Perspectives in Communication, Embedded-Systems and Signal-Processing (PiCES) – An International Journal
ISSN: 2566-932X, Vol. 3, Issue 3, June 2019

LInteger-An Approach to Store and Manipulate Large Integers

Baswaraju Swathi
Information Science and Engineering, New Horizon College of Engineering, Bangalore, baswarajuswathi@gmail.com

Abstract: ***Arithmetic operations are elementary in computing and programming. Infinite precision arithmetic indicates the calculations that are performed on numbers whose digits of accuracy are inclined only by the existing memory of the host system. This converse with the earlier fixed-precision arithmetic originated in arithmetic logic unit hardware, usually offers amid 8 and 64 bits of precision. In computer programming, an integer overflow occurs if an arithmetic operation attempts to generate a numeric value that is beyond the range that can be indicated with a given number of digits larger than the maximum or lower than the minimum representable value. Generally the outcome of an overflow is the least significant digits of the result are stored, the result is said to wrap around the maximum. An overflow condition may give results leading to unintentional behaviour. In particular, if the possibility has not been predicted, overflows can concile a program's reliability and security. In this paper we build a system that permits storage and manipulation of large integer values. Applications may vary from storing a 100! to a large bit customer identification number.***

Keywords: ***Infinite-precision; fixed-precision; arithmetic; overflow; wrap***

I. INTRODUCTION

Emerging computer applications necessitate the processing of outsized numbers, larger than what a CPU can hold. The dominant PCs can merely operate numbers not longer than 32 bits or 64 bits [1]. This is because of the size the registers and the data-path inside the CPU. As a consequence, performing arithmetic operations such as subtraction on big-integer numbers is to some extend limited. Various algorithms considered in an attempt to resolve this problem that operate on big-integer numbers by converting into a binary representation later performing bitwise operations on single bits. These algorithms are of complexity O(n)[3] where n is the total number of bits in each one operand. Arithmetic operations are elementary in computing and programming. Infinite precision arithmetic indicates the calculations that are performed on numbers whose digits of accuracy are inclined only by the existing memory of the host system.[2]

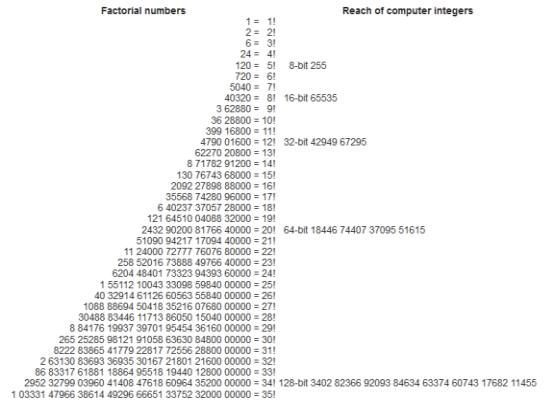

Fig 1. Factorial numbers vs. Reach of Computer integers

The above Figure 1 shows the size of factorial numbers in comparison to the usual primitive data-type limit of computers. Programming languages have built-in to maintain large numbers, and few have libraries accessible for arbitrary-precision integer and floating-point math. Relatively to store values as a fixed number of binary bits related to the size of the processor register, these implementations usually use variable-length arrays [4]of digits. Infinite precision is used in applications where the speed of arithmetic is not a restrictive parameter, where exact results with very large numbers are required.

In C++, the longest primitive data-type for most machines, which are 64-bit, is 8 bytes or 64 bits. The largest unsigned 64 bit integer is $2^{64}– 1$, which is a 20 digit number (1.84×10^{20}). If you need to deal with numbers that have a value much greater than, there is no data-type available in C++ that can handle values very large. For example, the factorial of 100 is a 158 digit number (9.332×10^{157}), which is much larger than any data-type available in C++ can handle. To overcome this issue, a class LInteger is created, which allows us to deal with large numbers. LInteger is a class used for mathematical operation which involves very big integer calculations that are outside the limit of all available primitive data types. The desired large integer can be stored. There is no theoretical limit on the upper bound of the range because memory is allocated dynamically but practically, the physical memory available is the only limit.

