PROJECT REPORT ON

Custom Big Integer Library in C

Project Report Submitted By:

NISHIT KEDIA (University Roll No: 10400111048)
Department of Computer Science & Engineering, IEM, Kolkata

UNDER THE GUIDANCE OF
PROF. TAMAL CHAKRABORTY
Professor, CSE Dept, IEM, Kolkata

`

INSTITUTE OF ENGINEERING & MANAGEMENT
Salt Lake, Sector- V, Kolkata-700091

Affiliated To
West Bengal University of Technology

CERTIFICATE

This is to certify that the project report entitled **Custom Big Integer Library in C** , submitted by **Aloy Gupta**, **(**University Roll number : **10400111048** , University Registration number : 111040110048) ,student of **INSTITUTE OF ENGINEERING & MANAGEMENT,** submitted in partial fulfillment of the request for degree of **B.Tech in Computer Science and Engineering** is a bona fide work carried out by him under the supervision and guidance of **Prof. Tamal Chakraborty** during the 8th semester of the Academic Session of 2014-2015.The content of this report has not been submitted to any other University or Institute for the award of any other degree.

I am glad to inform that the work is entirely original and its performance is found to be quite satisfactory.

_____ _____

Prof. Tamal Chakraborty
Professor,
CSE Department,
IEM Kolkata

Prof. Dr. Debika Bhattacharyya
Head of the Department,
CSE Department,
IEM Kolkata

_____

Prof. Dr. Amlan Kusum Nayak
Principal,
IEM Kolkata

ACKNOWLEDGEMENT

It is a matter of pleasure to express our sincere gratitude to my teacher cum mentor Prof. Tamal Chakraborty for his able guidance for helping me in this project work. I am really grateful to Prof. Tamal Chakraborty, for his guidance and valuable suggestions for the entire project and during the preparation of this report.

I would also like to express our sincere gratitude to Prof. Dr. Debika Bhattacharya, Head of the Department, Computer Science & Engineering at Institute of Engineering & Management, Prof. Dr. Amlan Kusum Nayak, our respected principal sir and Prof. Satyajit Chakrabarti, Director of Institute of Engineering & Management for their support and guidance in the entire project.

There are lots of people who contributed towards the completion of the project and I am really thankful to all of them. It is not possible to acknowledge all of these people individually. I wish to thank all of our friends, department support staff and all of those who are associated with this project.

Nishit Kedia, University Roll No : 10400111048

Department of CSE,

IEM,Kolkata

ABSTRACT

The purpose of this project is to implement a Big Integer library in C , which as the name suggests help with dealing with calculations which do not fit into the conventional data types int and long making use of 32-bit and 64-bit word architectures. Calculations dealing with large numbers find application in various fields like cryptography, astronomy, statistics and ,many more. The project consists of a library of 19 functions which help the end programmer to perform arithmetical, logical and other operations on large numbers (> 2^64 -1).The API of our library is very clean and well organized. It may not be highly robust but does offer a variety of functionalities and every effort has been made to ensure that the library is bug free. It is also very easy to use from the end programmer's perspective.

Keywords : big integers, cryptography, astronomy.

CONTENTS

List of Figures :

INTRODUCTION

We all have written those simple C programs of adding two numbers during the beginning days of our experience of learning programming. No wonder, they worked perfectly and gave the correct results. But, that's because most of the time we restricted the input to small arithmetic operations like 2+3 or even 96335511-326515 which the C program could compute successfully. But, what happens if the inputs are 658943321585852415455653231 2 and 2151235532325585323354599 ?

The answer is the program prints out a garbage value or crashes. That's because there is no data type in C which can represent such large numbers which are beyond the range of (-2^ 63 to 2^63 -1).

C is a language close to the computer hardware. All its data types are hardware mapped which means let's say, a 64 bit long is represented as a 64 bit word on a 64 bit architecture machine and the arithmetic circuits in the ALU directly work on these 64 bits to produce the results.

The library of functions implemented in this project is a solution to the problem of representing and manipulating large numbers in C. The library has been named simpleBigInt.

