Lab 1 and 2 – Operating with Huge Integers

COMPENG 2SI3 – Data Structures and Algorithms

Samarth Patel
pates199
400377664

Ratnasingham Tharmarasa

Submitted: March 12th, 2023

Description of the Data Structure and Algorithms

The HugeInteger class represents large integers that exceed the range of primitive integers in C++. This means the integers being operated on are beyond the range of -2^{63} to $2^{63} - 1$.

Each digit of the integer is kept in its proper location value by the class HugeInteger() using a vector data structure. The class also contains private variables; one of which represents the sign of the integer and another to store the HugeInteger's size.

The class has two constructors:

1) HugeInteger::HugeInteger(const std::string& val) – This constructor creates a HugeInteger in the form of an integer and stores it in a vector. The string can optionally begin with a negative sign, representing a negative value. The constructor will then assign the 'bool isNegative" value a 1 or a 0, depending on if the sign is negative (1) or not (0). Furthermore, the function will throw an exception of the string is empty. Next, the constructor iterates through each character to verify if it is a number. If so, 48 will be subtracted from the ASCII value and the value will be pushed into the vector.

2) HugeInteger::HugeInteger(int n) – this constructor uses input n to construct a random HugeInteger of size n. Note that the constructor ensures the first number is not 0. The number is then pushed to the vector.

Furthermore, to following methods were created to preform arithmetic operations between two HugeIntegers. A new HugeInteger instance is returned as the output.

HugeInteger HugeInteger::add(const HugeInteger& h)

The program performs the addition operation by adding the least significant bits of each digit from both vectors, together with any carry, and then determining if the result is more than 10. The software increments a string with the sum's value minus 10 and sets the carry to 1 if the total exceeds 10. If the total is less than or equal to 10, the program turns the value into a string, adds it to the string, and sets the carry to 0. The index is decremented for each sum that is appended to the string. The function also verifies that neither vector's index is less than 0 before continuing.

HugeInteger HugeInteger::subtract(const HugeInteger& h) –

This function subtracts two large integers and returns the result as a new HugeInteger object. It first initializes an empty string to hold the final result and a temporary HugeInteger object to hold the value of the second object. Then, it checks for special cases where the two input HugeIntegers have different signs or one of them has a value of zero. If the first HugeInteger is negative and the second is positive, it calls the add function and sets the sign of the result to negative. If the second HugeInteger is negative and the first is positive, it calls the add function and sets the sign of the result to positive.

The function then creates two new HugeInteger objects, one for each input HugeInteger, and begins subtracting their digits from right to left. It compares the size of the two HugeInteger objects and uses the larger one as the first object to subtract from. If the result of the subtraction is negative, it borrows a digit from the next left digit and adds 10 to the current digit to correct the subtraction. The function then builds a string representation of the result by concatenating each digit from left to right.

Finally, the function checks the signs of the original HugeInteger objects to determine the sign of the result. If the second object is larger than the first, it sets the sign of the result to negative. If the first object is larger than the second, it sets the sign of the result to positive. The function then returns the result as a new HugeInteger object.

HugeInteger HugeInteger::multiply(const HugeInteger& h)

This function multiplies two HugeIntegers. It first creates a temporary vector with a size of (size1 + size2), where size1 and size2 are the sizes of the two HugeIntegers to be multiplied and initializes each element of the vector to 0. The result is then stored in the appropriate element of the temporary vector after multiplying each element of the first HugeInteger by each element of the second HugeInteger. Any carry-over digits are kept in the vector's following element. It then eliminates any leading zeroes from the temporary vector, multiplies each element of the first HugeInteger by each element of the second HugeInteger, and puts the resulting digits in a string variable called "product" in the correct sequence. Before providing the result as a new HugeInteger, it adds a "-" sign in front of the product string if one of the HugeIntegers is negative.

The function uses two nested loops to multiply each element of the two HugeIntegers. The outer loop iterates through each element of the first HugeInteger from right to left, while the inner loop iterates through each element of the second HugeInteger from right to left. The least significant digit of the result is stored in the appropriate element of the temporary vector after each element's product is determined by executing a simple multiplication of the two components with any carry-over digits from the previous multiplication. It stores the most significant digit of the result as a carry-over value to be added to the next multiplication. If the multiplication of the last two elements results in a carry-over digit, it adds that digit to the next element of the temporary vector.

int HugeInteger::compareTo(const HugeInteger& h)

This function compares two objects of the class HugeInteger. The function takes an object of the HugeInteger class as an argument and returns an integer value, which represents the result of the comparison. The integer value can be 0, 1, or -1, depending on the relationship between the two objects being compared.

