding increases dramatically as the exponents increase from a one-digit integer to just a small two-digit integer. Allow yourself to think of the level of difficulty this presents to the decoder who possesses the decoding key. To any outsider who is not in possession of the decryption key, this task quickly becomes next to impossible. Finally note that in actual implementation, the RSA process is used with the integers *e* and *d* often exceeding 100 digits in length.

Observe that for larger values of *N*, factoring *N* into the primes *p* and *q*, the value of *d* cannot be determined, and the code cannot be broken. In our example, where both *N* and *e* are publicly known, it is hard to find the value of the decryption key *d*. As *N* = 55, we recognize that *p* and *q* must be 5 and 11 and *k* = 1 mod 40. As the encryption key is publicly known to be *e* = 23, it is quite simple to calculate a value for *d* with *k* = *ed* and *k* = 1 mod 40. Namely, as the first multiple of 23 which congruent to 1 mod 40 is 161 = 7 - 23, the value for *d* is 7, and the code can now be deciphered.

4.4.2.3: Generating very large keys in the RSA system using random function:

In computing, the term randomness refers to generating or using a set of truly random sequence of random numbers within some set range.
Random number generators have several important applications such as cryptography, programming and computer science.
Cryptographic algorithms maybe strong (meaning difficult to crack) or weak (meaning easier to crack).

All strong cryptography requires random numbers to generate keys.
In this research, random function has been defined to generate very large numbers.

This large numbers represent the keys that's used in public key cryptography, these keys must be random and unpredictable.

Here are the algorithms of random_bigint() and random_val(*bigint* value) functions were used in this project:

I- random_bigint():

It doesn't take any parameter and return random number.

```
1- assign constant value for a, b and m
2- for i= 0 to n
    2.1 x1=random(1000)
    2.2 insert x1 in x list ( x.mylist.push_back(x1) )
```

```
3- x= (a*x + b) % m
4- while (x<0)
        4.1 x=x+m
5- return x
```

II- random_val(*bigint* value):

This function takes one parameter (x) and returns random number less than this parameter.

```
1- Generate random number (r) using random_bigint() function .
2- While( r < x)
        2.1 r = r / x
3- Return r .
```

4.4.2.4: RSA algorithm: [42]

RSA scheme is as shown blow:

1. **Key Generation:**
    1. Random select two large prime numbers $p, q; n = p*q$
    2. Compute $\phi(n)=(p-1)(q-1)$, number of elements in z_n^*
    3. Select e, $1 < e < \phi(n)$, which is relative prime to $\phi(n)$ that's mean gcd(e, $\phi(n)$) = 1
    4. Compute d, $1 < d < \phi(n)$, such that $de = 1 \bmod \phi(n)$
    5. Public key: (e, n), secret key (d, n), the values of p, q, $\phi(n)$ should be secret
    6. n is known as the modulus.
    7. e is known as the public exponent or encryption exponent.
    8. d is known as the secret exponent or decryption exponent.

2. **Encryption**: m is plaintext message it is a positive integer, such that $m \in Z_n^*$, $C = E(m) = m^e \bmod n$

3. **Decryption**: $D(c)= c^d \bmod n$

Chapter five

Results

In this project a *bigint* data type had been designed. It had been demonstrated that a new data type could be used as any data type in computer language.

It had been implemented using C++ language. Double linked list was selected to store each number, because traversed the list in both directions.

The new data type used integer of base 1000, each number was represented in a separate list, each node in the list stored three digits.

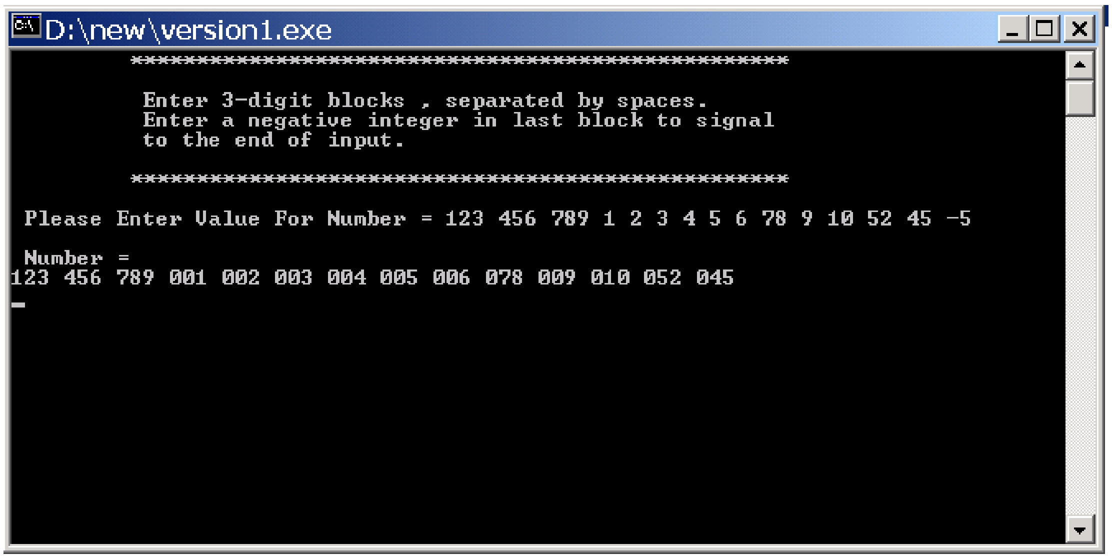
5.1: Input and Output operator:

5.1.1: Input operator:

It was the first operator we could overload. Through this operator we had entered large numbers by entering one digit at least and three digits at most separated by blank and entered a negative integer in the last block to signal to the end input.

An input operator reads these blocks and attached a node containing the value of each of these blocks to number.

Figure1: demonstrate how can we input and output our large number.



D:\new\version1.exe

```
         **************************************************

          Enter 3-digit blocks , separated by spaces.
          Enter a negative integer in last block to signal
          to the end of input.

         **************************************************

 Please Enter Value For Number = 123 456 789 1 2 3 4 5 6 78 9 10 52 45 -5

 Number =
123 456 789 001 002 003 004 005 006 078 009 010 052 045
_
```

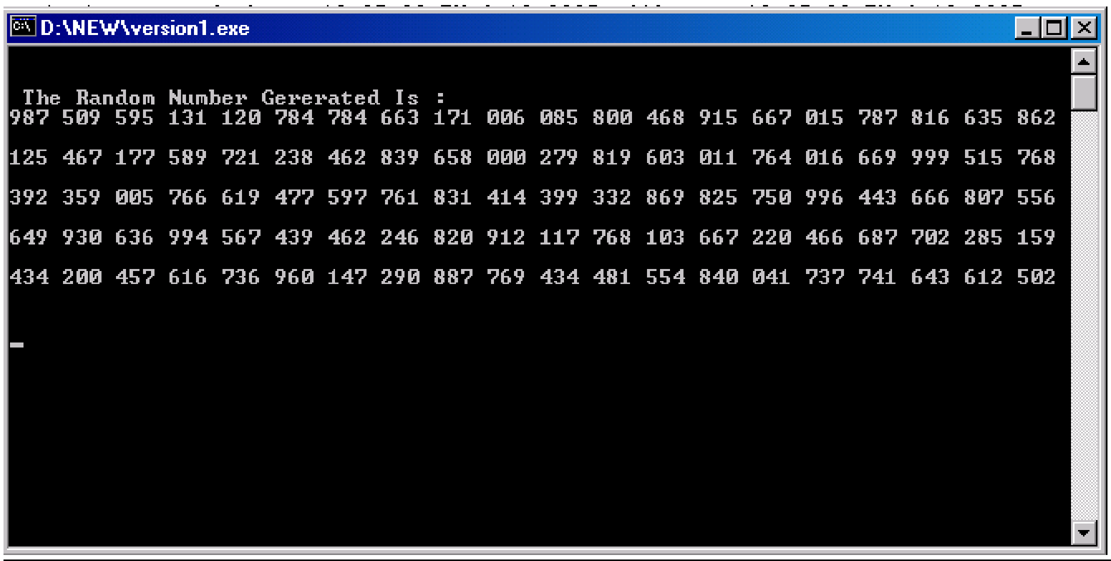
5.1.2: Out put operator:

It is an operator that displayed the contents of each node by traverse the list from left to right, as shown in figure 1.

5.2: Generating Random Number:

We had successfully built our own random function and used it in this study.

Figure 2: illustrate random number generated with about three hundred digits.



```
 The Random Number Gererated Is :
987 509 595 131 120 784 784 663 171 006 085 800 468 915 667 015 787 816 635 862

125 467 177 589 721 238 462 839 658 000 279 819 603 011 764 016 669 999 515 768

392 359 005 766 619 477 597 761 831 414 399 332 869 825 750 996 443 666 807 556

649 930 636 994 567 439 462 246 820 912 117 768 103 667 220 466 687 702 285 159

434 200 457 616 736 960 147 290 887 769 434 481 554 840 041 737 741 643 612 502
```

5.3: Arithmetic operations:

The new data type was a complete class capable of performing an arithmetic operations. An arithmetic and relational operators had been overloaded.

