Number Encoding and Unbounded Integers

Objective/Abstract:

The objective of this assignment is to convert an unbounded integer to its binary representation, particularly, it's 32-bit bit representation. After the conversion, the MSB and LSB must be stated.

Introduction:

In class, we learned how numeric systems are represented. Numeric systems depend on a base**,** otherwise known as the radix**.** The least significant bit, LSB, is the right-most bit, while the most significant bit, MSB, is the left-most non-zero bit. Each bit takes a power of 1 greater than the previous bit, beginning from the LSB. This forms a polynomial:

$$a1x^0 + a2x^1 + a3x^2 + \cdots aNx^n$$

Where a1 ... aN are the coefficients/elements of the numeric system and x is the radix, or base of the numeric system. for a binary system, this becomes:

$$\{1 \mid 0\}2^0 + \{1 \mid 0\}2^1 + \{1 \mid 0\}2^2 + \cdots \{1 \mid 0\}2^n$$

Therefore, the binary sequence: 1010 becomes:

$$1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 8 + 0 + 2 + 0 = 10\ base\ 10$$

To go backwards (from base 10 to base 2), we would apply a recursive division algorithm as such:

10/2 = 5 r 0; 5/2 = 4 r 1; 4/2 = 2 r 0; 2/2 = 1 r 0; 1/2 = 0 r 1

The remainders become the binary sequence. The LSB comes from the remainder of the first division (0 from 10/2 = 5 r 0), while the MSB comes from the remainder of the last division (1/2 = 0 r 1 in this case). Therefore, the decimal number 10 represents the binary pattern: 1010.

If we would like to apply this to another numeric system, then we would divide by the radix of the respective system and take the remainder of each computation in a similar to what we have just done to get the final value of the decimal representation in the desired numeric system.

Methods/Procedure:

For this homework, I have decided to continue working with C/C++. To allow the input of an integer of arbitrary length, I have created a very simple, and therefore quite inefficient, **BigInteger** class. Then, a recursive algorithm was applied to compute the binary representation of the input value.

Methods/Procedure:

Some example results attained from our algorithm using our custom BigInteger class are:

```
Please enter a whole number (any length): 9
Your number, 9, in binary is:
0000 0000 0000 0000 0000 0000 0000 1001
The function dec_to_binary32() took ~0.00 seconds --> ~0.00 minutes


Please enter a whole number (any length): 14
Your number, 14, in binary is:
0000 0000 0000 0000 0000 0000 0000 1110
The function dec_to_binary32() took ~0.00 seconds --> ~0.00 minutes


Please enter a whole number (any length): 15
Your number, 15, in binary is:
```

```
0000 0000 0000 0000 0000 0000 0000 1111
The function dec_to_binary32() took ~0.00 seconds --> ~0.00 minutes


Please enter a whole number (any length): 31
Your number, 31, in binary is:
0000 0000 0000 0000 0000 0000 0001 1111
The function dec_to_binary32() took ~0.00 seconds --> ~0.00 minutes


Please enter a whole number (any length): 32432
Your number, 32432, in binary is:
0000 0000 0000 0000 0111 1110 1011 0000
The function dec_to_binary32() took ~1.00 seconds --> ~0.02 minutes


Please enter a whole number (any length): 323452
Your number, 323452, in binary is:
0000 0000 0000 0100 1110 1111 0111 1100
The function dec_to_binary32() took ~2.00 seconds --> ~0.03 minutes


Please enter a whole number (any length): 454675
Your number, 454675, in binary is:
0000 0000 0000 0110 1111 0000 0001 0011
The function dec_to_binary32() took ~3.00 seconds --> ~0.05 minutes


Please enter a whole number (any length): 765200
Your number, 765200, in binary is:
0000 0000 0000 1011 1010 1101 0001 0000
The function dec_to_binary32() took ~6.00 seconds --> ~0.10 minutes
```