The function first checks if both objects are equal to zero. If they are, the function returns 0. If one object is negative and the other is positive, the function returns -1 if the first object is less than the second, and 1 if the first object is greater than the second.

If both objects are positive, the function compares their sizes. If the first object is larger than the second, the function returns 1, and if the second object is larger than the first, the function returns -1. If both integers have the same length, it compares their digits from left to right. If it finds a digit in the current integer that is less than the corresponding digit in the argument integer, it returns -1. If it finds a digit in the current integer that is greater than the corresponding digit in the argument integer, it returns 1. If it reaches the end of the loop without finding any differences, it returns 0.

If both objects are negative, the function follows the same procedure as above but with the opposite sign, so if the first object is greater than the second, the function returns -1, and if the first object is less than the second, the function returns 1. Finally, if none of the above conditions are met, the function returns 0.

Theoretical Analysis of Running Time and Memory Requirement:

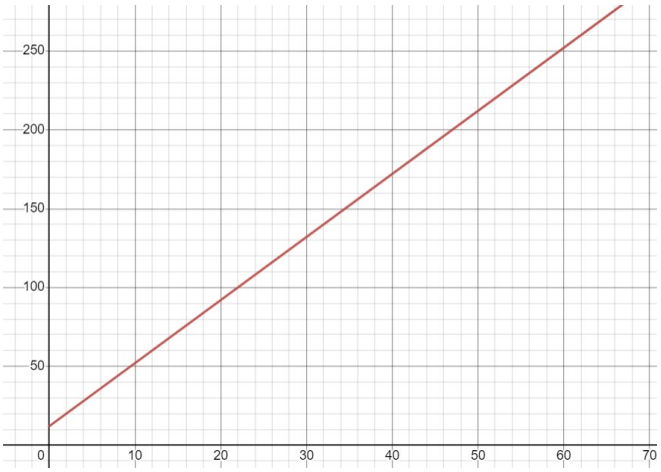
Memory:

The variables in the HugeInteger Class are:

```
std::vector<int>intList;
std::vector<int>resultList;
bool isNegative;
bool call;
int sign = 0;
```

There are two Boolean variables which take 1 byte each. The integer integers will take 2 bytes of memory. Vector intList is initialized as <int>, so it will take 2 bytes of memory per integer in the vector for a total of 2n. This is the same for vector resultList. Furthermore, a pointer to the first element of both vectors will take 4 bytes each.

As a result, HugeInteger is S(n) = 4n + 12.



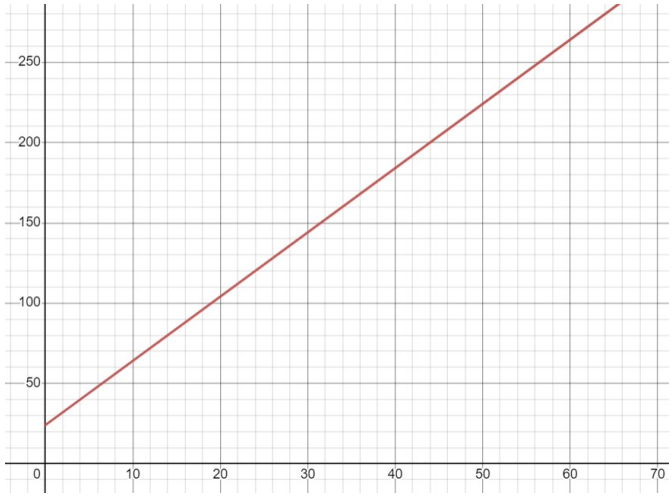
Note that S(n), in bytes, is on the y-axis and n, for integers, is on the x-axis.

Add:

The worst time for add method is Θ(n) as the function goes through each digit. The time required to execute the function increases with the magnitude of the integer. Since the add operation would still need to go through each digit in the HugeInteger, the average case execution time would likewise be n.

There are 6 integer variables in the function, which take a memory of 4 bytes each. There are also two temporary integer vectors, which take a memory of 2n each.

As a result, the total memory needed for this function is S(n) = 4n + 24.



Note that S(n), in bytes, is on the y-axis and n, for integers, is on the x-axis.