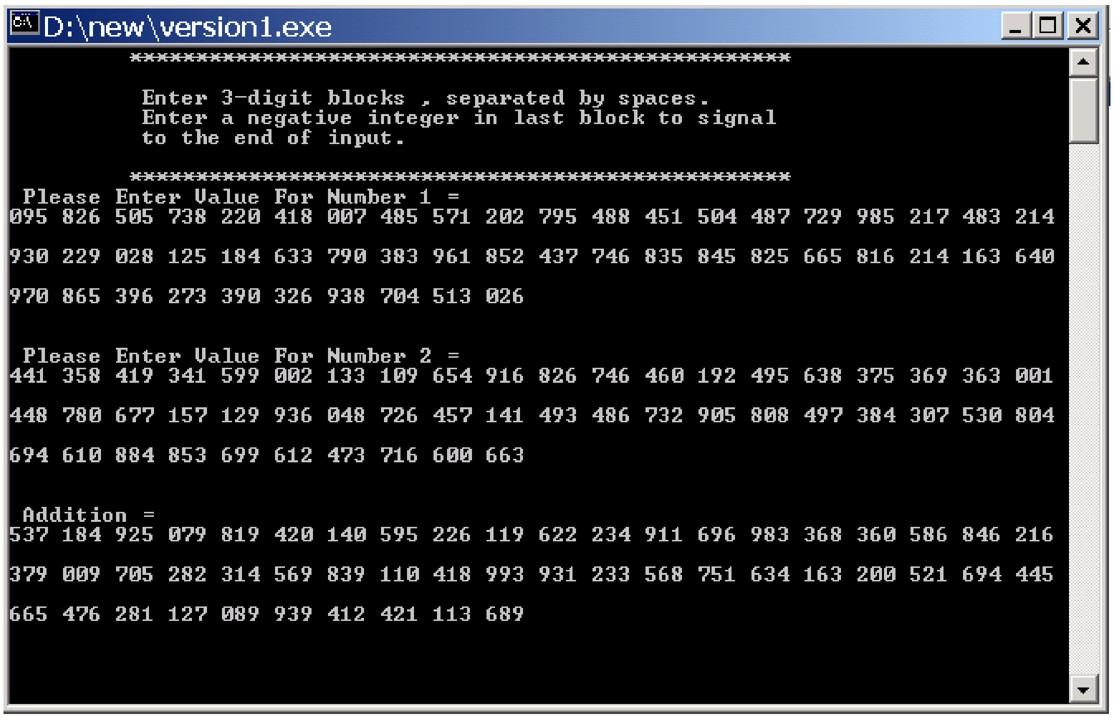
The relational operators were so important to do a comparison between numbers. The five arithmetic operations had been defined using *bigint* data type.

In each operation of the following, two large integers was generated and the result of addition, subtraction, multiplication, division and modulation could be declare:

5.3.1: Addition:

An addition of two numbers, each number consist of 150 digits.

Figure 3: Adding two large numbers.

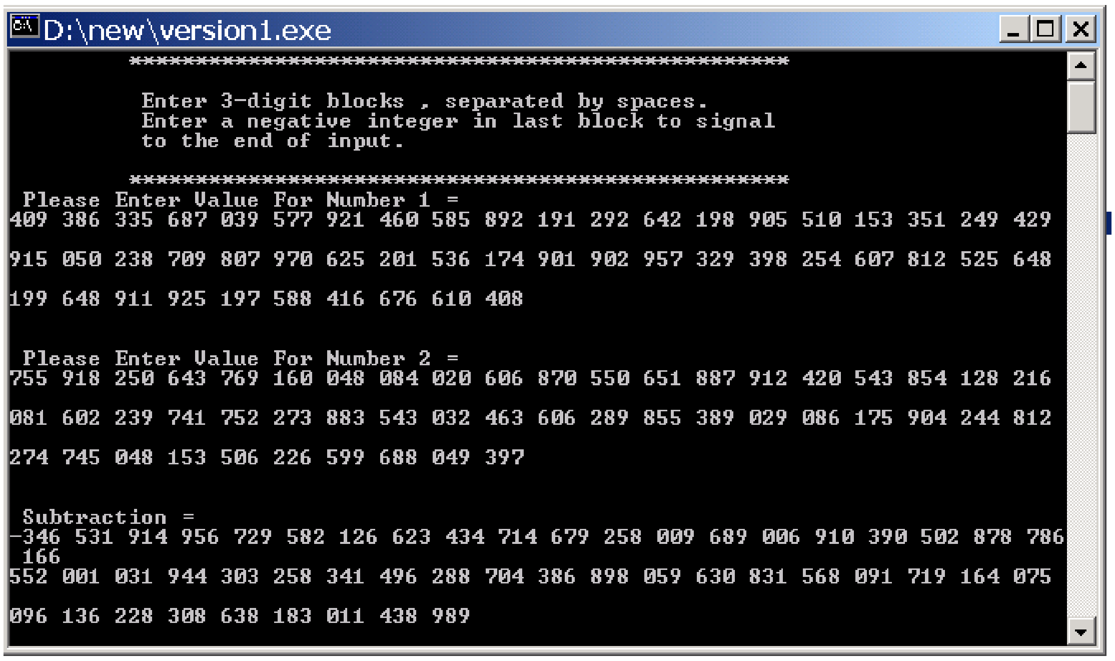


```
         **************************************************

          Enter 3-digit blocks , separated by spaces.
          Enter a negative integer in last block to signal
          to the end of input.

         **************************************************
 Please Enter Value For Number 1 =
095 826 505 738 220 418 007 485 571 202 795 488 451 504 487 729 985 217 483 214

930 229 028 125 184 633 790 383 961 852 437 746 835 845 825 665 816 214 163 640

970 865 396 273 390 326 938 704 513 026


 Please Enter Value For Number 2 =
441 358 419 341 599 002 133 109 654 916 826 746 460 192 495 638 375 369 363 001

448 780 677 157 129 936 048 726 457 141 493 486 732 905 808 497 384 307 530 804

694 610 884 853 699 612 473 716 600 663


 Addition =
537 184 925 079 819 420 140 595 226 119 622 234 911 696 983 368 360 586 846 216

379 009 705 282 314 569 839 110 418 993 931 233 568 751 634 163 200 521 694 445

665 476 281 127 089 939 412 421 113 689
```

5.3.2: Subtraction:

Subtraction two numbers, each number consist of 150 digits.

Figure 4: subtraction of two large numbers:



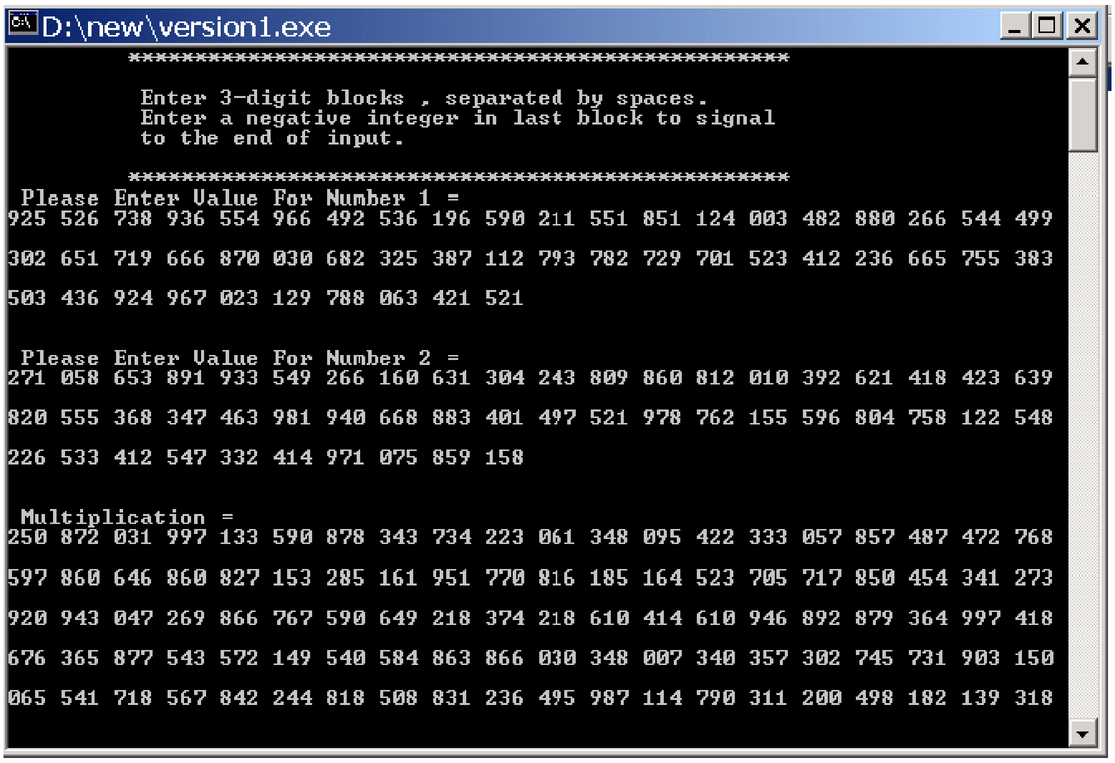
```
         **************************************************

          Enter 3-digit blocks , separated by spaces.
          Enter a negative integer in last block to signal
          to the end of input.

         **************************************************
 Please Enter Value For Number 1 =
409 386 335 687 039 577 921 460 585 892 191 292 642 198 905 510 153 351 249 429

915 050 238 709 807 970 625 201 536 174 901 902 957 329 398 254 607 812 525 648

199 648 911 925 197 588 416 676 610 408


 Please Enter Value For Number 2 =
755 918 250 643 769 160 048 084 020 606 870 550 651 887 912 420 543 854 128 216

081 602 239 741 752 273 883 543 032 463 606 289 855 389 029 086 175 904 244 812

274 745 048 153 506 226 599 688 049 397


 Subtraction =
-346 531 914 956 729 582 126 623 434 714 679 258 009 689 006 910 390 502 878 786
 166
552 001 031 944 303 258 341 496 288 704 386 898 059 630 831 568 091 719 164 075

096 136 228 308 638 183 011 438 989
```