The Problem Addressed

Here's a simple C program which adds two integers :

```
#include<stdio.h>

int main(){
long a,b,c;
scanf("%d %d",&a,&b);
c=a+b;
printf("%d",c);
return 0;
}
```

The output is :
25331
12983
38314

This works perfectly because both the inputs are less than 2^63 -1 , which is the maximum value a long can represent in modern x86-64 based architectures.

But, let's examine a different scenario :
25331 < ---------- fits into long datatype
8692215331211004415555512295275018 < ---------- doesn't fit into long

The program crashes because of the second input as long data type generally corresponds to the word size of the computer which in most modern machines is 64 bits.

The Solution : The Abstract Data Type (ADT)

```
#define BISIZE 3000
struct bigint{
        char sign;                          /* sign of the biginteger */
        char value[BISIZE+1];           /* to store the value of the bigintger of */
                                            /* BISIZE number of decimal digits */
};
/* a derived datatype for biginteger */
typedef struct bigint bigint;
```

The Working of the ADT

A struct in the C programming language is a complex data type declaration that defines a physically grouped list of variables to be placed under one name in a block of memory, allowing the different variables to be accessed via a single pointer, or the struct declared name which returns the same address. The struct is a natural organizing type for records like the mixed data types in lists of directory entries reading a hard drive (file length, name, extension, physical (cylinder, disk, head indexes) address, etc.), or other mixed record type (patient names, address, telephone... insurance codes, balance, etc.).

The custom ADT for the simpleBigInt library called ‘bigint’ is a C structure consisting of a char type variable called sign and an array of characters of length BISIZE+1. BISIZE is #define macro defining the maximum length of the big integer that is being allowed to represent. The char array indexes from 0 to BISIZE-1 is used to represent the digits of a big integer number as characters. The BISIZE array index stores the nul (‘\0’) value like any other string representation

in C. The sign variable stores the sign of the big integer (+ or -). Zero (0) is treated as a positive value by default. When big integers are taken as input, if any sign is not specified, then positive (+) sign is assumed by default. The base of representation of the ADT is decimal (base 10).

The functions specified in the library operate on variables of the bigint struct type. Some of them take parameters, while some of them even return pointers to bigint variables for further usage and manipulation.

A practical way to get more out of this representation is to increase base; this way, the same number of bits can be represented with less storage. The reason for choosing 10, other than the fact that it makes doing examples easy, is that it makes printing the numbers out a matter of traversing the array; other bases require complex conversions of bases. If we keep base as a power of 10 (e.g. 10,000), we can still easily print the numbers (fixing up leading zeros when we find them), and still improve performance.

The library of functions uses not so complex algorithms to accurately calculate various operations like addition, subtraction, multiplication, division, hcf, lcm, power, remainder, etc.

Each of the bigints take up maximum of BISIZE bytes. The default value of BISIZE is 3000 as defined in the header file.

So, size of each bigint= 3000*1 byte=**3000 bytes**=3000/1024 KB=**2.93 KB** of RAM.

For playing around with very large numbers, the Unix/Linux command bc implements "arbitrary" precision arithmetic; typing bc at the command prompt and then typing something ridiculously large like 2^1000 (2 to the 1000 power) gives the correct result quickly. Infact, the very idea of this project has occurred while trying the bc command and later realizing although the standard Java library comes with a BigInteger class, which can be used for the exact same purpose, the C Standard library doesn’t have any such functionality.

The next sections will have a detailed look on the syntax, usage and performance of the library.

The Source Code Repository

The source code of the project as of 14th May,2015 is hosted at the following Github repository : https://github.com/aloygupta/simpleBigInt

At the time of submission of this report, the following functions are implemented which are declared in the bigint.h header file with comments describing what each of them does. The implementations of the functions are located in the bigint.c file.