II. PROPOSED METHODOLOGY

The class LInteger serves as a user-defined data-type in C++ that can handle large integers efficiently with an easy-to-use syntax. The LInteger class has 2 data members. A pointer called 'val' which is a pointer that points to type long int and the other is 'length' which is of type long. The data member 'val' is used to dynamically allocate an array of long integers which are used to store the digits and the the data member 'length' specifies the length of array pointed to by 'val'. The LInteger class contains constructors that allow it to be initialized by giving the value as an integer, another LInteger object which has some value or an integer string. This allows the LInteger to be initialized to a number however large the user desires. It contains functions to add and subtract LInteger objects with another LInteger object, an integer or a string to give a signed result in the form of a LInteger object and is overloaded by the + and - operator so that it becomes very easy to use. It also has functions to multiply a LInteger object with another LInteger object, an integer or a string representing an integer to give the signed result in terms of LInteger objects and these functions are overloaded by the * operator so that it becomes easy to use them in any program.

A. Modules

- *mod(long a, long b)* – A user-defined function that returns absolute value of a modulo absolute value of b. In this program the parameter, long b, is always 10 and the parameter long a can be any long integer. Used to separate the one's digit from the remaining number. The return-type is long int.
- *digitCount(long n)* – A user-defined function that takes a long integer n and returns the number of digits in n. The return-type is n.
- *LInteger()* – A non-parameterized constructor that sets the value of data members val to NULL and length to 0.
- *LInteger(long n)* – A parameterized constructor that stores the value of long integer n in a LInteger object. The data member length will be initialized to digitCount(n) and val will point to an array containing the digits of n.
- *LInteger(string s)* – A parameterized constructor that takes a string s, which represents an integer number of any length and initializes the LInteger object with that value.
- *LInteger(constLInteger&num)* – A parameterized constructor that takes another reference of another LInteger object as an argument. This acts as a copy constructor.
- *display()* – A user-defined member function that displays the value stored in a LInteger object. The return-type is void.
- *operator = (string s)* – Operator overloading function that overloads '=' operator with a string as an argument. This is used for re-initialization of a LInteger object using a string. It basically calls the constructor that initializes LInteger object with a string. Return-type is void.
- *operator = (long n)* – Operator overloading functions that overloads '=' operator with a long integer as an argument. This is used for re-initialization of a LInteger object using an integer. It basically calls the constructor that initializes LInteger object with a long int. Return-type is void.
- *operator>> (istream&din, LInteger&a)* – Operator overloading function which is a friend of class LInteger that takes the reference variable din of an istream object and a reference variable a of LInteger object. This is used to take inputs for LInteger using cin with the regular syntax. Return-type is reference variable of istream object. This is done so that cascading can be achieved with cin.
- *operator<< (ostream&dout, LInteger&a)* – Operator overloading function which is a friend of class LInteger that takes the reference variable dout of an ostream object and a reference variable a of LInteger object. This is used to output the value of LInteger object using cout with the regular syntax. Return-type is reference variable of ostream object. This is done so that cascading can be achieved with cout.
- *operator + (LInteger a, LInteger b)* – Operator overloading function which is a friend of class LInteger. It takes 2 LInteger objects, which are the 2 operands, as arguments and calculates and returns the sum of the two arguments a and b. The return-type is LInteger.
- *operator+(LInteger&a, long b)* – Operator overloading function which is a friend of class LInteger. It takes a LInteger reference variable and a long int as arguments. It adds a LIntegera and a long integer b. The return-type is LInteger.
- *operator+ (LInteger&a, string b)* – Operator overloading function which is a friend of class LInteger. It takes a LInteger reference variable and a string representing an integer value as arguments. It returns the sum of value stored in LIntegera and string b. The return-type is LInteger.
- *operator - (LInteger a, LInteger b)* – Operator overloading function which is a friend of class LInteger. It takes 2 LInteger objects, which are the 2 operands, as arguments and calculates and returns the difference of the two arguments a and b. The return-type is LInteger.
- *operator - (LInteger&a, long b)* – Operator overloading function which is a friend of class LInteger. It takes a LInteger reference variable and a long int as arguments. It subtracts LIntegera and a long integer b. The return-type is LInteger.
- *operator - (LInteger&a, string b)* – Operator overloading function which is a friend of class LInteger. It takes a LInteger reference variable and a

string representing an integer value as arguments. It returns the diffrence of value stored in LIntegera and string b. The return-type is LInteger.