We notice that the first number is less than number2, and the result is in negative number.

5.3.3: Multiplication:

Multiply two numbers each one consist of 150 digits, the result will be number of 300 digits.

Figure 5: Multiply two large numbers.

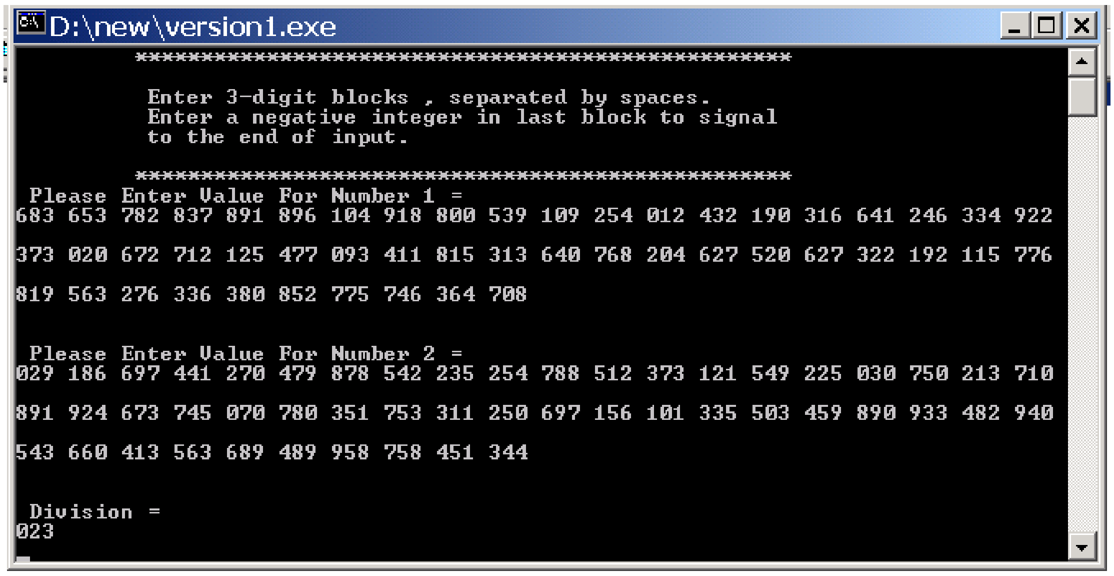


```
          ****************************************************

           Enter 3-digit blocks , separated by spaces.
           Enter a negative integer in last block to signal
           to the end of input.

          ****************************************************
 Please Enter Value For Number 1 =
925 526 738 936 554 966 492 536 196 590 211 551 851 124 003 482 880 266 544 499

302 651 719 666 870 030 682 325 387 112 793 782 729 701 523 412 236 665 755 383

503 436 924 967 023 129 788 063 421 521


 Please Enter Value For Number 2 =
271 058 653 891 933 549 266 160 631 304 243 809 860 812 010 392 621 418 423 639

820 555 368 347 463 981 940 668 883 401 497 521 978 762 155 596 804 758 122 548

226 533 412 547 332 414 971 075 859 158


 Multiplication =
250 872 031 997 133 590 878 343 734 223 061 348 095 422 333 057 857 487 472 768

597 860 646 860 827 153 285 161 951 770 816 185 164 523 705 717 850 454 341 273

920 943 047 269 866 767 590 649 218 374 218 610 414 610 946 892 879 364 997 418

676 365 877 543 572 149 540 584 863 866 030 348 007 340 357 302 745 731 903 150

065 541 718 567 842 244 818 508 831 236 495 987 114 790 311 200 498 182 139 318
```

5.3.4: Division:

Division of two numbers consist of 150 digits.

Figure 6: Divide two large numbers.

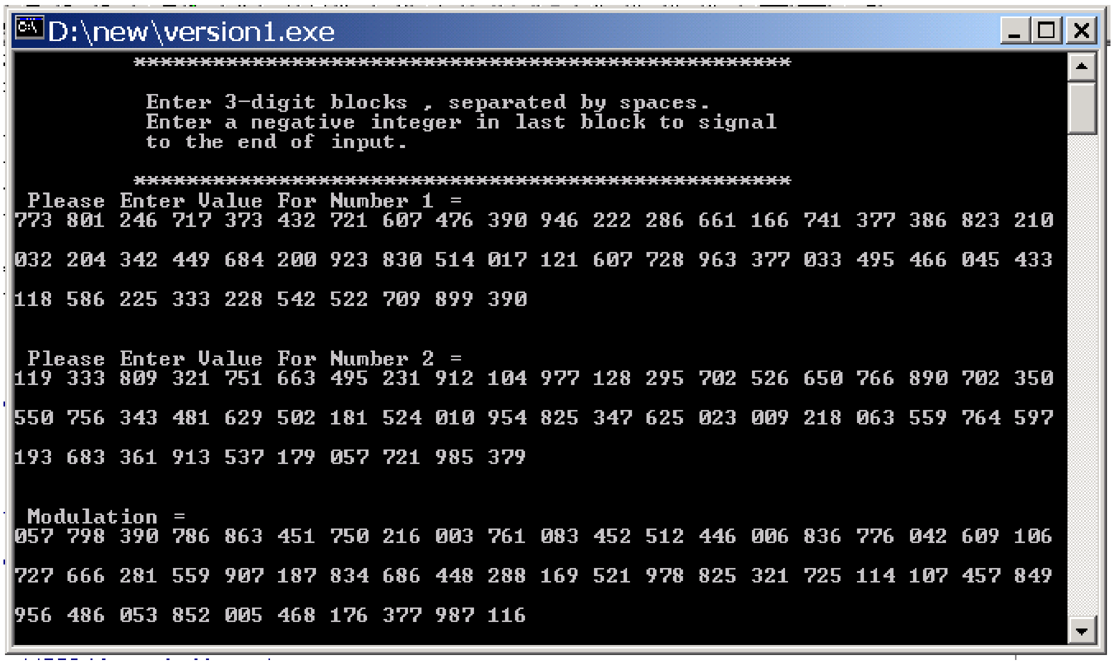


```
D:\new\version1.exe
         **************************************************

          Enter 3-digit blocks , separated by spaces.
          Enter a negative integer in last block to signal
          to the end of input.

         **************************************************
 Please Enter Value For Number 1 =
683 653 782 837 891 896 104 918 800 539 109 254 012 432 190 316 641 246 334 922

373 020 672 712 125 477 093 411 815 313 640 768 204 627 520 627 322 192 115 776

819 563 276 336 380 852 775 746 364 708


 Please Enter Value For Number 2 =
029 186 697 441 270 479 878 542 235 254 788 512 373 121 549 225 030 750 213 710

891 924 673 745 070 780 351 753 311 250 697 156 101 335 503 459 890 933 482 940

543 660 413 563 689 489 958 758 451 344


 Division =
023
```

5.3.5: Modulation:

The modulation of two numbers consist of more than 100 digits.

Figure 7: Modulation of two large numbers.