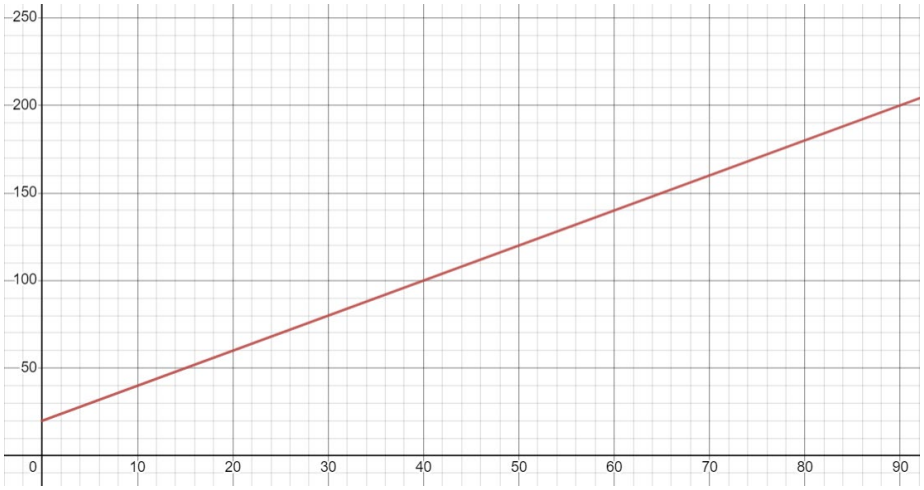
Subtract:

The worst run time for subtract method is Θ(n). This is because the function needs to loop though each digit to perform subtract.

This function has a string whose memory depends on its length. finalString. The maximum size of the string would be the sum of the lengths of the two input HugeInteger objects. Assuming each digit requires 1 byte of memory, and both HugeInteger objects have n digits each, the maximum size of finalString would be 2n bytes.

There are also 3 variables, each taking a memory of 4 bytes. Furthermore, there is a pointer variable which requires 8 bytes of memory in a 64-bit system.

In total, S(n) = 2n + 20

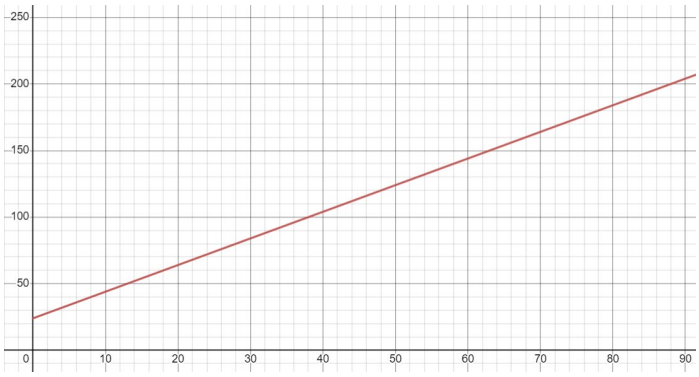


Note that S(n), in bytes, is on the y-axis and n, for integers, is on the x-axis.

Multiply:

The run time of this function is $\Theta(n^2)$. This is because there are 2 nested loops. In the worst case, if both of the sizes are equal, the complexity would be n^2.

There are 6 integer type variables used, each with a memory of 4 bytes. There is also a vector of integers, requiring 2n bytes of memory. As a result, S(n) = 2n + 24.



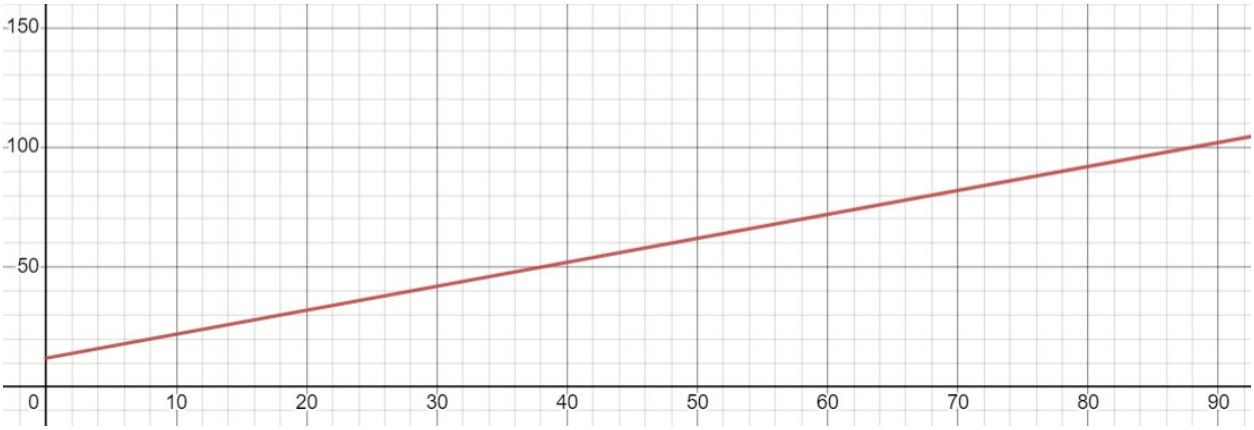
Note that S(n), in bytes, is on the y-axis and n, for integers, is on the x-axis.

