COMPENG 2SI4 Lab 1,2 Report

Areeb Syed

syeda16

400026264

Description of Data Structures and Algorithm

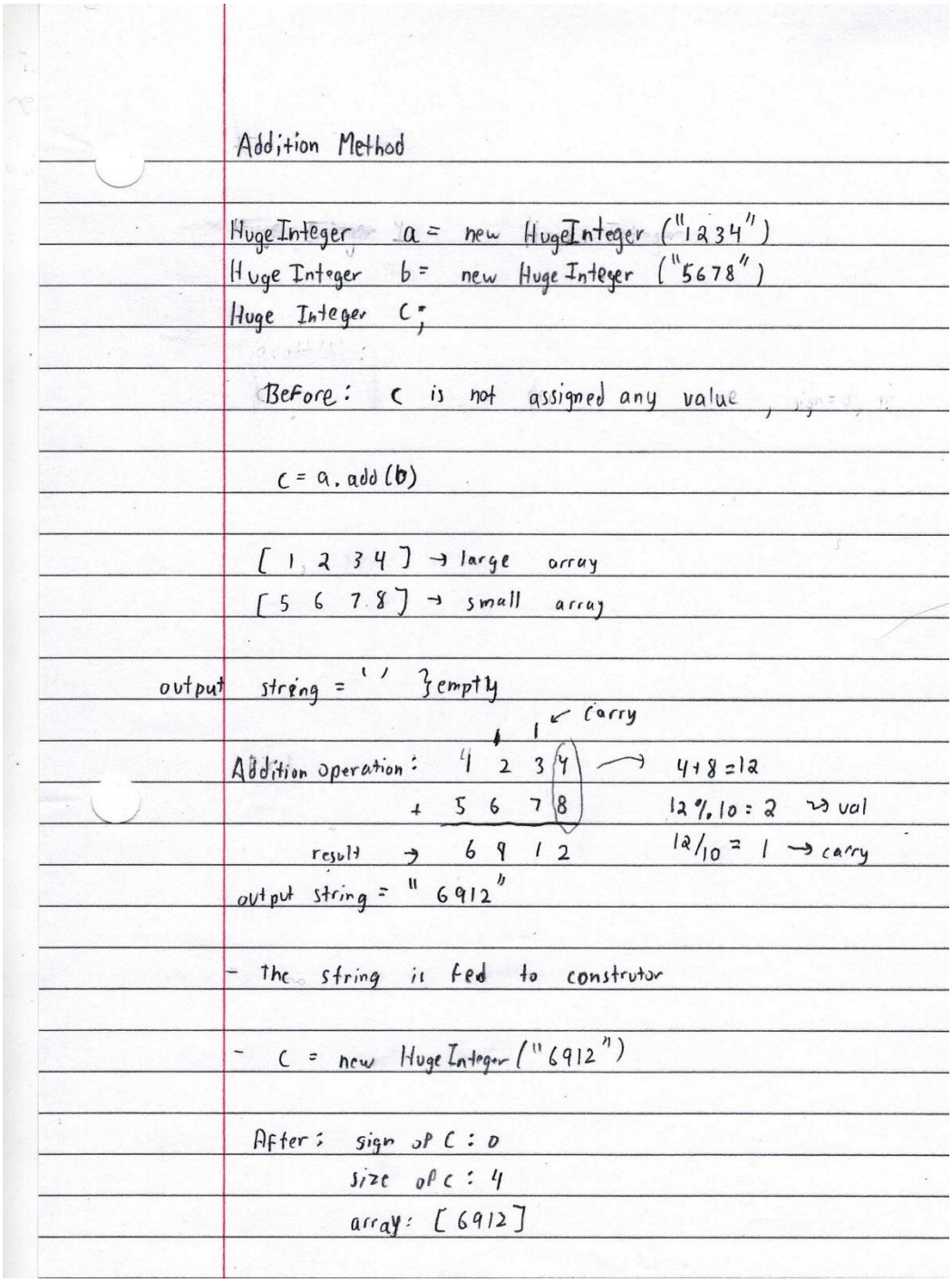
Data Structure

The Huge Integer class uses an integer array to store the digits of the Huge Integer. Additionally, it contains an integer variable called size which stores the number of digits in the Huge Integer. Also, it contains another integer variable that stores the sign of the Huge Integer. If sign is equal to 1, then the Huge Integer is negative and if sign is equal to 0, then the Huge Integer is positive. These three fields within the Huge Integer class need to be maintained at the execution of every constructor and method. For the constructor which has a string as a parameter, it first checks if all the inputs are viable. Then is removes any leading zeroes. Finally it loops through the string and places each digit into the array while also setting the size and sign field. The other constructor, which takes an integer n as a parameter, randomly generates integers and places it into an array of size n. It also randomly generates the sign of the Huge Integer. For example, if "-00123456789" was entered as the parameter of the constructor, then size = 9, sign = 1, and the array will be [1 2 3 4 5 6 7 8 9].