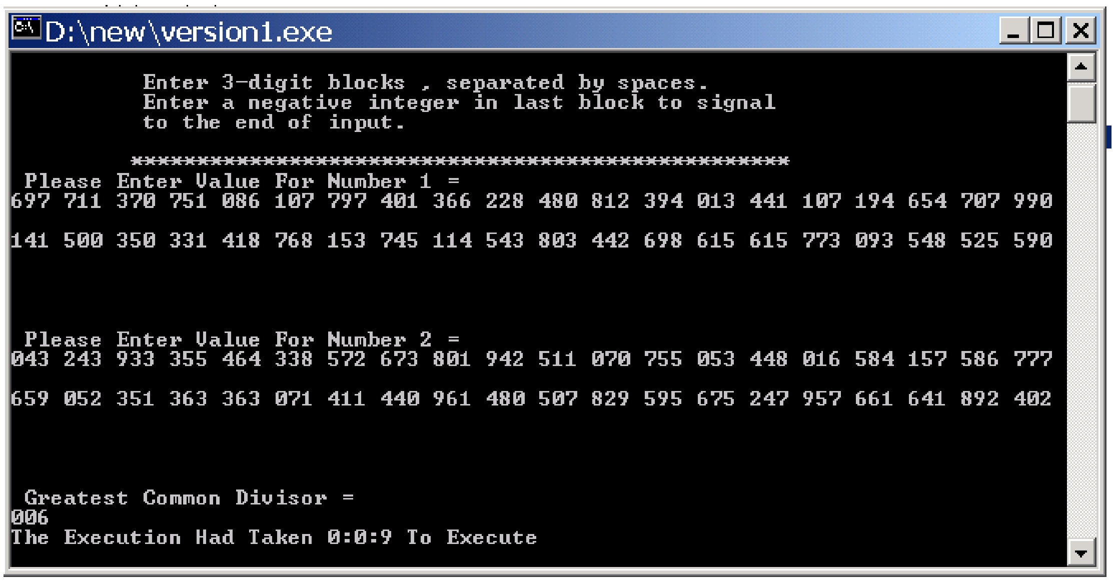
```
D:\new\version1.exe
         **************************************************

          Enter 3-digit blocks , separated by spaces.
          Enter a negative integer in last block to signal
          to the end of input.

         **************************************************
 Please Enter Value For Number 1 =
773 801 246 717 373 432 721 607 476 390 946 222 286 661 166 741 377 386 823 210

032 204 342 449 684 200 923 830 514 017 121 607 728 963 377 033 495 466 045 433

118 586 225 333 228 542 522 709 899 390


 Please Enter Value For Number 2 =
119 333 809 321 751 663 495 231 912 104 977 128 295 702 526 650 766 890 702 350

550 756 343 481 629 502 181 524 010 954 825 347 625 023 009 218 063 559 764 597

193 683 361 913 537 179 057 721 985 379


 Modulation =
057 798 390 786 863 451 750 216 003 761 083 452 512 446 006 836 776 042 609 106

727 666 281 559 907 187 834 686 448 288 169 521 978 825 321 725 114 107 457 849

956 486 053 852 005 468 176 377 987 116
```

5.4: Obtaining Greatest Common Divisor(GCD):

The need for the Greatest Common Divisor (GCD) arise in cryptography.

We had used an Euclidean Algorithm to obtain the GCD of two large integers. We were dealt with large integers, so, it took several seconds.

Figure 8: declare the GCD of two large integers and the time that it had taken.



```
          Enter 3-digit blocks , separated by spaces.
          Enter a negative integer in last block to signal
          to the end of input.

         **************************************************
 Please Enter Value For Number 1 =
697 711 370 751 086 107 797 401 366 228 480 812 394 013 441 107 194 654 707 990

141 500 350 331 418 768 153 745 114 543 803 442 698 615 615 773 093 548 525 590



 Please Enter Value For Number 2 =
043 243 933 355 464 338 572 673 801 942 511 070 755 053 448 016 584 157 586 777

659 052 351 363 363 071 411 440 961 480 507 829 595 675 247 957 661 641 892 402



 Greatest Common Divisor =
006
The Execution Had Taken 0:0:9 To Execute
```

5.5: Inverse Modulo:

As shown in previous chapter, inverse modulo was important in many applications in cryptograph.

By using *bigint* data type and Extend Euclidean algorithm we could obtain the inverse modulo of any large number.

5.6: Repeated square and multiply algorithm:

In public key cryptography the exponentiation is the most important operation. By using RSMA we had reduced the number of multiplication x^e

It had been defined and adapted by a new data type gave a perfect result in a few second.

RSMA algorithm was used in encryption and decryption operations.

5.7: Miller Rabin Primality Test:

As we had explained in previous chapter, prime number was very important in cryptography. In this study we had used Miller Rabin test to make sure that all numbers were generated as prime.

It was the most difficult operation. It was took a long of time to generate too long prime number.

5.8. RSA Public key cryptosystem:

The implementation of this study was RSA Public key cryptosystem: it could be separated into three modules:

1-Key generation.

2-Encryption.

3-Decryption.

5.8.1 Key generation:

Here is the steps had been followed:

1- Select two large prime numbers.
2- Calculate $n=p \times q$
3- Calculate $\phi(n) = (p - 1) \times (q - 1)$
4- Select large integer e, on condition that gcd $(\phi(n), e) = 1$ and $1 < e < \phi(n)$
5- Calculate d, $d = e^{-1} \bmod \phi(n)$ so, the public key {e} and private key {d} .

5.8.2: Encryption:

The encryption operation was used the public key {e} to encrypt the message {m}, which must be less than {n}.

$$\boxed{c = m^e \bmod n}$$

m = message

c = cipher

e = public key

d = private key

5.8.3: Decryption:

The cipher text was decrypted by using private key {d} to get the original message.

$$\boxed{m = c^d \bmod n}$$

Chapter six

Conclusion and Recommendation

6.1: Conclusion

It has been demonstrated that a new data type can be successfully incorporated into C++ language. We had defined a lot of functions that provide the programmer with a means to manipulate all parts of the data type.

The class *bigint* provides a multiprecision integer arithmetic. A variable of type *bigint* can hold an integer with arbitrarily many digits. We had defined the type *bigint* and we wish you can regard a variable of type *bigint* as an int variable without length restriction.

The class *bigint* is important to do a multiple-precision arithmetic routines written in C++ to carry out the usual large natural number calculations required in cryptography calculations.

A program is designed to define *bigint* class, which has been built using overloading operators for objects of a new class. Operators are defined as member functions and friend functions. The program also defines a lot of algorithms. It includes the classical multiple-precision arithmetic algorithms: addition, subtraction, multiplication, division and modulation. It also includes number theory functions such as greatest common divisor, modular arithmetic, modular exponentiation, convert any large integer number to its binary representation, check if a large integer is probably prime or not, and inverses modular.

The project applied the new data type in the RSA pubic key cryptosystem, which is excellent tests for the integrity of most of the functions in the project.

The RSA (Rivest-Shamir-Adleman) cryptosystem is widely used for secure communication in browsers, bank ATM machines, credit card machines, mobile phones, smart cards, and the Windows operating system. It works by manipulating large integers. To thwart eavesdroppers, the RSA cryptosystem will manipulate large integers (hundreds of digits). The built-in C/C++ type *int* is only capable of dealing with 16 or 32 bit integers, providing little or no security. We were designed, implemented, and analyzed an extended precision arithmetic data type that is capable of manipulating much larger integers. We will use this data type to encrypt and decrypt messages using RSA.

6.2: Recommendation

In this study *bigint* number base was 1000 and was represented in a list of type integer, it makes the calculation so easy but there was an allocated memory that's never used. We wish others to do a good manipulation.

Also, The effort made in this project was devoted to integer arithmetic, we wish others to extend this class to include real numbers. Real arithmetic is not hard to be achieved with multiple precisions floating numbers. They are so essential in performing accurate numerical computation. This mean a(*bigfloat)* class for real number arithmetic could be defined. It would require a bit of thinking, but it should not require so much extra coding because integer arithmetic and real number arithmetic are very similar. It would be a very respectable accomplishment for a novice programmer.

References

[1]almaak.usc.edu/~tbrun/course/lecture15.pdf. Downloaded on 4/2005.

[2]Art friedman, Lars Klander, Mark Michaelis, and Herb Schildt. 1999. C/C++ Annotated Archives. Tota McGraw-Hill Publishing Company Limited NEW DELHI.

[3]cs.colgate.edu/faculty/stina/courses/cosc/102/f02/labs/Lab01/Lab1.html. Downloaded on 3/2005.

[4]eee.ucc.ie/staff/marnanel/files/papers/Daly-Marnane-Popvici-ISSC2003.pdf. Downloaded on 4/2005.

[5]encyclopedia.laboralawtalk.com/Copime. Downloaded on 4/2005.

[6]Eric W. Weisstein et al. "Carmichael Number" From MathWorld--A Wolfram Web Resource. http:://mathworld.wolfram.com/CarmichaelNumber.html. Downloaded on 3/2005.