The bigint.h header file is being given below. (It can also be viewed in the repository).

```
#ifndef _BIGINT_H_
#define _BIGINT_H_

#include<stdlib.h>

#define BISIZE 3000

struct bigint{
        char sign;                                          /* sign of the
biginteger */
        char value[BISIZE+1];          /* to store the value of the bigintger
of */
                                                                /* BISIZE number
of decimal digits */
};

/* a derived datatype for biginteger */

typedef struct bigint bigint;

/* Reads the sign and value of the biginteger in the form of
 * array of charcaters (decimal digits).In case of positive biginteger,giving
 * the '+' sign as input is optional. The function also adds '0's
 * to the rest of the char array. (It returns bigint *   )
 */

bigint * readBigInt(void);


/* Converts the string literal passed as argument into its
 * biginteger representation. If length of argument is BISIZE+1
 * it should include sign as first character, else size should be
 * <=BISIZE (and positive sign is implicitly assigned
 */
```

```
bigint *assignBigInt(char *n);



/* Prints the biginteger passed as argument.Prints the sign only if its
negative.
 * Returns EOF upon error
 */

int printBigInt(bigint *n);



/*  Print the biginteger passed as argument followed by a newline character.
 *      It returns the value returned by printBigInt(bigint *n)
 */

int printlnBigInt(bigint *n);



/*  Converts an integer to its biginteger representation.
 *      Returns a pointer to the newly allocated bigint
 */

bigint * toBigInt(int x);


/* Returns a pointer to a biginteger which represents the absolute value
 *  of another biginteger passed as argument
 */

bigint * absBigInt(bigint *n);



/* Performs addition of two bigintegers passed as argument.
 * Returns (bigint *)
 */

bigint * addBigInt(bigint * x, bigint *y);



/* Perform subtraction of two bigintegers m and n passed as arguments
 * Returns a pointer to the resultant biginteger
 */

bigint * subBigInt(bigint *m,bigint *n);


/* Multiplies two bigintegers passed as argument and returns a pointer
 * to the resultant bigint.The multiplication is carried out using
 * long multiplication method.
 */
```

```
bigint * multiplyBigInt(bigint *m,bigint *n);


/* Performs division of x by y using long division method and
 * returns the quotient as argument
 */

bigint * divideBigInt(bigint *x,bigint *y);


/* Returns a pointer to the remainder when x is divided by y.
 * Returns NULL when y is zero.
 */

bigint * remBigInt(bigint *x, bigint *y);



/* Comapres two biginteger variables m and n and returns an integer
 * indicating the comparison.
 * m>n then returns 1
 * m==n then returns 0
 * m<n then returns -1
 */

int compareToBigInt(bigint *m,bigint *n);




/* Returns the number of digits in the biginteger variable passed
 * as argument.Maximum returned value can be BISIZE.
 * When value passed is zero, returns 1 , as 1 zero digit
 * is atleast required to represent the number zero
 */

int lengthBigInt(bigint *n);


/* Returns a pointer to the resultant bigint m ^n ,
 * where m is a biginteger and n is an integer
 */

bigint *powBigInt(bigint *m,int n);



/* Returns a new bigint which is a copy of the one passed
 * as argument
 */

bigint *copyBigInt(bigint *n);



/* Reverses a bigint passed as argument in-place */
```

```
void reverseBigInt(bigint *n);



/* Appends a char(digit) to a bigint passed as argument.
 * Thus if n is value 12345 and '6' is passed then the bigint
 * is shifted one place left and '6' is appended.
 * The original bigint is changed in-place
 */

int appendDigitBigInt(bigint *n,char c);


/* Returns a pointer to the bigint representing Greatest Common Divisor (gcd)
 * of two bigintegrs passed as argument.
 */

bigint * gcdBigInt(bigint *m , bigint *n);



/* Returns a pointer to the bigint representing Least Common Multiple (lcm)
 * of two bigintegers passed as argument.
 */

bigint * lcmBigInt(bigint *m,bigint *n);




#endif
```

The bigint.c file which can be viewed at the repository consists of 19 functions. Not all of them are being shown in this report but a few functions and the algorithm working behind them will be explained in coming sections.

Syntax And Usage

The necessary statements to use the features of the library are pretty straightforward and easy to use and understand. The concepts behind the syntax are simple and yet can be used to compose powerful programs. Some of the basic usage and tips for the current version (in C) of the project is discussed here.