- *operator*(LInteger a, LInteger b)* - Operator overloading function which is a friend of class LInteger. It takes 2 LInteger objects, which are the 2 operands, as arguments and calculates and returns the product of the two arguments a and b. The return-type is LInteger.
- *operator* (LInteger&a, long b)* - Operator overloading function which is a friend of class LInteger. It takes a LInteger reference variable and a long int as arguments. It multiplies a LInteger a with a long integer b. The return-type is LInteger.
- *operator* (LInteger&a, string b)* - Operator overloading function which is a friend of class LInteger. It takes a LInteger reference variable and a string representing an integer value as arguments. It returns the product of value stored in LInteger a and string b. The return-type is LInteger.

III. IMPLEMENTATION

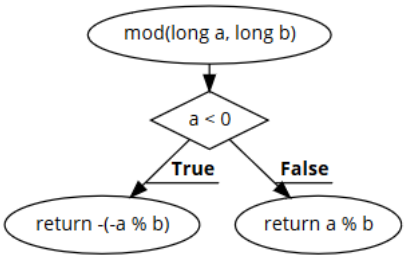

Fig 2. Flow chart for mod function (mod(long a, longb))

Figure 2 shows the flow chart for the mod function. It takes two arguments a and b of type long int. In this program b is always 10. It returns the negative of absolute value of a modulo b. If a is negative, it negates a and its modulo with b is returned as result. If a is positive, then the a modulo b is returned as result. The result is also a long int type.

Figure 3 shows the digitCount function. It takes an argument n of type long int and returns the number of decimal digits in the number n. It initially initializes a variable count to 1. Another variable tens is initialized with a value of 10. After the while loop ends, count will have the number of digits in n.

Figure 4 shows the flow chart of constructor LInteger(long n). At first, it sets the value of the data member length to the number of digits in n using digitCount() function. Now val points to an array with number of elements same as length and those elements are allocated dynamically. Now each digit is assigned to each location in the array val.

Figure 5 shows the flow chart for the constructor LInteger(string s). The string s represents an integer number as a string. If the integer is negative, the first character will be '-', that is, s[0] will be '-'. The data member length is initialized as equal to the length of the string s. if s[0] is '-', then the length is decremented by 1. The elements are stored as digits in the dynamically allocated array val. Now Another loop runs from last position of val array till 0, decrementing the value of length if the array element is 0. This is to remove the leading zeroes. For example, if user enters "000000001", it's same as 1.

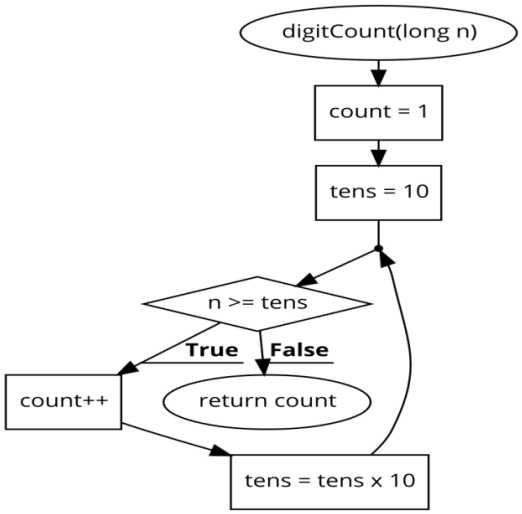

Fig 3. Flow chart for digitCount function (digitCount(long n))