[7]Eric W. Weisstein et al. "Modular Inverse." From MathWorld--A Wolfram Web Resource. http:://mathworld.wolfram.com/ModularInverse.html. Downloaded on 3/2005.

[8]Eric W. Weisstein et al. "Prime Number." From MathWorld--A Wolfram Web Resource. http:://mathworld.wolfram.com/PrimeNumber.html. Downloaded on 3/2005.

[9]gethelp.devx.com/techtips/cpp-pro/10min/10min0599.asp.Downloaded 10/2004.

[10]Herbert Schildt. 2003. The complete Reference C++. Fourth edition. Tata McGraw. Hill Publishing Company Limited NEW DELHI.

[11]java.sun.com/JDK-1.0/api/java.Lang.Number.html#-top-. Downloaded on 4/2005.

[12]Kenneth h. Rosen. Elementary number theory and its application. 1993. Third edition. AT&T Bell Laboratories.

[13]Larry R. Uyhoff. C++ Introduction To Data Structure.

[14]math.carleton.ca/~help/Mathworks_R13Doc/Techdoc/matlab_prog/ch10-p20.htm. Downloaded on 4/2005.

[15]math.uwyo.edu/~moorhous/quantum/talk3.pdf. Downloaded on 2/2005.

[16]mayukhbose.com/tutorials/overloading/index.php. Downloaded on 3/2005.

[17]Menezee, A. ; Ocrschot, P. van and Vanstone, S. (1996). Hand book of Applied Cryptography. CRC Press. Available on www.cacr.math.uwaterloo.ca/hac/

[18]msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts_ia-iz_3ss4.asp. Downloaded on 4/2005.

[19]msdn.microsoft.com/library/default.asp?url=/library/en-us/vblr7/html/vadatinteger.asp. Downloaded on 4/2005.

[20]perl.about.com/cs/programminhtips/a/aa072301_2.htm.
Downloaded on 4/2005.

[21]primes.utm.edu/glossary/page.php?sort=RelativelyPrime.
Downloaded on 3/2005.

[22]visualbasic.about.com/od/usevb6/1/aa032903d.htm. Downloaded on 4/2005

[23]www.absoluteastronomy.com/encyclopedia/n/nu/numeral_system.htm.
Downloaded on 3/2005.

[24]www.answers.com/main/ntquery?jsessionid=6dr2m9n17klo7?method=4&dsid=2222&dekey=Primality+test&gwp=8&curtab=2222_1&sbid=lc03a.
Downloaded on 3/2005.

[25]www.answers.com/main/ntquery?method=4&dsid=2222&dekey=Cryptography&gwp=8&curtab=2222_1. Downloaded on 3/2005.

[26]www.answers.com/main/ntquery?method=4&dsid=2222&dekey=Fermat+primality+test&gwp=8&curtab=2222_1.
Downloaded on 3/2005.

[27]www.answers.com/main/ntquery?method=4&dsid=2222&dekey=Miller-Rabin=primality+test&gwp=8&curtab=2222_1. Downloaded on 3/2005.

[28]www.answers.com/main/ntquery?method=4&dsid=2222&dekey=Modular+exponentiation&gwp=8&curtab=2222_1. Downloaded on 3/2005.

[29]www.answers.com/main/ntquery?method=4&dsid=2222&dekey=Number+theory&gwp=8&curtab=2222_1. Downloaded on 3/2005.

[30]www.answers.com/main/ntquery?method=4&dsid=2222&

dekey=Pseudoprime&gwp=8&curtab=2222_1. Downloaded on 3/2005.

[31]www.answers.com/main/ntquery?method=4&dsid=2222&dekey=Public-key+cryptography&gwp=8&curtab=2222_1. Downloaded on 3/2005.

[32]www.answers.com/main/ntquery?method=4&dsid=2222&dekey=Solovay-Strassen+primality+test &gwp=8&curtab=2222_1. Downloaded on 3/2005.

[33]www.bletchleypark.net/cryptology/Number-Theory.pdf. Downloaded on 4/2005.

[34]www.cryptomathic.com/labs/rabinprimalitytest.html. Downloaded on 3/2005.

[35]www.csc.liv.ac.uk/~Frans/COMP101/week3/types.html#dataTypes. Downloaded on 4/2005.

[36]www.c-sharpcorner.com/Code/2004/sept/DataTypeandVariables.asp. Downloaded on 4/2005.

[37]www.delphibasic.co.uk/Aricle.asp?. Downloaded on 4/2005.

[38]www.eng.tau.ac.il/~yash/crypt-netsec/zuck-rsa-rabin-elgamal.pdf. Downloaded on 1/2005.

[39]www.esat.kuleuven.ac.be/~cosicart/pdf/AB-9400.pdf. Downloaded on 3/2005.

[40]www.functionx.com/objectpascal/lesson05.htm. Downloaded on 4/2005.

[41]www.gamespp.com/c/introductionToCppMetrowerkslessono8.html. Downloaded on 3/2005.

[42]www.lehigh.edu/~tkr2/teaching/ie170/lectures/lecture23.pdf. Downloaded on 3/2005.

[43]www.ma.umist.ac.uk/avb/117ws9.html. Downloaded on 3/2005.

[44]www.ma.utexas.edu/ users/ode11/ch3-numtheory.pdf. Downloaded on 4/2005.

[45]www.mast.queensu.ca/~math418/m418oh/m418oh12.pdf. Downloaded on 12/2004.

[46]www.math.nmsu.edu/crypto/public_html/BegRSA.html. Downloaded on 3/2005.

[47]www.math.nmsu.edu/crypto/public_html/Fundamentals.html. Downloaded on 3/2005.

[48]www.math.nmsu.edu/crypto/public_html/Publickey.html. Downloaded on 3/2005.

[49]www.math.umbc.edu/~campell//Numthy/class/BasicNumberthy.html. Downloaded on 3/2005.

[50]www.tacc.utexas.edu/services/userguides/pgi/pgiws-ug/pgi32u06.htm#heading71. Downloaded on 4/2005.

[51]www.tacc.utexas.edu/services/userguides/pgi/pgiws-ug/pgi32u06.htm#heading74. Downloaded on 4/2005.

[52]www1.cs.columbia.edu/~tal/4995/lect_notes/ITC6.pdf. Downloaded on 4/2005.

[53]www-Fs.informatik.uni-tuebingen.de/~reinhard/krypto/English/2.2.e.html. Downloaded on 3/2005.

[54]www-math.cudenver.edu/~wcherowi/courses/m5410/exeucalg.html. Downloaded on 4/2005.

[55]www-users.itlabs.umn.edu/classes/spring-2003/csci113/1113_Final_Project.pdf. Downloaded on 12/2004.