The codes for our implementation of this algorithm and "BigInteger" follows:

```
/*
 *     Author:            Haleeq Usman
 *     Dev. IDE:          Eclipse IDE for C/C++ Developers
 *                        Version: Indigo Service Release 2
 *                        Build id: 20120216-1857
 *     Dev. OS:           Windows 7 Home Premium SP1 (64-bitS)
 *     Dev. System:       Dell XPS 8300, 8GB RAM Intel(R) Core(TM)
 *                        i7-2600 CPU @ 3.40Ghz (quad core at 3.40Ghz)
 *
 *     This file contains our static/global variables/constants,
 *     data structures (enum, structs, classes, etc...), and
 *     our project headers.
 *
 */

#ifndef GLOBALS_H_ // process only if GLOBALS_H_ constant is not defined
#define GLOBALS_H_ // prevent multiple processes by defining constant.
// Our standard input/output header: printf, etc...
```

```
#include <stdio.h>
// Contains input/ouput functions in the 'std' namespace
// ex: cout, cin, etc...
#include <iostream>
// Contains manipulation functions in the 'std' namespace for formatting
// ex: setprecision, etc...
#include <iomanip>
// Contains cross-platform data types: int8. uint8, int16, etc...
#include <stdint.h>
// We use this to measure how effective our algorithms are.
#include <time.h>
// Used in our threaded algorithm.
#include <windows.h>
// We use this for our BigInteger class
#include <vector>
// Let's include our BigInteger header
#include "BigInteger.h"
// We use this to reverse our BigInteger vector
#include <algorithm>


/* Prototypes for our functions */
void dec_to_binary32(void);

#endif /* GLOBALS_H_ */




/*
 *     Author:             Haleeq Usman
 *     Dev. IDE:           Eclipse IDE for C/C++ Developers
 *                         Version: Indigo Service Release 2
 *                         Build id: 20120216-1857
 *     Dev. OS:            Windows 7 Home Premium SP1 (64-bitS)
 *     Dev. System:        Dell XPS 8300, 8GB RAM Intel(R) Core(TM)
 *                         i7-2600 CPU @ 3.40Ghz (quad core at 3.40Ghz)
 *
 *     This file contains the definition for our simple BigInteger class.
 *
 */

#ifndef BIGINTEGER_H_
#define BIGINTEGER_H_

#include <iostream>

class BigInteger {
// We friend these to avoid conflict with the global io stream overloads.
      friend std::ostream &operator<<(std::ostream &output, const BigInteger
                                      &big_int);
      friend std::istream &operator>>(std::istream &input, BigInteger &big_int);
public:
      // We only add few constructors for our BigInteger to keep
```

```
// things simple.
BigInteger(void);
BigInteger(unsigned int value);
BigInteger(char *value);
BigInteger(std::vector<char> &vect);

// When garbage collection is ready, we must
// deallocate the resources we are using.
// We make the deconstructor virtual in case of inheritance
// or deletion of the current object while utilizing polymorphism.
virtual ~BigInteger();

// This will return the numeric value as a string.
void toString(void);

// This will return the numeric value of our BigInteger.
unsigned int toUInteger(void);

/**
 * Returns 1 if the BigInteger is greater than 1.
 * Returns 0 otherwise.
 */
int isGTZero(void);

/**
 * Returns 1 if the two BigIntegers have equal values.
 */
int equalTo(const BigInteger &big_int);

/**
 * Returns size of vector in our BigInteger.
 */
size_t getLen(void);

// Let's overload a few operators to give the class
// the feel of an ordinary data type. We overload
// only a few operators to keep things simple.

// We take the second argument by a constant reference
// sometimes so that we do not utilize additional memory
// creating a second copy of the argument, but we make sure
// that the value cannot change (const). This makes it act
// like it was passed byVal, but without the resource
// hogging associated with byVal.


// Assignment operator overload definitions.
/**
 * Assigns an unsigned integer value to our BigInteger.
 */
BigInteger &operator=(unsigned int value);

/**
 * Assigns the value of one big integer to the other.
```