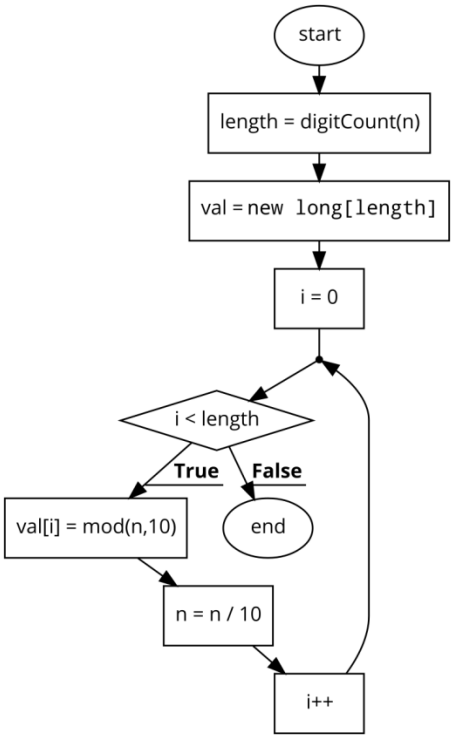

Fig 4. Flow chart for constructor LInteger(long n)

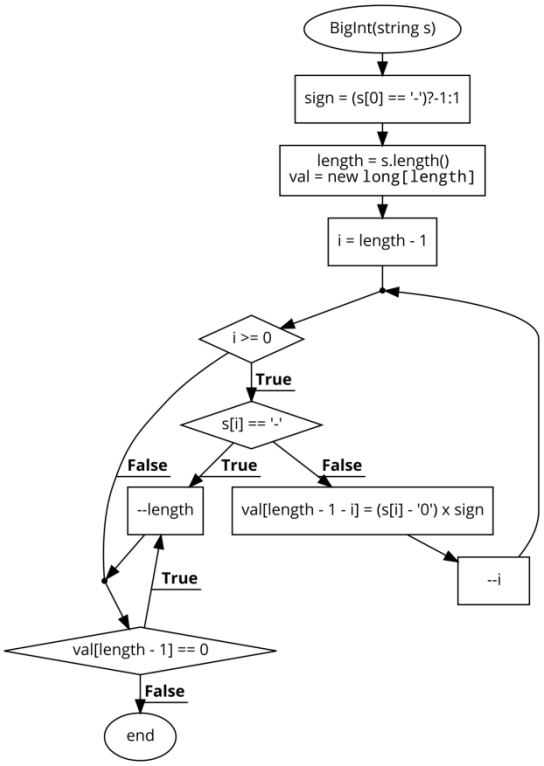

Fig 5. Flow chart for LInteger(string s)

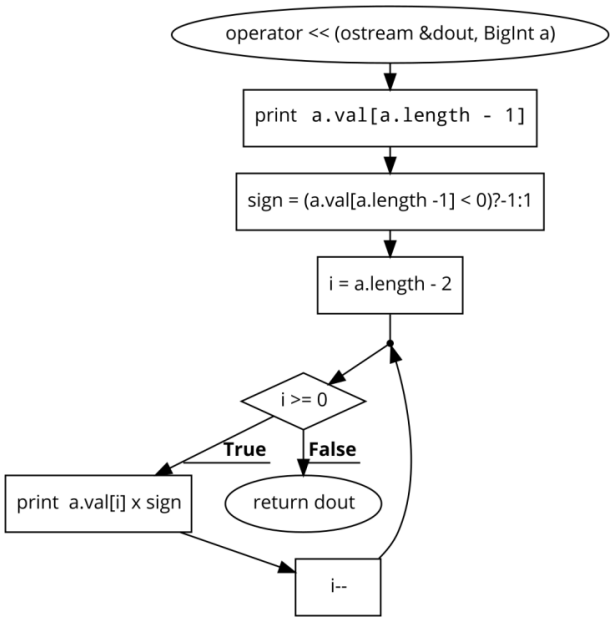

Fig 6. Flow chart of function operator << (ostream&dout, LInteger a)

The above Figure 6 shows the flow chart of function that overloads operator '<<' with ostream object reference and a LInteger object reference as its arguments. In LInteger object, if the value to be stored is negative, all the elements in val are negative. So to print the values correctly, at first the most significant digit, that is a.val[a.length – 1], is printed as it is. Then, the absolute values of all the other digits are printed. In this function, a variable sign is used to print the absolute. If the number is negative, then sign is initialized with -1. If the number is negative, sign will be initialized with 1. To get absolute value of any number, it is multiplied with sign. The function then returns the ostream object to allow cascading of stream insertion operator (<<) with cout for ease of use.