Declaration

A pointer to a bigint structure is to be declared using the form :

```
bigint * num;
```

It would be preferable to assign the pointers to NULL right after declaration to prevent misuse in later code. To allocate raw memory containing garbage value, the syntax would be:

```
num=(bigint *)malloc(sizeof(bigint));
```

The above declaration is absolutely NOT preferred because having garbage values initialized as that is never the desired purpose. A more preferred practice would be to initialize the variable (mostly to zero) right after declaration.

Initialization

To initialize a big integer to a particular value after declaration, there are two possible ways:

if the value fits into the conventional int datatype on that machine, the syntax to convert I and represent it in the bigint equivalent would be :

```
num=totoBigInt(value);
```

Where value is the ‘int’ sized integer. If value is 25 , then it would be

```
num=toBigInt(25);
```

The other, using assignBigInt(char *value) function

It accepts a string literal and converts it into the equivalent bigint representation.

```
num=assignBigInt("-848525805154849529529845111014524412859921");
```

Arithmetic

To add two numbers whose pointer variables are named x and y and the result is to pointed to by variable named z, the following is the code :

```
z=addBigInt(x,y);
```

Printing

To print a bigint, there are two versions, the second version adds a newline after printing but the first one doesn't.

```
printBigInt(x);
```

```
printlnBigInt(x);
```
// this version prints with newline

Destruction

bigint objects when no longer needed, should be freed manually using the free() function. Not doing so, will lead to memory leaks. It is also adviced to assign them to NULL immediately. The syntax to free is :

```
free(x);
```

Algorithms Used

There are 19 functions in the library. Some of he functions does mundane tasks like printing the bigint number, calculating HCF, LCM of two bigints making use of other library functions like multiplyBigInt(bigint *x, bigint *y) and addBigInt(bigint *x, bigint *y) which are of actual interest from an algorithmic view perspective.

The Code of the following functions are being discussed in this report. The **comments** along the code makes the algorithm clear and how it works

- bigint *assignBigInt(char *n) // converts string literal to bigint representaion
- bigint * addBigInt(bigint * x, bigint *y) // adds two bigints
- bigint * multiplyBigInt(bigint *m,bigint *n) // multiply two bigints
- bigint * divideBigInt(bigint *x,bigint *y) // divides two bigints

Code for converting string literal to bigint

```
/* Converts the string literal passed as argument into its
 * biginteger representation. If length of argument is BISIZE+1
 * it should include sign as first character, else size should be
 * <=BISIZE (and positive sign is implicitly assigned
 */

bigint *assignBigInt(char *n)
{
        int len=strlen(n);      /* length of string literal passed */
        int index;      /* array index variable */
        int zeroFlag=1;         /* flag to indicate whether string literal
                                        represents zero or not*/
        /*allocate memory for a biginteger */

        bigint *temp;


        if(len>(BISIZE+1))
        return NULL;

        if(len==(BISIZE+1))
        {
                if(n[0]!='+' && n[0]!='-')
                return NULL;
```

```
        }
        temp=(bigint *)malloc(sizeof(bigint));

        if(n[0]=='+' || n[0]=='-')
        temp->sign=n[0];
        else
        temp->sign='+';


        /* assign last variable of array of characters to nul */

        temp->value[BISIZE]='\0';

        index=BISIZE-1;

        len-=1;

        while(len>=1)
        {
                if((n[len]-'0')>0)
                zeroFlag=0;

                temp->value[index--]=n[len--];
                if(index<0)
                break;
        }


        if(n[len]!='+' && n[len]!='-')
        temp->value[index--]=n[len--];

        while(index>=0)
        temp->value[index--]='0';

        if(zeroFlag)
        temp->sign='+';  /* mathematically, zero is always given +ve sign */

        return temp;
}
```