Appendix

```
                    ////////***************************bigint.cpp ********************
#include<iostream>
#include<iomanip>
#include<list>
#include<time.h>
#include<stdlib.h>
#include "bigint.h"
/////***************************Main Program **************
int main()
{
  bigint msg,result;
          time_t t1,t2,t3;
          int hr,min,sec;
          int i;

  zero.mylist.push_back(0);
  one.mylist.push_back(1);
  two.mylist.push_back(2);

   cout<<"    ****************************************************"<<endl;
   cout<<"     *  Enter 3-digit blocks , separated by spaces.                   *"<<endl;
   cout<<"     *  Enter a negative integer in last block to signal              *"<<endl;
   cout<<"     *  to the end of input.                                          *"<<endl;
   cout<<"     ****************************************************"<<endl;

cout<<"Enter Your MSG Please :";
cin>>msg;
t1=time(NULL);
result=RSA(msg);
t2=time(NULL);
t3=t2-t1;
hr=t3/3600;
t3=t3%3600;
min=t3/60;
sec=t3%60;
cout<<"\n The Result = \n"<<result;
cout<<"The Execution Had Taken "<<hr<<":"<<min<<":"<<sec<<" To Execute";
cin>>i;
          return 0;
}
```

```
///////////********************bigint.h*******************//////////
#include<iostream>
#include<iomanip>
#include<list>
#include<stdlib.h>

using namespace std ;
class bigint
{
	public:
	friend istream& operator>>(istream& in,bigint& number);
	friend ostream& operator<<(ostream& out,bigint& number);
	friend bool operator==(bigint number1,int number2);
	friend bool operator!=(bigint number1,int number2);
	friend bool operator==(bigint number1,bigint number2);
	friend bool operator!=(bigint number1,bigint number2);
	friend bool operator>(bigint number1,int number2);
	friend bool operator<(bigint number1,int number2);
	friend bool operator<=(int number1,bigint number2);
	friend bool operator>=(bigint number1,int number2);
	friend bool operator<=(bigint number1,int number2);
	friend bool operator<(bigint number1,bigint number2);
	friend bool operator>(bigint number1,bigint number2);
	friend bigint operator+(bigint number1,bigint number2);
	friend bigint operator-(bigint number1,bigint number2);
	friend bigint operator*(bigint number1,bigint number2);
	friend bigint operator/(bigint number1,bigint number2);
	friend bigint operator%(bigint number1,bigint number2);
	friend bigint get_value(bigint number1, bigint number2);
	friend bigint absolute(bigint number);
	friend bigint change_sign(bigint number);
	friend bigint divid(bigint number1,bigint number2);
	friend bigint pin_num(bigint number1);
	friend bigint EA(bigint number1,bigint number2);
	friend bigint EEA(bigint number1,bigint number2, bigint &d,bigint &x,bigint &y);
	friend bigint RSMA(bigint x, bigint e, bigint n);
	friend bigint calc(bigint n, bigint &s, bigint &r);
	friend bool   Prime(bigint x);
	friend bigint random_bigint();
	friend bigint random_val(bigint number2);
	friend bigint discard(bigint number);
	friend bigint num_list(int number);
	friend bigint RSA(bigint plain);
	list<int> mylist;
};
bigint zero,one,two;
////**************** insert number to list
bigint num_list(int number)
{
bigint value;
int result;
 if((number>=0)&&(number<=999))
    value.mylist.insert(value.mylist.begin(),number);
 else
   if(number>999)
   {
     while(number > 0)
		{
					result=number%1000;
					number=number/1000;
```

```
                                        value.mylist.insert(value.mylist.begin(),result);
              }
     }
return value;
}
//******overload assignment operator (==)******
bool operator==(bigint number1 ,int number2)
{
bigint number;
number1=discard(number1);
number=num_list(number2);
   if(number1==number)
                           return true;
   else
                           return false;
}
//******overload not equal operator (!=)******
bool operator!=(bigint number1 ,int number2)
{
bigint number;
number1=discard(number1);
number=num_list(number2);
 if(number1!=number)
                           return true;
 else
                           return false;
}
//******overload assignment operator (==)******
bool operator==(bigint number1 ,bigint number2)
{
number1=discard(number1);
number2=discard(number2);
int size1=number1.mylist.size(),size2=number2.mylist.size();
list<int>::iterator it1 = number1.mylist.begin();
list<int>::iterator it2 = number2.mylist.begin();
if(size1==size2)
{
 while(it1 != number1.mylist.end() && it2 != number2.mylist.end())
 {
    if((*it1)==(*it2))
       {
       it1++;
       it2++;
       }
    else
    return false;
 }
}
 else
    if(size1!=size2)
 return false;
}
//******overload not equal operator (!=)******
bool operator!=(bigint number1,bigint number2)
{
int size1=number1.mylist.size(),
     size2=number2.mylist.size();
list<int>::iterator it1 = number1.mylist.begin();
list<int>::iterator it2 = number2.mylist.begin();
if(size1==size2)
```

```
{
while(it1 != number1.mylist.end() && it2 != number2.mylist.end())
{
    if(*it1!=*it2)
    {
    return true;
    break;
    }
    else
    if((*it1)==(*it2))
    {
    it1++;
    it2++;
    }
    else     return false;
   }
}
else
 {
   if(size1!=size2)
              return true ;
              else
              return false;
 }
}
//******overload greater than operator (>)******
bool operator>(bigint number1 ,int number2)
{
bigint number;
number1=discard(number1);
number=num_list(number2);
   if(number1>number)
                        return true;
   else
                        return false;
}
//******overload less than operator (<)******
bool operator<(bigint number1 ,int number2)
{
number1=discard(number1);
list<int>::iterator it_x=number1.mylist.begin();
 if(*it_x < number2)
                        return true;
   else
                        return false;
}
//******overload greater than or equal operator (>=)******
bool operator>=(bigint number1 ,int number2)
{
bigint number;
number1=discard(number1);
number=num_list(number2);
   if((number1>number)||(number1==number))
                        return true;
   else
                        return false;
}
//******overload less than or equal operator (<=)******
bool operator<=(bigint number1 ,int number2)
{
```

```
number1=discard(number1);
list<int>::iterator it_x=number1.mylist.begin();
   if((*it_x<number2)||(*it_x==number2))
                          return true;
   else
                          return false;
}
//******overload greater than operator (>)******
bool operator>(bigint number1,bigint number2)
{
number1=discard(number1);
number2=discard(number2);

int size1=number1.mylist.size(),
              size2=number2.mylist.size();
list<int>::iterator it1 = number1.mylist.begin();
list<int>::iterator it2 = number2.mylist.begin();

if(size1==size2)
{
while(it1 != number1.mylist.end() && it2 != number2.mylist.end())
{
    if(*it1>*it2)
    {
    return true;
    break;
    }
    else
    if((*it1)==(*it2))
    {
    it1++;
    it2++;
    }
    else
    return false;
   }
}
else
{
   if(size1>size2)
             return true ;
  else
             return false;
}
}
//******overload less than operator (<)******
bool operator<(bigint number1,bigint number2)
{
number1=discard(number1);
number2=discard(number2);
bool x;
int size1=number1.mylist.size(), size2=number2.mylist.size();
list<int>::iterator it1 = number1.mylist.begin();
list<int>::iterator it2 = number2.mylist.begin();

if(size1==size2)
{
while(it1 != number1.mylist.end() && it2 != number2.mylist.end())
{
    if(*it1<*it2)
    {
    return true;
    break;
    }
    else
    if((*it1)==(*it2))
    {
    it1++;
    it2++;
    }
    else
    return false;
   }
}
else
{
  if(size1<size2)
            return true;
  else
            return false;
}
}
//******overload less than operator (<)******
bool operator<=(int number1,bigint number2)
{
bigint number;
number2=discard(number2);
number=num_list(number1);
  if(number1<=number)
                        return true;
  else
                        return false;
}
//**********define operator >>
istream& operator>>(istream& in , bigint& number)
{
  int block ;
  cin>>block;
  if (block > 999)
             cerr<<"Illegal block --" <<block<<"--ignoring \n";
  else
    number.mylist.push_back(block);
            for(;;)
                        {
                                    cin>>block;
                                    if (block < 0) return in;
                                        if (block > 999)
                                            cerr<<"Illegal block --" <<block<<"--ignoring \n";
                                        else
                                             number.mylist.push_back(block);
                        }
}
//*******************define operator <<
ostream& operator<<(ostream& out,bigint& number)
{
            out << setfill('0');
            int charcount = 0 ;
            list<int>::iterator it;
            for(it = number.mylist.begin();it !=number.mylist.end(); it++)
            {
               if(*it<0) out<<*it<<' ';
               else
               {
                        out << setw(3)<< *it << ' ';
                        charcount++ ;
                        if(charcount > 0 && charcount % 20 ==0)
                        out<< endl;
                }
            }
            out << endl;
}
//////////////////// defione absolute function
bigint absolute(bigint number)
{
number=discard(number);

list<int>::iterator it = number.mylist.begin();