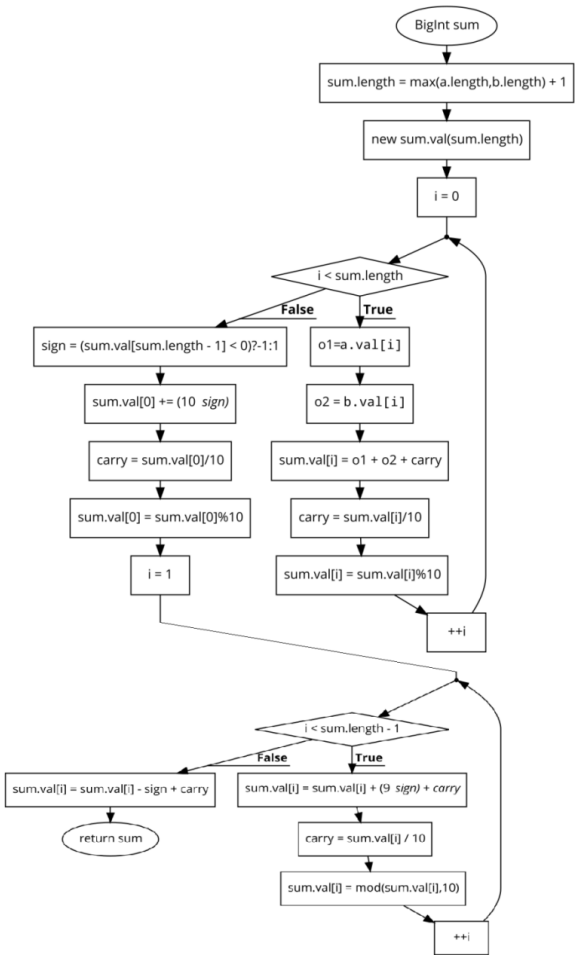

Fig 7. Flow chart of function operator + (LInteger a, LInteger b)

The above Figure 7 shows the flow chart of function that overloads opertator'+' and takes LIntegera and LInteger b as arguments. Another LInteger object called sum is created with length 1 more than the length of number which is bigger between a and b, that is, sum.length = max(a.length, b.length) + 1. Now 2 variables o1 and o2 store the digits of a and b at index i as it iterates through sum.length times. Now, o1 and o2 are added and the result module 10 is stored at index i of sum.val and result divided by 10 is taken as carry. The result which is stored in LInteger sum is returned. Addition with LInteger has a time complexity of O(n) where n is the number of digits of the greater operand.

Figure 8 shows the function that overloads '*' operator to multiply 2 large numbers by taking LIntegera andLInteger b as arguments. An object of LInteger's' is

declared which is used to store the result. The length of s is the sum of lengths of the operands, a and b, that is, s.length= a.length + b.length. The method of multiplication used here is tradition long multiplication method where every digit of the multiplier is multiplied with every digit of multiplicand. The result is stored in LInteger s which is returned. The time complexity of this function is O(nm) where n is length of LInteger a and m is length of LInteger b. For multiplication of 2 LInteger numbers of same length, the time complexity is $O(n^2)$.