Code for adding two bigints and returning the resultant bigint

```
/* Performs addition of two bigintegers passed as argument.
 * Returns (bigint *)
 */

bigint * addBigInt(bigint * x, bigint *y)
{
        int carry=0;    /* initially no carry */
        int digit;              /* variable to hold digit */
        int i;                  /* loop variable */

        /* variable to hold x+y */
```

```
        bigint *result=(bigint *)malloc(sizeof(bigint));

        /* assign nul character */
        result->value[BISIZE]='\0';

        if((x->sign=='+' && y->sign=='+') || (x->sign=='-' && y->sign=='-'))
        {
                /* 'if' block executes if both have same sign */

                result->sign=x->sign; /* assign sign */

                /* the loop to add two bigintegers */

                for(i=BISIZE-1;i>=0;i--)
                {
                /* calculate the result of addition of two digits and carry */
                        digit=carry + (x->value[i]-'0') + (y->value[i]-'0');

                if(digit>9)
                {
                carry=1;        /* set carry if digit>9 else reset */
                digit %=10;     /* digit is calculated modulo 10 if digit>9 */
                        }
                        else
                        carry=0;

                        /* assign digit as char in array */

                        result->value[i]='0'+digit;
                }
                /* overflow occurs when a last carry is generated */

                if(carry==1)
                        return NULL;    /* return NULL in case of overflow */
        }

        else if(x->sign=='+' && y->sign=='-')
        {
                y->sign='+';
                result=subBigInt(x,y);
                y->sign='-';
        }
        else
        {
                x->sign='+';
                result=subBigInt(y,x);
                x->sign='-';
        }

        return result; /* return pointer to the resultant bigint */
}
```

Code for multiplying two bigints and returning the resultant bigint

```
/* Multiplies two bigintegers passed as argument and returns a pointer
 * to the resultant bigint.The multiplication is carried out using
 * long multiplication method.
 */


bigint * multiplyBigInt(bigint *m,bigint *n)
{
        int i,j,k;
        int carry;
        int places=0;
        int p;

        /* length of multiplicand */
        int len1=lengthBigInt(m);

        /* length of multiplier */
        int len2=lengthBigInt(n);

        bigint *result;
        bigint *temp,*temp2;

        if(len1<len2)
        {
                result=multiplyBigInt(n,m);
                return result;
        }

        result=toBigInt(0);

        for(j=BISIZE-1;j>=(BISIZE-len2);j--)
        {
                temp=(bigint *)malloc(sizeof(bigint));
                temp->value[BISIZE]='\0';
                temp->sign='+';

                /* append required number of zeroes */
                for(i=BISIZE-1;i>(BISIZE-1-places);i--)
                {
                        temp->value[i]='0';
                }

                k=BISIZE-places-1;
                carry=0;

        /* carry out multiplication using long multiplication method */

                for(i=BISIZE-1;i>=(BISIZE-len1);i--)
                {
                        p=(m->value[i]-'0')*(n->value[j]-'0');
                        p+=carry;
                        temp->value[k--]='0'+(p%10);
                        carry=p/10;
                }
```

```
                temp->value[k]='0'+carry;

                /* fill the rest of the front digits with zero */
                while(k)
                {
                        temp->value[--k]='0';
                }

                /* get intermediate result */
                temp2=addBigInt(result,temp);

                free(result);
                result=temp2;
                free(temp);
                ++places;
        }

        /* assign appropriate sign */
        if(m->sign==n->sign)
        result->sign='+';
        else
        result->sign='-';

        return result;
}
```