Compare:

This function has 3 integer variables, which take 4 bytes of memory each.

However, this function's run time is $\Theta(n)$. This is due to the case of when both HugeIntegers are the same size, where in that case the function would have to loop through both integers to find the larger of the two.

Therefore, the total complexity is O(n) = n + 12.



Note that S(n), in bytes, is on the y-axis and n, for integers, is on the x-axis.

Test Procedure

String Input:

- Valid string (“3481”)
- Invalid beginning (“a48023”)
- Invalid middle (“3948g394”)
- Invalid end (“192834j”)
- Empty String (“ ”)
- Leading Zero (“019283”)

Random HugeInteger:

- N > 0
- N = 0
- N < 0

Addition:

- Size of HugeInteger > h HugeInteger
- Size of HugeInteger < h HugeInteger
- Size of HugeInteger = h HugeInteger
- Positive HugeInteger and positive h HugeInteger
- Positive HugeInteger and negative h HugeInteger
- Negative HugeInteger and negative h HugeInteger
- HugeInteger + h HugeInteger with a carry
- HugeInteger = h HugeInteger
- HugeInteger + h HugeInteger without a carry
- HugeInteger = h HugeInteger = 0

Subtraction:

- Size of HugeInteger > h HugeInteger
- Size of HugeInteger < h HugeInteger
- Size of HugeInteger = h HugeInteger
- Positive HugeInteger and positive h HugeInteger
- Positive HugeInteger and negative h HugeInteger
- Negative HugeInteger and negative h HugeInteger
- HugeInteger + h HugeInteger with a carry
- HugeInteger = h HugeInteger
- HugeInteger + h HugeInteger without a carry
- HugeInteger = h HugeInteger = 0
- HugeInteger = 0 and h HugeInteger != 0
- HugeInteger != 0 and h HugeInteger = 0

Multiplication:

- Size of HugeInteger > h HugeInteger
- Size of HugeInteger < h HugeInteger
- Size of HugeInteger = h HugeInteger
- Positive HugeInteger and positive h HugeInteger
- Positive HugeInteger and negative h HugeInteger
- Negative HugeInteger and negative h HugeInteger
- HugeInteger + h HugeInteger with a carry
- HugeInteger = h HugeInteger
- HugeInteger + h HugeInteger without a carry
- HugeInteger = h HugeInteger = 0
- HugeInteger = 0 and h HugeInteger != 0
- HugeInteger != 0 and h HugeInteger = 0

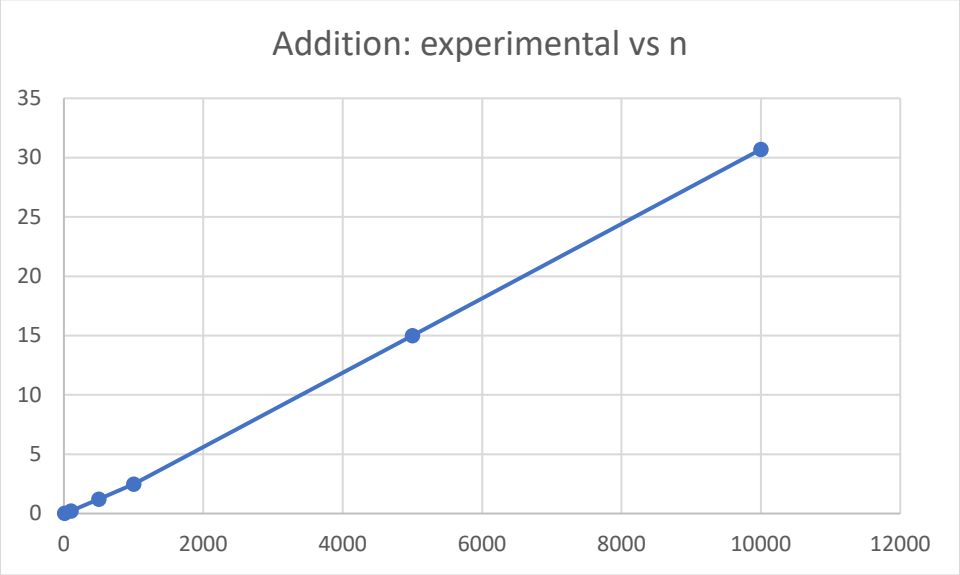
Comparison:

- HugeInteger size > h HugeInteger size
- HugeInteger size < h HugeInteger size
- HugeInteger size = h HugeInteger size
- HugeInteger > 0 AND h HugeInteger > 0
- HugeInteger < 0 AND h HugeInteger < 0
- HugeInteger > 0 AND h HugeInteger < 0
- HugeInteger < 0 AND h HugeInteger > 0

Run Time Analysis

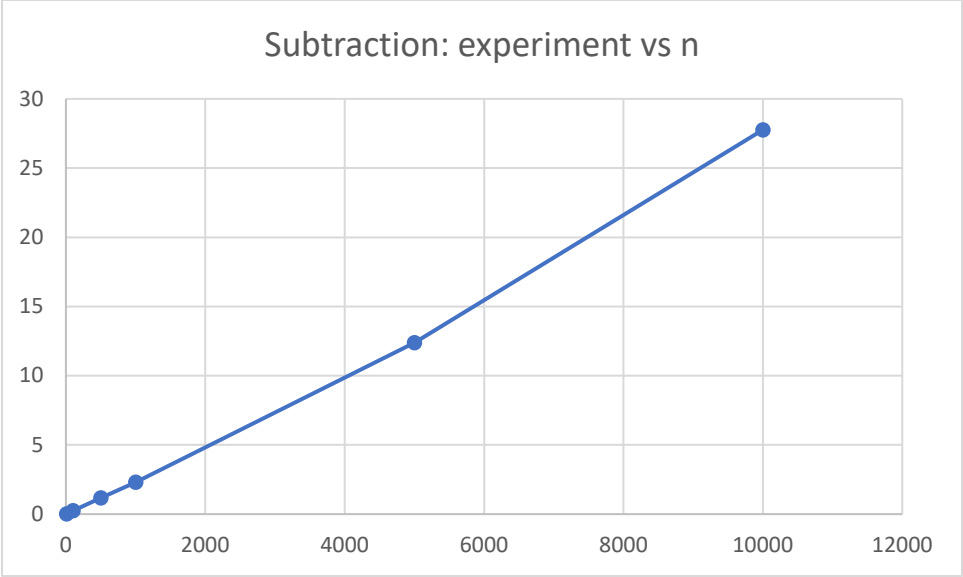
Addition

n	Experiment (ms)	BOOST	Theoretical
10	0.0239382	0.00021349	Θ(n)
100	0.223115	0.0004095	Θ(n)
500	1.203961	0.00025417	Θ(n)
1000	2.49385	0.00026133	Θ(n)
5000	15.01089	0.000230192	Θ(n)
10000	30.6892	0.00026018	Θ(n)