```
 */
BigInteger operator=(const BigInteger &value);

/**
 * Assigns a BigInteger value to our BigInteger (reference).
 */
BigInteger &operator=(BigInteger &value);

/**
 * Adds a BigInteger to our BigInteger object
 * and sets the result to our BigInteger.
 */
BigInteger &operator+=(const BigInteger &value);


// Addition operator overload definitions.
/**
 * Adds an unsigned integer value to our BigInteger object.
 */
BigInteger operator+(unsigned int value);

/**
 * Adds the string representation of an integer value
 * to our BigInteger object.
 */
BigInteger operator+(char *value);

/**
 * Adds the value of a BigInteger to our object.
 */
BigInteger operator+(const BigInteger &value);


// Subtraction operator overload definitions.
/**
 *  Subtracts an unsigned integer value from our BigInteger object.
 */
BigInteger operator-(unsigned int value);

/**
 * Subtracts the string representation of an integer value from
 * our BigInteger object.
 */
BigInteger operator-(char *value);

/**
 * Subtracts the value of a BigInteger from our object.
 */
BigInteger operator-(const BigInteger &value);

/**
 * Subtracts a BigInteger from our BigInteger object
 * and sets the result to our BigInteger.
 */
```

```
        BigInteger &operator-=(const BigInteger &value);


        // Division operator overload definition
        /**
         * Divides our BigInteger object by the unsigned integer value.
         */
        BigInteger operator/(unsigned int value);

        /**
         * Divides our BigInteger object by another BigInteger
         * and sets the result to our BigInteger.
         */
        BigInteger &operator/=(const BigInteger &value);

        /**
         * Divides our BigInteger object by another BigInteger.
         */
        BigInteger operator/(const BigInteger &big_int);

        /**
         * Modules our BigInteger object by an unsgined integer.
         */
        BigInteger operator%(unsigned int value);

private:
        // This will be the vector to hold our digits.
        std::vector<char> digits;
        // Lets store the iterator to our vector.
        std::vector<char>::const_iterator it;

        /**
         * retrieves the digits in the unsigned int 'value' and places
         * them into the character vector 'digits'.
         */
        void get_digits(std::vector<char> &vect, unsigned int value);

        /**
         * Adds the element of a vector to our BigInteger.
         */
        BigInteger addVectors(const std::vector<char> &val_digits);

        /**
         * Subtracts the element of a vector from our BigInteger.
         */
        BigInteger subtractVectors(const std::vector<char> &val_digits);

};
#endif /* BIGINTEGER_H_ */



/*
 *
```

```
 *     Author:             Haleeq Usman
 *     Dev. IDE:           Eclipse IDE for C/C++ Developers
 *                         Version: Indigo Service Release 2
 *                         Build id: 20120216-1857
 *     Dev. OS:            Windows 7 Home Premium SP1 (64-bitS)
 *     Dev. System:        Dell XPS 8300, 8GB RAM Intel(R) Core(TM)
 *                         i7-2600 CPU @ 3.40Ghz (quad core at 3.40Ghz)
 *
 *     This file contains the implementation for our simple BigInteger class.
 *
 */

#include "headers/globals.h"

BigInteger::BigInteger() {
	// Nothing to do for now
}

BigInteger::BigInteger(unsigned int value) {
	// We allocate memory for our int_stacks placeholder
	// using each digit of the unsigned integer value.
	get_digits(digits, value);

}

BigInteger::BigInteger(char *value) {
	// We allocate memory for our int_stacks placeholder
	// using the elements of the character array.
	int len, i;
	len = strlen(value);

	for (i = 0; i < len; ++i)
		digits.push_back(value[i]);
}

BigInteger::BigInteger(std::vector<char> &vect) {
	digits = vect;
}

void BigInteger::get_digits(std::vector<char> &vect, unsigned int value) {
	if (value > 9) {
		get_digits(vect, value / 10);
	}
	// Another way I can experiment with for performance boost
	// is to cast the int into a char using 'OR' for its first