Code for dividing two bigints and returning the resultant bigint

```
/* Performs division of x by y using long division method and
 * returns the quotient as argument
 */


bigint * divideBigInt(bigint *x,bigint *y)
{
        int len1,len2;
        int i;
        int in1,in2,in3=0;


        bigint *dividend,*quotient,*rem,*digit,*term1,*term2;

        bigint *ZERO=toBigInt(0);
        if(compareToBigInt(y,ZERO)==0)
        {
                free(ZERO);
                return NULL;
        }
        free(ZERO);
        len1=lengthBigInt(x);
        len2=lengthBigInt(y);
```

```
if(len1<len2)
return toBigInt(0);

rem=toBigInt(0);
quotient=toBigInt(0);

if(y->sign=='-')       /* check whether divisor is negative or not */
{
y->sign='+';    /* if negative, make it positive and set flag */
in3=1;
}

while(len1>0)
{
/* initially dividend is set to zero, as remainder is zero initially
* but for further iterations it gets the value of the remainder
         */
        dividend=copyBigInt(rem);
        free(rem);

/* bring down the new adjacent digit and append it to the dividend
 * (copied from remainder in previous step) to get the new dividend,
         * to perform division on
         */
        appendDigitBigInt(dividend,x->value[BISIZE-(len1--)]);

/* loop through 0 to 9 to check which is the right digit to get
         * appended to the quotient
         */
        for(i=0;i<=9;i++)
        {
                digit=toBigInt(i);
                term1=multiplyBigInt(y,digit);

                in1=compareToBigInt(dividend,term1);
                free(digit);

        /* check to see whether remainder is negative or positive
                 * and stop when there's transition form positive to
                 * zero or negative
                 */

                digit=toBigInt(i+1);
                term2=multiplyBigInt(y,digit);

                in2=compareToBigInt(dividend,term2);
                free(digit);

                if(in1==0)
                {
                        appendDigitBigInt(quotient,'0'+i);
                        rem=subBigInt(dividend,term1);
                        break;
```

```
                        }

                        if(in1==1)
                        {
                                if(in2==0)
                                {
                                        appendDigitBigInt(quotient,'0'+(i+1));
                                        rem=subBigInt(dividend,term2);
                                        break;
                                }

                                else if(in2==-1)
                                {
                                        appendDigitBigInt(quotient,'0'+i);
                                        rem=subBigInt(dividend,term1);
                                        break;
                                }
                                else
                                {
                                }

                        }

                        free(term1);
                        free(term2);
                }

        }

        if(in3==1)
        y->sign='-';    /* restore sign if required */

        /* assign proper sign to quotient */
        if(x->sign==y->sign)
        quotient->sign='+';
        else
        quotient->sign='-';

        return quotient;
}
```

Run Time Complexities

- assignBigInt(char *n) has complexity of Θ(length(n))
- addBigInt(bigint *x,bigint *y) has complexity of Θ(l) where l is length(x) or length(y) whichever is bigger
- multiplyBigInt(bigint *x,bigint *y) has complexity of Θ((n-len1)*(n-len2)) where len1 is the smaller of length(x) and length(y) while len2 is the bigger
- divideBigInt(bigint *x,bigint *y) has a complexity of Θ(len) where len is length(x)

Demo Programs

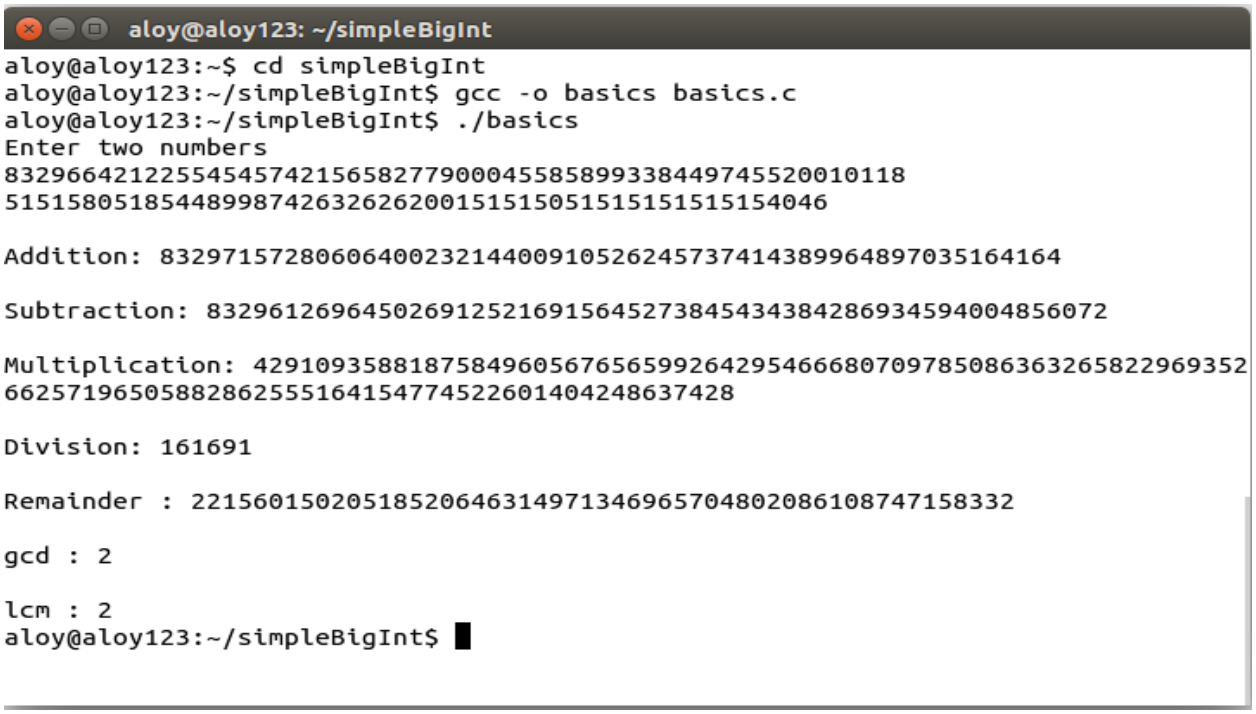
Three demo programs highlighting the usage of the library is being given in this report. Apart from these, the library can be used to solve a large variety of problems in the domain of cryptography, statistics, astronomy, mathematics, and any sort of number crunching activity.