if(*it<0)
{
number.mylist.pop_front();
number.mylist.insert(number.mylist.begin(),abs(*it));
}
return number;
}
////////////// change sign to negative
bigint change_sign(bigint number)
{
number=discard(number);
list<int>::iterator it = number.mylist.begin();
int x;
x=((*it) * (-1));
number.mylist.pop_front();
number.mylist.insert(number.mylist.begin(),x);

return number;
}
///*********** define function get_value
bigint get_value(bigint number1, bigint number2)
{
bigint value,num,val;
number1=discard(number1);
number2=discard(number2);

list<int>::iterator it_1 = number1.mylist.begin();
list<int>::iterator it_2 = number2.mylist.begin();

 if((*it_1<0)&&(*it_2<0))
 {
    number1=absolute(number1);
    number2=absolute(number2);
            if(number1>number2)
            {
                        value=number1 - number2;
                        num=change_sign(value);
                        val=num;
            }
            else
            if(number2>number1)
            {
                        value=number2-number1;
```

```
                        val=value;
            }
}
else
 if(((*it_1>=0)&&(*it_2>=0))&&(number1<number2))
 {
  value=number2-number1;
  num=change_sign(value);
  val=num;
 }
if(val!=zero) discard(val);
return val;
}
///************Overload Summation Operator********///
bigint operator+(bigint number1,bigint number2)
{
number1=discard(number1);
number2=discard(number2);
bigint sum,num,val,value;
int m,defsize,first,second,result,carry=0;
int size1=number1.mylist.size(), size2=number2.mylist.size(),maxsize=(size1 < size2 ? size2 : size1);
list<int>::reverse_iterator it1 = number1.mylist.rbegin();
list<int>::reverse_iterator it2 = number2.mylist.rbegin();
list<int>::iterator it_1 = number1.mylist.begin();
list<int>::iterator it_2 = number2.mylist.begin();

if((*it_1==0)&&(*it_2<0))
            sum=number2;
else
if((*it_1<0)&&(*it_2==0))
            sum=number1;
else
 if((*it_1<0)&&(*it_2<0))
   {
    number1=absolute(number1);
    number2=absolute(number2);
    value=number1+number2;
    num=change_sign(value);
    sum=num;
   }
else
 if((*it_1<0)&&(*it_2>0))
 {
            number1=absolute(number1);
            if(number1>number2)
            {
                        value=number1 - number2;
                        num=change_sign(value);
                        sum=num;
            }
            else
            if(number2>number1)
            {
                        value=number2-number1;
                        sum=value;
            }
}
else
if((*it_2<0)&&(*it_1>0))
 {
   number2=absolute(number2);
    if(number1>number2)
    {
         value=number1 - number2;
          sum=value;
    }
   else
   if(number2>number1)
   {
        value=number2-number1;
        num=change_sign(value);
        sum=num;
   }
}
else
 if((number1<number2)||(number1>=number2))
{
     if(size1 > size2)
     {
     defsize=maxsize-size2;
     for(m=0;m<defsize;m++)
     number2.mylist.insert(number2.mylist.begin(),0);
     }
     else
     {
     defsize=maxsize-size1;
     for(m=0;m<defsize;m++)
     number1.mylist.insert(number1.mylist.begin(),0);
     }
              while (it1 != number1.mylist.rend() && it2 != number2.mylist.rend())
               {
                             if (it1 != number1.mylist.rend())
                                           {
                                                         first = *it1;
                                                         it1++;
                                           }
                             else
                             first=0;
                             if(it2 != number2.mylist.rend())
                                           {
                                                         second = *it2;
                                                         it2++;
                                           }
                             else
                             second = 0;
                             int temp = first + second + carry ;
                             result = temp % 1000;
                             carry = temp / 1000;
                             sum.mylist.insert(sum.mylist.begin(),result);
      }
               if (carry > 0)
                             sum.mylist.insert(sum.mylist.begin(),carry);
     }
     sum=discard(sum);
                             return sum ;
}
/////******Overload Subtraction Operation **********////
bigint operator-(bigint number1,bigint number2)
{
bigint sub,value,num;
```

```
number1=discard(number1);
number2=discard(number2);

int x,m,defsize,first,second,result;
int size1=number1.mylist.size(), size2=number2.mylist.size(), maxsize=(size1 < size2 ? size2 : size1);
list<int>::reverse_iterator it1 = number1.mylist.rbegin();
list<int>::reverse_iterator it2 = number2.mylist.rbegin();
list<int>::iterator it_1 = number1.mylist.begin();
list<int>::iterator it_2 = number2.mylist.begin();

if((number1==zero)&&(*it_2<0))
{
 number2=absolute(number2);
 sub=number2;
}
else
if((*it_1<0)&&(number2==zero))
              sub=number1;
else
if((*it_1>0)&&(number2==zero))
              sub=number1;
else
 if((*it_1<0)&&(*it_2<0))
               sub=get_value(number1,number2);
else
 if((*it_1<0)&&(*it_2>0))
    {
      number1=absolute(number1);
      number2=absolute(number2);
      value=number1+number2;
      num=change_sign(value);
      sub=num;
    }
 else
 if((number1==zero)&&(*it_2>0))
    {
      num=change_sign(number2);
      sub=num;
    }

 else
 if((*it_1>0)&&(*it_2<0))
 {
              number2=absolute(number2);
   sub=number1+number2;
 }
 else
 if(number1<number2)
               sub=get_value(number1,number2);
 else
if(number1==number2)
              sub=zero;
 else
 if(number1>number2)
    {
       if(size1 > size2)
     {
     defsize=maxsize-size2;
     for(m=0;m<defsize;m++)
     number2.mylist.insert(number2.mylist.begin(),0);
     }
     else
     {
     defsize=maxsize-size1;
     for(m=0;m<defsize;m++)
     number1.mylist.insert(number1.mylist.begin(),0);
     }

              while (it1 != number1.mylist.rend() && it2 != number2.mylist.rend())
              {
                    if (it1 != number1.mylist.rend())
                    {
                          if(*it1 < *it2)
                          {
                             first=*it1 + 1000;
                             it1++;
                              x=*(it1) - 1;
                              *(it1)=x;
                           }
                          else
                         {
                            first = *it1;
                            it1++;
                          }
              }
              else
               first=0;
               if(it2 != number2.mylist.rend())
              {
                      second = *it2;
                      it2++;
              }
              else
                second = 0;
                int temp = first - second  ;
                 result = temp % 1000;
                 sub.mylist.insert(sub.mylist.begin(),result);
    }
}
sub=discard(sub);
  return sub ;
}
///*******Overload Multiplication Operation *********///
bigint operator*(bigint number1,bigint number2)
{
bigint number11,number22,mult;
int b,c,n,j;
int i,ii,s;
number1=discard(number1);
number2=discard(number2);
    s=0;
   unsigned long int sum[200][200],sum_num[200][200],mult_num[200];
   unsigned long int temp1,result,result1,carry,carry1;
   int size1=number1.mylist.size(),size2=number2.mylist.size(),maxsize=(size1 < size2 ? size2 : size1);
       n=size1+size2;
number11=number1;
number22=number2;

number1=absolute(number1);
number2=absolute(number2);
```

```
if((number1==zero)&&(number2>=zero))
              mult=zero;
else
if((number2==zero)&&(number1>=zero))
              mult=zero;
else
if((number1==one)&&(number2>zero))
              mult=number2;
else
if((number2==one)&&(number1>zero))
              mult=number1;
else
{
 for(i=0;i<=size1+size2+1;i++)
 {
   for(j=0;j<=size1+size2+1;j++)
   {
     sum[i][j]=0;
     sum[j][i]=0;
     sum_num[i][j]=0;
     sum_num[j][i]=0;
     mult_num[i]=0;
   }
}
       b=0;
        for(list<int>::reverse_iterator it2=number2.mylist.rbegin();it2!=number2.mylist.rend();it2++)
        {
             c=0;j=0;
          for(list<int>::reverse_iterator it1 = number1.mylist.rbegin();it1 != number1.mylist.rend();it1++)
             {
                    i=0;
                   if((c>=1)||(b>=1))
                    {
                        for(ii=0;ii<c+b;ii++)
                        {
                           sum[j][i]=0;
                          i++;
                        }
                     }
                    result1=(*it1)*(*it2);
                     while((result1>999)||(result1>0))
                     {
                          temp1=result1%1000;
                         sum[j][i]=temp1;
                          result1=result1/1000;
                          i++;
                      }
                    j++;
                    c=c+1;
              }//end first for loop
    if(s<=size2)
    {
       carry=0;
        for(i=0;i<n;i++)
         {
          sum_num[s][i]=0;
          for(j=0;j<size1+1;j++)
            {
            sum_num[s][i]=sum_num[s][i]+sum[j][i];
          }
          sum_num[s][i]=sum_num[s][i]+carry;
          carry=sum_num[s][i]/1000;
          sum_num[s][i]=sum_num[s][i]%1000;