	// 8 bits. I am not sure how static_cast works. So its worth
	// the experimentation later. For simplicity, we do this for now.
	vect.push_back(static_cast<char>('0' + (value % 10)));
}

int BigInteger::isGTZero() {
	int return_val;
	return_val = 0;
	if (digits.size() > 1) {
```

```
		for (it = digits.begin(); it != digits.end(); ++it) {
			if (*it != '0') {
				return_val = 1;
				break;
			}
		}
	} else {
		if (digits.at(0) != '0')
			return_val = 1;
	}
	return return_val;
}

size_t BigInteger::getLen() {
	return digits.size();
}

int BigInteger::equalTo(const BigInteger &big_int) {
	size_t offset, size;
	int return_val, i;
	offset = 0;
	if (big_int.digits.size() < digits.size()) {
		size = big_int.digits.size();
		for (offset = 0; offset < (digits.size() - big_int.digits.size());
				++offset) {
			if (digits[offset] != '0') {
				return 0;
			}
		}
	} else if (big_int.digits.size() > digits.size()) {
		size = digits.size();
		for (offset = 0; offset < (big_int.digits.size() - digits.size());
				++offset) {
			if (big_int.digits[offset] != '0')
				return 0;
		}
	} else {
		size = big_int.digits.size();
	}

	return_val = 1;
	for (i = 0; i<size; ++i) {
		if (big_int.digits.size() < digits.size()) {
			if (digits[i + offset] != big_int.digits[i]) {
				return_val = 0;
				break;
			}
		} else {
			if (digits[i] != big_int.digits[i + offset]) {
				return_val = 0;
				break;
			}
		}
	}
```

```
	return return_val;
}

void BigInteger::toString() {
	for (it = digits.begin(); it != digits.end(); ++it)
		std::cout << *it;
}

unsigned int BigInteger::toUInteger() {
	return (unsigned int) atoi(digits.data());
}

std::ostream &operator<<(std::ostream &output, const BigInteger &big_int) {
	std::vector<char>::const_iterator it;
	for (it = big_int.digits.begin(); it != big_int.digits.end(); ++it)
		output << *it;
	return output;
}

std::istream &operator>>(std::istream &input, BigInteger &big_int) {
	char c;
	while ((c != '\n') && (c != '\r\n')) {
		input.get(c);
		if ((c != '\n') && (c != '\r\n')) {
			big_int.digits.push_back(c);
		}
	}
	return input;
}

BigInteger BigInteger::addVectors(const std::vector<char> &val_digits) {
	std::vector<char> tmp_digits;
	size_t val_size, digits_size, i;
	int tmp_val, remainder, carry;
	val_size = val_digits.size();
	digits_size = digits.size();

	// We transverse in the opposite direction and apply
	// Ordinary base 10 addition.
	carry = 0;
	tmp_val = 0;
	i = 1;
	if (val_size < digits.size()) {
		for (it = digits.end() - 1; it != digits.begin() - 1; --it) {
			tmp_val = val_size >= i ? val_digits.at(val_size - i) - '0' : 0;
			tmp_val += *it - '0' + carry;
			remainder = tmp_val % 10;
			carry = 0;
			if (tmp_val > 9)
				carry = 1;
			tmp_digits.push_back(static_cast<char>('0' + remainder));
			i++;
		}
```

```
	} else {
		for (it = val_digits.end() - 1; it != val_digits.begin() - 1; --it) {
			tmp_val = digits_size >= i ? digits.at(digits_size - i) -'0' : 0;
			tmp_val += *it - '0' + carry;
			remainder = tmp_val % 10;
			carry = 0;
			if (tmp_val > 9)
				carry = 1;
			tmp_digits.push_back(static_cast<char>('0' + remainder));
			i++;
		}
	}

	if (carry > 0)
		tmp_digits.push_back(static_cast<char>('0' + carry));

	reverse(tmp_digits.begin(), tmp_digits.end());
	BigInteger return_val(tmp_digits);
	return return_val;
}

BigInteger BigInteger::subtractVectors(const std::vector<char> &val_digits) {
	std::vector<char> tmp_digits;
	size_t val_size, digits_size, i;
	int tmp_val, tmp_digit, carried;

	val_size = val_digits.size();
	digits_size = digits.size();

	// We transverse in the opposite direction and apply
	// Ordinary base 10 addition.
	carried = 0;
	tmp_val = 0;
	tmp_digit = 0;
	i = 1;
	if (val_size <= digits.size()) {
		for (it = digits.end() - 1; it != digits.begin() - 1; --it) {
			tmp_val = val_size >= i ? val_digits.at(val_size - i) - '0' : 0;
			tmp_digit = (*it - '0') - carried;
			if (tmp_digit < tmp_val) {
				carried = 1;
				tmp_val = tmp_digit + 10 - tmp_val;
			} else {
				carried = 0;
				tmp_val = tmp_digit - tmp_val;
			}