Program to calculate addition, subtraction, multiplication, division, remainder, gcd, lcm of two large integers.

```
#include <stdio.h>
#include "bigint.h"

int main()
{
     bigint * x,*y,*z;
     printf("Enter two numbers\n");
     x=readBigInt();
     y=readBigInt();

     z=addBigInt(x,y);
     printf("\nAddition: ");
     printlnBigInt(z);

     free(z);

     z=subBigInt(x,y);
     printf("\nSubtraction: ");
     printlnBigInt(z);
     free(z);

     z=multiplyBigInt(x,y);
     printf("\nMultiplication: ");
     printlnBigInt(z);
     free(z);

     z=divideBigInt(x,y);
     printf("\nDivision: ");
```

```
	printlnBigInt(z);
	free(z);

	z=remBigInt(x,y);
	printf("\nRemainder : ");
	printlnBigInt(z);

	free(z);
	z=gcdBigInt(x,y);
	printf("\ngcd : ");
	printlnBigInt(z);

	z=gcdBigInt(x,y);
	printf("\nlcm : ");
	printlnBigInt(z);

	free(z);
	free(x);
	free(y);

	return 0;
}
```

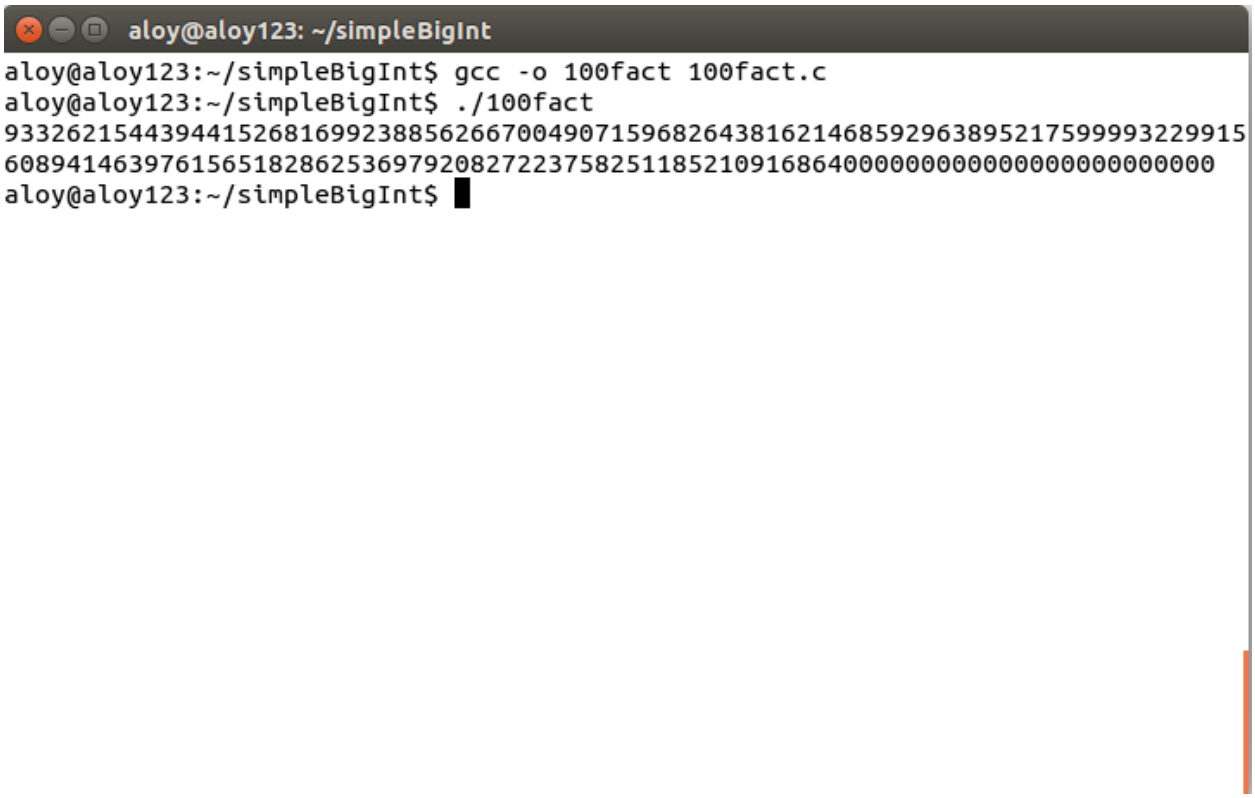
Screenshot of Output (Fig 1.)