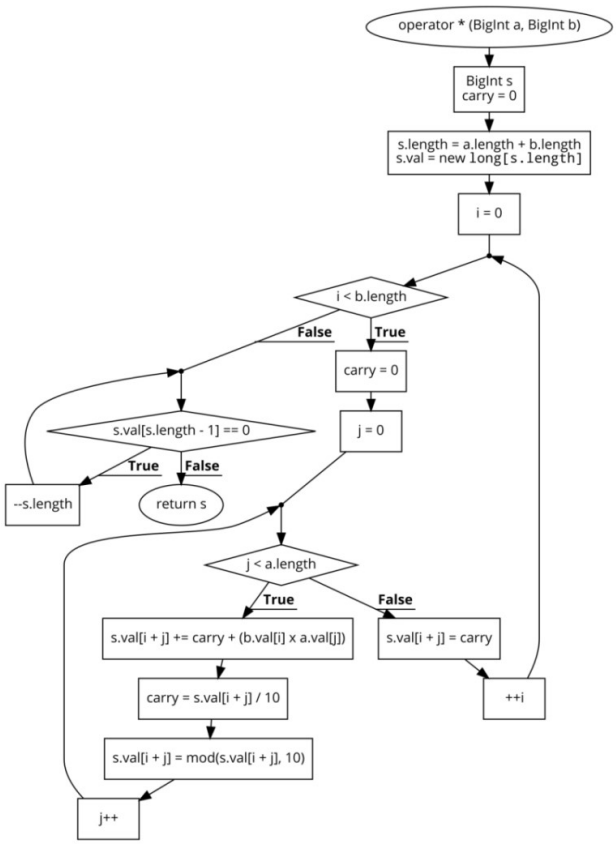

Fig 8. Flow chart of function operator * (LInteger a, LInteger b)

Figure 9 shows the flow chart of the function that overloads '*' operator and takes one LInteger object and one long int object. First, a LInteger object s is declared. The length of s is set to the sum of length of LIntegera and length of long int b which are the two operands. The long int b is multiplied with every digit of LIntegera and the result modulo 10 is stored in corresponding indices of s.val and the carry is set to result divided by 10. After multiplying the last digit with the multiplier (long int b), there can be a multiple digit value in the carry. Store each digit after the already existing most significant digit one by one. This is faster than the function in fig 3.7 as this performs multiplication in O(n) time where n is the number of digits in LInteger a.

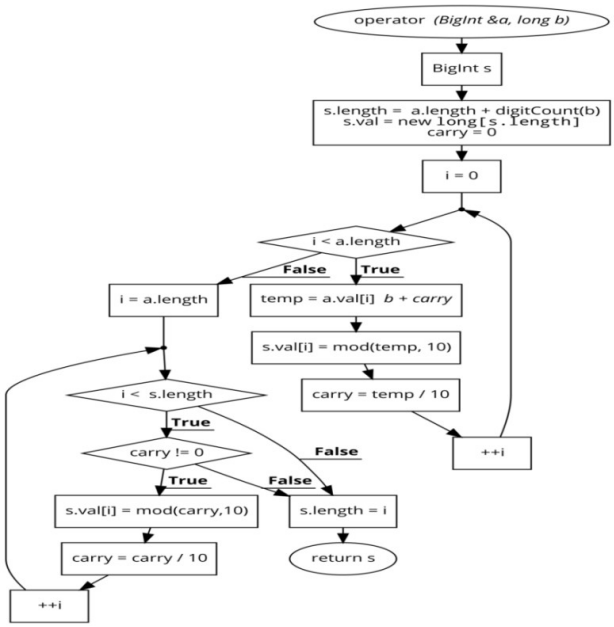

Fig9. Flow chart for function operator * (LInteger a, long int b)

IV. RESULTS AND DISCUSSSION

The Table I gives a brief description of all the variable names with their respective data types.

A. Table and Explanation

The Table1 shows all the variables used in the source code along with their description and their respective data types.

Variable	Data-type	Description
Length	long	Data member of LInteger which stores the number of decimal digits of the given number
val	long*	Data member of LInteger which is used to allocate an array to store the digits dynamically
a,b	LInteger	Operands in LInteger object form for addition, subtraction or multiplication.
b	long	Second operand for addition, subtraction or multiplication of type long int
b	string	Second operand for addition, subtraction or multiplication of type string
j,j	long	Loop iterators
sum, s	LInteger	LInteger objects which store the result to be returned by addition, subtraction or multiplication operator
count	int	Variable that stores the number of digits in a long int. This is used in function DigitCount()
carry	long	Variable used to store carry in intermediate operations of addition, subtraction and multiplication

sign	short int	Variable used to store the sign of a number in LInteger. Its value is -1 for negative and +1 for positive
fact	LInteger	LInteger object used to calculate the factorial of num
num	int	Variable that stores the number whose factorial is to be calculated
tens	long	Variable that is used to count the number of digits in digitCount() function

Table 1. Table of all variables and description used in code

The LInteger class used to calculate the factorial of a number. Factorial is a mathematical function that grows very large when the input increases. For example, 10 factorial is 362800, a 6 digit number. The factorial of 100 contains 158 digits which is number much larger than any built in data type in c++. The Figure 10 shows the output of the program that calculates factorial when input is 10 using LInteger. Figure 11 shows the output when the input number for factorial is 100.