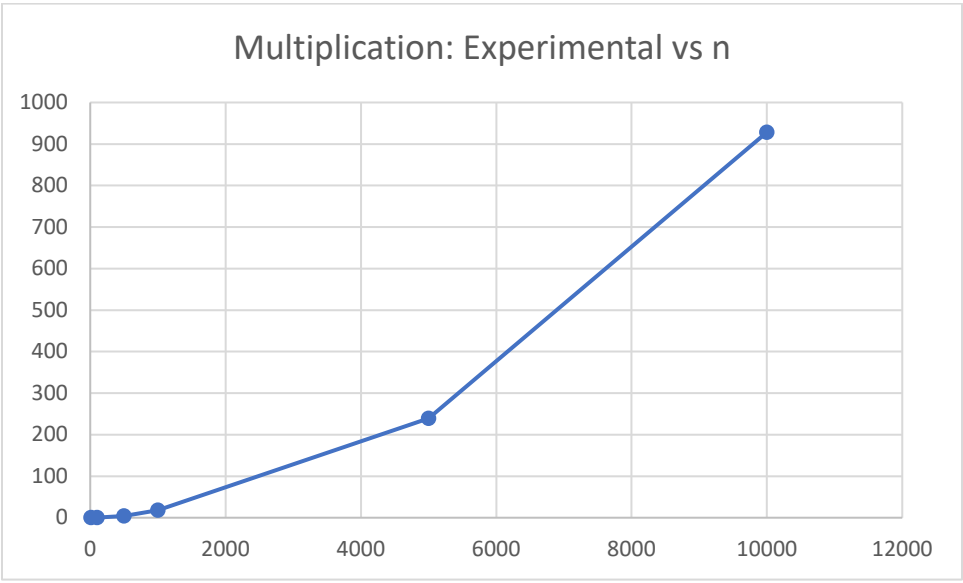
Subtraction

n	Experiment (ms)	BOOST	Theoretical
10	0.0260238	0.00006853	Θ(n)
100	0.239234	0.00005918	Θ(n)
500	1.16293	0.00005965	Θ(n)
1000	2.30192	0.00005938	Θ(n)
5000	12.3918	0.00005642	Θ(n)
10000	27.7639	0.0005407	Θ(n)



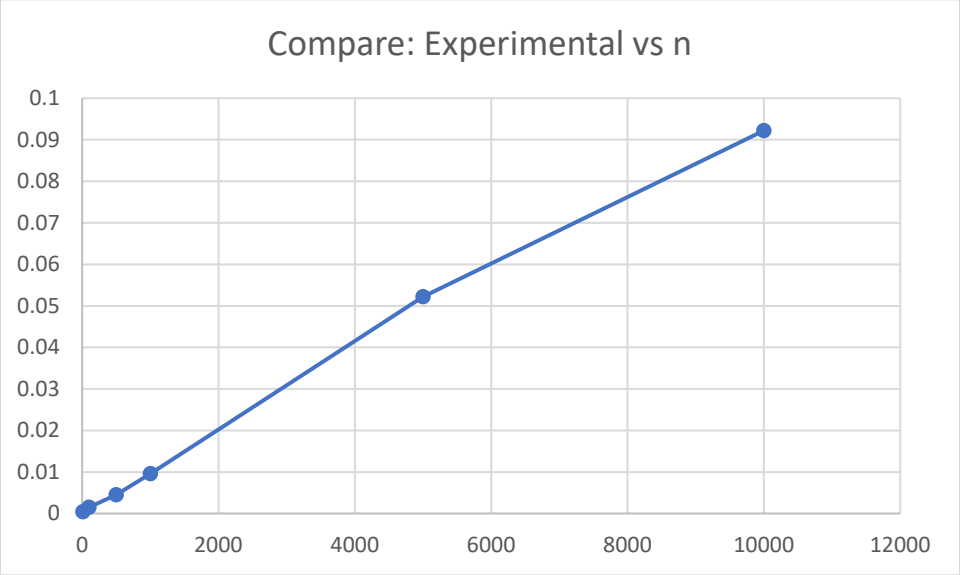
Multiply:

n	Experiment (ms)	BOOST	Theoretical
10	0.0429348	0.00012938	Θ(n²)
100	0.301823	0.001793849	Θ(n²)
500	4.11293485	0.028203	Θ(n²)
1000	18.30183	0.070282	Θ(n²)
5000	239.29384	0.928351	Θ(n²)
10000	928.38473	3.029348	Θ(n²)



Compare:

n	Experiment (ms)	BOOST	Theoretical
10	0.00041283	0.00004712	Θ(n)
100	0.001512	0.00004001	Θ(n)
500	0.004487	0.00003148	Θ(n)
1000	0.009611	0.00005432	Θ(n)
5000	0.05218	0.00003016	Θ(n)
10000	0.092218	0.00004269	Θ(n)



Discussion of Results and Comparison

It was discovered that the theoretically predicted values for the addition and subtraction functions were comparable to the actual measured runtime values. The upward trend in both plots, with just a small amount of divergence, indicates that the running time increases with n. The runtime of the comparison function similarly increases linearly with input size in a similar manner. The graphs for addition, subtraction, and comparison all show an upward trend overall, with some slight deviations due to outliers and measurement errors.

It was discovered that the multiplication graph was rising clearly at an exponential pace, which varied the theoretical running time was n^2. The difference that may be the result of data outliers or mathematical errors in the boost run-time.

Improvements that could have been made include how many of variables in the code were of type integer, which ultimately used up more memory. Instead, these variables could have been removed entirely or changed to Boolean types as they consume less memory. The multiplication method could have also implemented the Kasturba multiplication method, which allows multiplication of two integers to have a runtime that is sub quadratic.

All in all, the HugeInteger class offers an effective and potent tool for handling huge integers. The implementation can be further optimised and improved using the performance analysis.