Program to calculate 100 factorial

```
#include<stdio.h>
#include "bigint.h"

int main(){

bigint *fact, *x, *tmp;
int i;

fact=toBigInt(1);

for(i=1;i<=100;i++){
x=toBigInt(i);
tmp=multiplyBigInt(fact,x);
free(x);
free(fact);
fact=tmp;
}

printlnBigInt(fact);

free(fact);

return 0;
}
```

Screenshot of Output (Fig 2.)



Program to display distance between two galaxies in metres

```
/* The Sagittarius Dwarf Elliptical Galaxy is the next closest ,
at 662,000,000,000,000,000 km (70,000 light years) from the Sun.
*/


#include<stdio.h>

#include "bigint.h"

int main(){


bigint *distance, *CONVERT;

int d;


CONVERT=multiplyBigInt(toBigInt(300000000),toBigInt(31536000));
```

```
printf("Enter distance between two galaxies in light years\n");
scanf("%d",&d);
distance=multiplyBigInt(toBigInt(d),CONVERT);
printf("Distance between the galaxies in metres is : ");
printlnBigInt(distance);

free(distance);
free(CONVERT);

return 0;
}
```

Screenshot of Output (Fig 3.)

Scope For Improvement

The following points have been identified as areas that can be improved upon :

- Transform the project into C++ enabling use of operator overloading and representation as a class instead of struct
- There's scope for improving the division and power algorithms in terms of speed and efficiency.
- Extending the functionality to support floating point numbers as well. Currently, the library only deals with integers not numbers
- Use a larger BASE for representation instead of the decimal base so that representation takes up much lesser space.

Conclusion

Thus, the project fulfills the purpose of demonstrating how to deal with really large integers and perform arithmetic and logical operations on them. Of course, there are third party libraries which are more robust and more efficient than the simpleBigInt library that we have developed. There are lots of ways to improve the algorithms presented here. We have used long multiplication and long division algorithms to implement the multiplyBigInt and divideBigInt functions One of the key challenges the project successfully addresses is keeping the API simple and easy to use. As far as the algorithms used, they perform well under most circumstances.

Bibliography

[1] Introduction to Analytic Number Theory by Tom M. Apostol, 2, Published by Springer (2010)
[2] ALGORITHMS by Sanjoy Dasgupta, Published by McGraw Hill Education (India) Private Limited (2006)
[3] Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest.
[4] Online Reference :
http://faculty.cse.tamu.edu/djimenez/ut/utsa/cs3343/lecture20.html