        }
    }
   b=b+1;
   s++;
         for(i=0;i<=size1+size2+1;i++)
        {
              for(j=0;j<=size1+size2+1;j++)

                                    {
                                         sum[i][j]=0;
                                         sum[j][i]=0;
                                       }
                              }
}
      carry1=0;
      for(i=0;i<n;i++)
      {
         mult_num[i]=0;
         for(j=0;j<maxsize;j++)
         {
           mult_num[i]= mult_num[i]+ sum_num[j][i];
          }
          mult_num[i]=mult_num[i]+carry1;
          result=mult_num[i]%1000;
          carry1=mult_num[i]/1000;
          mult.mylist.insert(mult.mylist.begin(),result);
       }
     if (carry1 > 0)
              mult.mylist.insert(mult.mylist.begin(),carry1);
 }
list<int>::iterator it_11 = number11.mylist.begin();
list<int>::iterator it_22 = number22.mylist.begin();

if((*it_11<0)&&(*it_22>0))
              mult=change_sign(mult);
else
if((*it_11>0)&&(*it_22<0))
              mult=change_sign(mult);
mult=discard(mult);
 return mult;
}
///**********Overload Division Operator***********///
bigint divid(bigint number1,bigint number2)
{
bigint div,div1,div2,div3,div4,div5;
int defsize,m;
unsigned long int result,c=0;
int size1=number1.mylist.size(),size2=number2.mylist.size(),maxsize=(size1 < size2 ? size2 : size1);
     if(size1 > size2)
     {
     defsize=maxsize-size2;
     for(m=0;m<defsize;m++)
     number2.mylist.insert(number2.mylist.begin(),0);
     }
              while(number1 >= number2)
              {
```

```
              number1=number1-number2;
               c++;
              if(c==999)
              {
                 div.mylist.push_back(c);
              c=0;
      }
    }
int size=div.mylist.size();
if (size != 0)
{
  while(size > 0)
              {
              result=size%1000;
              size=size/1000;
              div1.mylist.insert(div1.mylist.begin(),result);
              }
                 div2.mylist.push_back(999);
         div3=div1*div2;
   if(c>0)
   {
      div4.mylist.push_back(c);
      div5=div3+div4;
   }
}
else
   div5.mylist.push_back(c);
return div5;
}
///**********Overload Division Operator************///
bigint operator/(bigint number1,bigint number2)
{
bigint R,base,q,q1,q3,number2_2,temp;
bigint number11,number22;
number1=discard(number1);
number2=discard(number2);

int x,i,n,size=number1.mylist.size();
int flag;

number11=number1;
number22=number2;

number1=absolute(number1);
number2=absolute(number2);
if((number1==zero)||(number2==zero)||(number1<number2))
q1=zero;
else
if(number2==one)
q1=number1;
else
if(number1==number2)
q1=one;
else
if(number1>number2)
{
R=number1;
number2_2=number2;
i=0;
n=size-1;
```

```
 do
{
     for(x=1;x<=(n-i);x++)
             number2.mylist.push_back(0);
     q.mylist.erase(q.mylist.begin(),q.mylist.end());
     q=divid(R,number2);
     list<int>::reverse_iterator it1=q.mylist.rbegin();
     q1.mylist.push_back(*it1);
     temp=q*number2;
     R = R - temp;
     i++;
     number2=number2_2;
  }while(i<=n);
}
list<int>::iterator it_11 = number11.mylist.begin();
list<int>::iterator it_22 = number22.mylist.begin();

if((*it_11<0)&&(*it_22>0))
flag=1;
else
if((*it_11>0)&&(*it_22<0))
flag=2;

 if((flag==1)||(flag==2))
  {
             q1=discard(q1);
             q1=change_sign(q1);
  }

q1=discard(q1);
 return q1;
}
///***********overload Mod operator*************///
bigint operator%(bigint number1,bigint number2)
{
bigint R,base,q,q1,q3,number2_2,temp;
int x,n,size=number1.mylist.size();
number1=discard(number1);
number2=discard(number2);

if((number1==zero)||(number2==zero)||(number2==one)||(number1==number2))
             R=zero;
else
if(number1<number2)
             R=number1;
else
if(number1>number2)
{
R=number1;
number2_2=number2;
int i=0;
n=size-1;
   do
   {
     for(x=1;x<=(n-i);x++)
           number2.mylist.push_back(0);
     q.mylist.erase(q.mylist.begin(),q.mylist.end());
     q=divid(R,number2);
     list<int>::reverse_iterator it1=q.mylist.rbegin();
     q1.mylist.push_back(*it1);
```

```
      temp=q*number2;
      R = R - temp;
      i++;
      number2=number2_2;
  }while(i<=n);
}
R=discard(R);
return R;
}
///**************Obtain Pinary_number *********
bigint pin_num(bigint number1)
{
bigint mod,number2,pnum,q;

while(number1>zero)
{
    mod=number1%two;
    list<int>::reverse_iterator it1=mod.mylist.rbegin();
    pnum.mylist.insert(pnum.mylist.begin(),*it1);
    number1=number1/two;
}
return pnum;
}
///***************Extended Euclidean Algorithem*******
bigint EEA(bigint number1,bigint number2, bigint &d,bigint &x,bigint &y)
{
bigint q,r,x0,y0,x1,y1,gcd;
bigint val1,val2,val3,vn;
vn=number2;
number1=discard(number1);
number2=discard(number2);

x0=one;
y0=zero;
x1=zero;
y1=one;
              while (number2 >zero)
              {
                  q=number1/number2;
                  val1=q*number2;
                  r = number1-val1;
                  val2=q*x1;
                  x=x0-val2;
                  val3=q*y1;
                   y=y0-val3;
                   number1=number2 ;
                  number2=r;
                  x0=x1;
                  x1=x;
                  y0=y1;
                   y1=y;
            }
d=number1;
x=x0;
y=y0;

    if(x<=0) x=x+vn;
     else
     if(x>vn) x=x-vn;
return d,x,y;
```

```
}
///***************Euclidean Algorithem*******
bigint EA(bigint number1,bigint number2)
{
bigint r;
number1=discard(number1);
number2=discard(number2);
        while(number2>zero)
        {
                r=number1%number2;
                number1=number2;
                number2=r;
        }
  if(number1!=zero)
  number1=discard(number1);
return number1;
}
//************* define random function
bigint random_bigint()
{
bigint x;
int i,x1;
randomize();
for(i=0;i<1;i++)
{
x1=random(1000);
x.mylist.push_back(x1);
}
while(x<=zero)
x=x+one;
x=discard(x);
return x;
}
//*************** define random function within sepecific range
bigint random_val(bigint number)
{
bigint r,r1;
r=random_bigint();
  if(r>number)
  while(r>number)
  {
        r=r/number;
  }
r=discard(r);
return r;
}
//******* define RSMA
bigint RSMA(bigint x, bigint e1, bigint n)
{
bigint t,y,e,squr;
e=pin_num(e1);
list<int>::reverse_iterator ite=e.mylist.rbegin();
 if (*ite == 1)
  y=x;
 else
  y=one;
  t=x;
  *ite++;

 while(ite!=e.mylist.rend())
```

```
   {
    squr=t*t;
    t= squr%n;
    if(*ite==1)
     y=(t*y)%n;
    *ite++;
   }
  y=discard(y);
 return y;
}
//************** Calculatt 2 to power s multply by r
bigint calc(bigint n1, bigint &s, bigint &r)
{
bigint c;
c=zero;
   while(n1>zero)
                {
                  if((n1%two)==zero)
                               c=c+one;
                               n1=n1/two;
                               if((n1%two)==one)
                               break;
                }
              r=n1;
              s=c;
return s,r;
}
//********Miller Rabin Primality test
bool Prime(bigint n)
{
 bool p;
 n=discard(n);
 bigint five,d,xx,y;
 five.mylist.push_back(5);
 bigint i,c,x,n1,n11,r,s,e,j,j1,t,t1,s1,e1,er,er1,n2;
list<int>::reverse_iterator it=n.mylist.rbegin();
 c=zero;
  n1=n-one;
  n11=n1;
  n2=n-two;
if((*it%2==0)||(*it%5==0))
{
p = false;
}
 else
  {
    calc(n1,s,r);

   do
    {
                x=random_val(n2);
     }while((EA(x,n)!=one)&&(x==one));
    t=RSMA(x,r,n);
    if((t!=one)||(t!=n11))
     {
               j=one;
       while((j<=s)&&(t != n11))
        {
          t1=t*t;
          t=t1 % n;
          if(t==one)
           p = false;
          j=j+one;
        }//// end while loop
       if(t!=n11)
        return false;
     }///end if
     return true;
  }
return p;
}
///************************** Discard Function
bigint discard(bigint number)
{
list<int>::iterator it2=number.mylist.begin();
int size=number.mylist.size();
if((size==1)&&(*it2==0))
number=number;
else
if(size>1)
{
 while(it2!=number.mylist.end())
 {
 if(*it2 ==0)
 {
  number.mylist.pop_front();
  it2=number.mylist.begin();
 }
 else
 break;
 }
}
return number;
}
/////////////////////*************** RSA Function*********************
bigint RSA(bigint plain)
{
bigint p,q,n,vn,e,e1,d,x,y,d1;
bigint cipher,msg;
  randomize();




////*******************************************************
///////*************************Key Generation*****************
 do
  {
   do
   {
      p=random_bigint();
      cout<<"p = "<<p;
   }while((Prime(p)==false));
    do
    {
      q=random_bigint();
      cout<<"q = "<<q;
    }while((Prime(q)==false));
  }while(p==q);
  n=p*q;
  vn=(p-one)*(q-one);
////********** Find e
  do
  {
     e1=random_val(vn);
     cout<<"e1="<<e1;
   }while((EA(e1,vn))!=one);
///*********** find d;
  e=e1;
  EEA(e,vn,d,x,y);
  cout<<" d = "<<d;
  cout<<"inverse modulus = "<<x;
   cout<<"\n p="<<p;
   cout<<"\n q="<<q;
   cout<<"\n n="<<n;
   cout<<"\n vn="<<vn;
   cout<<"Public Key ="<<e;
    cout<<"Private Key ="<<x;
////************************************************************
/////********************** Encryption  ********************
///////// Encrypt Message
              cipher=RSMA(plain,e,n);
              cout<<"Encrypted Message = "<<cipher;
///*************************************************************
///*********************Decrypt Message **********************
              msg = RSMA(cipher,x,n);
              cout<<"decrypt message = "<<msg;

return msg;
}
```