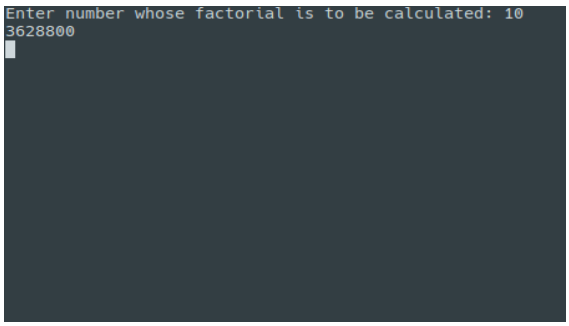

Fig 9. 10! Using LInteger

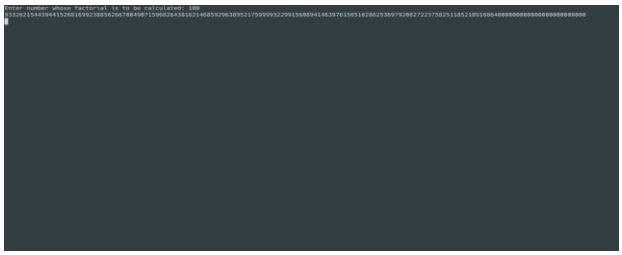

Fig 10. 100! Using LInteger

The multiplication function that multiplies 2 LInteger objects of same size has a time complexity of $O(n^2)$. There is an algorithm of multiplication using FFT(Fast fourier transform) that performs multiplication with time complexity O(nlogn)[4].

- In LInteger, the array 'val' stores one digit in one location. This is a waste of space and also for numbers that have many digits, it also takes more time to perform the calculations. It's possible to store more than 1 digit in one location of the array to save space as well as increase the speed of the calculations. For example, storing 5 digits in one location of the array uses only $1/5^{th}$ the space as the current program and the program will also run 5 times faster(Time complexity remains the same)[5].
- Other operators like division, modulo, bitwise operators, relational operators, shorthand assignment operators and increment and decrement operators can also be implemented

V. CONCLUSION

LInteger is very useful for calculations that deal with very large numbers. Factorial is one such example, where the factorial of numbers like 100 itself is a 158 digit number which far exceeds the limit of any primitive data-type in C++. There is no limit to how large a number represented by LInteger can be since the memory is dynamically allocated. But if the numbers the user is dealing with is under small and under the range of primitive data-types, then it is strongly recommended to use primitive data-types as they are considerably faster than LInteger. LInteger is useful if speed and time is not much of a concern as long as the result is given.

REFERENCES

[1] Youssef Bassil, Aziz Barbar ,"Sequential & Parallel Algorithms for Big-Integer Numbers Subtraction" , Global Journal of Computer Science and Technology Vol. 9 Issue 5 (Ver 2.0), January 2010.

[2] AP Computer Science, THE LARGE INTEGER CASE STUDY IN C++.

[3] Patrick Tan, Koen Vyverman, "Ludicrously Large Numbers - Using Arbitrary Precision Arithmetic in SAS,Applications", SAS Global Forum 2008 Applications Development.

[4] Sanjeev Gangwar1 , Prashant Kumar Yadav ,Parallel Algorithm for Adding Long Integers, Jaunpur, India, ACEIT Conference Proceeding 2016.

[5] Baswaraju Swathi, A Comparative Study and Analysis on the Performance of the Algorithms, IJCSMC,Volume 5.

[6] https://www.cs.utexas.edu/~shmat/courses/cs361s/blexim.txt

[7] http://www.xatlantis.ch/index.php/education/zeus-framework/9-big-integers