			tmp_digits.push_back(static_cast<char>('0' + tmp_val));
			i++;
		}
	} else {
		tmp_digits.push_back('0');
	}

	if (carried == 1) {
```

```
			tmp_digits.clear();
			tmp_digits.push_back('0');
		}

		reverse(tmp_digits.begin(), tmp_digits.end());
		BigInteger return_val(tmp_digits);

		return return_val;
}

BigInteger &BigInteger::operator=(unsigned int value) {
		digits.erase(digits.begin(), digits.end());
		get_digits(digits, value);
		return *this;
}

BigInteger &BigInteger::operator=(BigInteger &value) {
		digits = value.digits;
		return *this;
}

/**
 * Implementation for our unsigned integer addition
 */
BigInteger &BigInteger::operator+=(const BigInteger &value) {
		this->digits = operator +(value).digits;
		return *this;
}

BigInteger BigInteger::operator+(unsigned int value) {
		std::vector<char> val_digits;
		get_digits(val_digits, value);
		return addVectors(val_digits);
}

BigInteger BigInteger::operator+(const BigInteger &big_int) {
		BigInteger return_val(addVectors(big_int.digits));
		return return_val;
}

BigInteger BigInteger::operator+(char *value) {
		std::vector<char> val_digits, tmp_digits;
		size_t val_size, digits_size, i;
		int tmp_val, remainder, carry, dummy;
		val_size = strlen(value);
		digits_size = digits.size();

		// We transverse in the opposite direction and apply
		// Ordinary base 10 addition.
		carry = 0;
		tmp_val = 0;
		i = 1;
		if (val_size < digits.size()) {
			for (it = digits.end() - 1; it != digits.begin() - 1; --it) {
```

```
                        tmp_val = val_size >= i ? value[val_size - i] - '0' : 0;
                        tmp_val += *it - '0' + carry;
                        remainder = tmp_val % 10;
                        carry = 0;
                        if (tmp_val > 9)
                                carry = 1;
                        tmp_digits.push_back(static_cast<char>('0' + remainder));
                        i++;
                }
        } else {
                val_size--;
                for (dummy = val_size; dummy >= 0; --dummy) {
                        tmp_val = digits_size >= i ? digits.at(digits_size - i)- '0' : 0;
                        tmp_val += value[dummy] - '0' + carry;
                        remainder = tmp_val % 10;
                        carry = 0;
                        if (tmp_val > 9)
                                carry = 1;
                        tmp_digits.push_back(static_cast<char>('0' + remainder));
                        i++;
                }
        }

        if (carry > 0)
                tmp_digits.push_back(static_cast<char>('0' + carry));

        reverse(tmp_digits.begin(), tmp_digits.end());
        BigInteger return_val(tmp_digits);

        return return_val;
}

/**
 * Implementation for our unsigned integer subtraction
 */
BigInteger &BigInteger::operator-=(const BigInteger &value) {
        this->digits = operator -(value).digits;
        return *this;
}

BigInteger BigInteger::operator-(unsigned int value) {
        std::vector<char> val_digits;
        get_digits(val_digits, value);
        return subtractVectors(val_digits);
}

BigInteger BigInteger::operator-(const BigInteger &big_int) {
        BigInteger return_val(subtractVectors(big_int.digits));
        return return_val;
}

BigInteger BigInteger::operator-(char *value) {
        std::vector<char> val_digits, tmp_digits;
        size_t val_size, digits_size, i;
```

```
	int tmp_val, tmp_digit, carried, dummy;
	val_size = strlen(value);
	digits_size = digits.size();

	// We transverse in the opposite direction and apply
	// Ordinary base 10 addition.
	carried = 0;
	tmp_val = 0;
	tmp_digit = 0;
	i = 1;
	if (val_size <= digits.size()) {
		for (it = digits.end() - 1; it != digits.begin() - 1; --it) {
			tmp_val = val_size >= i ? value[val_size - i] - '0' : 0;
			tmp_digit = (*it - '0') - carried;
			if (tmp_digit < tmp_val) {
				carried = 1;
				tmp_val = tmp_digit + 10 - tmp_val;
			} else {
				carried = 0;
				tmp_val = tmp_digit - tmp_val;
			}
			tmp_digits.push_back(static_cast<char>('0' + tmp_val));
			i++;
		}
	} else {
		tmp_digits.push_back('0');
	}

	if (carried == 1) {
		tmp_digits.clear();
		tmp_digits.push_back('0');
	}

	reverse(tmp_digits.begin(), tmp_digits.end());
	BigInteger return_val(tmp_digits);

	return return_val;
}

/**
 * Implementation for our unsigned integer division
 */

BigInteger &BigInteger::operator/=(const BigInteger &value) {
	this->digits = operator /(value).digits;
	return *this;
}

BigInteger BigInteger::operator/(unsigned int value) {
	BigInteger tmp_val(digits), return_val;

	if (tmp_val.toUInteger() == value) {
		return_val = 1;
		return return_val;
```

```
	}

	return_val = 0;
	while (1) {
		tmp_val -= value;
		if (tmp_val.toUInteger() == value)
			return_val += 1;
		if (tmp_val.isGTZero())
			return_val += 1;
		else
			break;
	}

	return return_val;
}

BigInteger BigInteger::operator/(const BigInteger &big_int) {
	BigInteger tmp_val(digits), return_val;

	if (tmp_val.equalTo(big_int)) {
		return_val = 1;
		return return_val;
	}

	return_val = 0;
	while (1) {
		if (tmp_val.equalTo(big_int)) {
			return_val += 1;
		}
		tmp_val -= big_int;
		if (tmp_val.isGTZero())
			return_val += 1;
		else
			break;
	}

	return return_val;
}

BigInteger BigInteger::operator%(unsigned int value) {
	BigInteger tmp_val(digits), remainder_val;
	while (1) {
		if (tmp_val.toUInteger() == value) {
			remainder_val = 0;
		} else {
			remainder_val = tmp_val;
		}
		tmp_val -= value;
		if (!tmp_val.isGTZero())
			break;
	}
	return remainder_val;
}
```

```
BigInteger::~BigInteger() {
	// TODO destructor stub
}


/*
 *	Author:			Haleeq Usman
 *	Dev. IDE:		Eclipse IDE for C/C++ Developers
 *					Version: Indigo Service Release 2
 *					Build id: 20120216-1857
 *	Dev. OS:		Windows 7 Home Premium SP1 (64-bitS)
 *	Dev. System:	Dell XPS 8300, 8GB RAM Intel(R) Core(TM)
 *					i7-2600 CPU @ 3.40Ghz (quad core at 3.40Ghz)
 *
 *	This file contains the implemenation for the decimal to binary
 *	converter.
 *
 */

#include "headers/globals.h"

void dec_to_binary32() {
	time_t start_time;
	double elapsed_time;
	BigInteger input;
	std::vector<char> output;
	std::vector<char>::const_iterator it;
	int i;

	std::cout << "Please enter a whole number (any length): ";
	std::cin >> input;

	std::cout << "Your number, " << input << ", in binary is:\n";

	start_time = time(NULL);

	while (1) {
		if((input % 2).isGTZero())
			output.push_back('1');
		else
			output.push_back('0');
		input /= 2;
		if(!input.isGTZero()) {
			break;
		}
	}

	size_t num_bits = 32;
	if(output.size() < num_bits) {
		num_bits -= output.size();
		for(i=0; i<num_bits; ++i) {
			output.push_back('0');
		}
	}
```

```
	reverse(output.begin(), output.end());
	i = 0;
	for(it = output.begin(); it != output.end(); ++it) {
		++i;
		std::cout << *it;
		if(i % 4 == 0)
			printf(" ");
	}
	printf("\n");

	// Let's measure how long it took to hit the end of the function
	elapsed_time = difftime(time(NULL), start_time);
	printf(
			"The function sum_ints_fast() took ~%.2lf seconds --> ~%.2lf
			minutes\n",
			elapsed_time, elapsed_time / 60);
}
/*
 *	Author:			Haleeq Usman
 *	Dev. IDE:		Eclipse IDE for C/C++ Developers
 *				Version: Indigo Service Release 2
 *				Build id: 20120216-1857
 *	Dev. OS:		Windows 7 Home Premium SP1 (64-bitS)
 *	Dev. System:		Dell XPS 8300, 8GB RAM Intel(R) Core(TM)
 *				i7-2600 CPU @ 3.40Ghz (quad core at 3.40Ghz)
 *
 *	This file contains the main executable/start point of program.
 *
 */

#include "source/headers/globals.h"

int main() {
	dec_to_binary32();
	return 0;
}
```

Conclusion

We notice that our conversion algorithm is accurate. However, it takes a heavy toll as the integer value increases. We noticed that the time elapsed to formulate the conversion was about 6 seconds. This is very inefficient! The inefficiency comes from the very simple implementation of our custom data type, our custom **BigInteger** class. To account for the time complexity, we would need to consider every

recursive pass. However, due to the time constraint and limited nature of the assignment, we have chosen to only display the time elapsed.