Addition Operation

For the addition operation, the algorithm used to perform the addition of 2 Huge Integers is the similar to how one would add any 2 integers. First, the sign of the 2 Huge Integers are compared and based on that a decision is made. If any one of the Huge Integers have a sign = 1, then the subtract function is called such that the Huge Integer with the sign = 0 will be subtracted by the Huge Integer with sign = 1. The corresponding Huge Integer returned from the subtraction function is returned. If both Huge Integers are negative, then a flag will be set to 1 to indicate that the final Huge Integer will be negative. If both are positive, then the addition starts. The size of both Huge Integers are compared and the integer arrays storing the digits are copied into a temporary array storing the larger and smaller Huge Integer. Then going through both arrays from the back, each corresponding digit in one array is added to the corresponding digit of the other along with a carry value initially set to 0 and stored in a temporary variable called add. Then the carry value is computed by taking the variable add and dividing it by 10. The value that represents the digit in the sum of both Huge Integers is found by taking the remainder of add/10 i.e. add%10 and adding it to the carry value. This final value is stored in a string. This process is repeated as we move towards the front of the arrays and each successive sum digit is added to the string which represents the addition of the 2 Huge Integers. This loop stops when we reach the front of the smaller array. Then afterwards we go through the rest of the larger array and compute the add variable by adding the corresponding digit with the previous carry value. We compute the new carry value using the same method as before. And we compute the final digit stored in the string the same way as well. When we reach the front of the large array, we exit the loop and check to see if we have any remaining carry value. If we do, we add it to the front of the string. Then we check to see if the flag that indicates that we have a final negative number is 1. If so, we add a negative sign in front of the

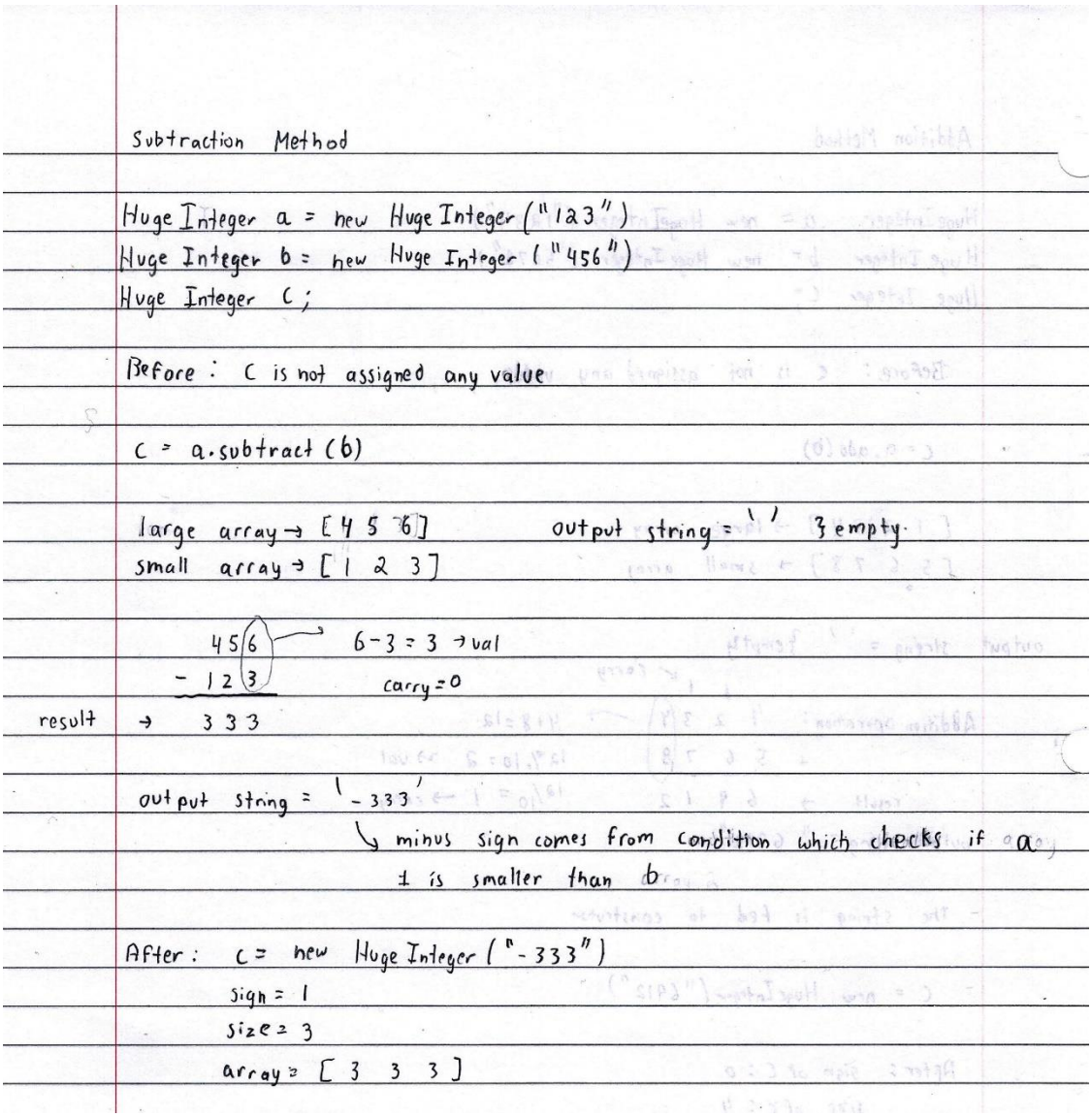
string. Finally, a new Huge Integer is created with the string as a parameter which represents the summation of the 2 Huge Integers inputted. The new Huge Integer is then returned. For example:



Subtract Operation

For the subtract operation, the algorithm used to perform the subtraction of 2 Huge Integers is the similar to how one would subtract any 2 integers. First, the sign of the 2 Huge Integers are compared and based on that a decision is made. If any one of the Huge Integers have a sign = 1, then the addition

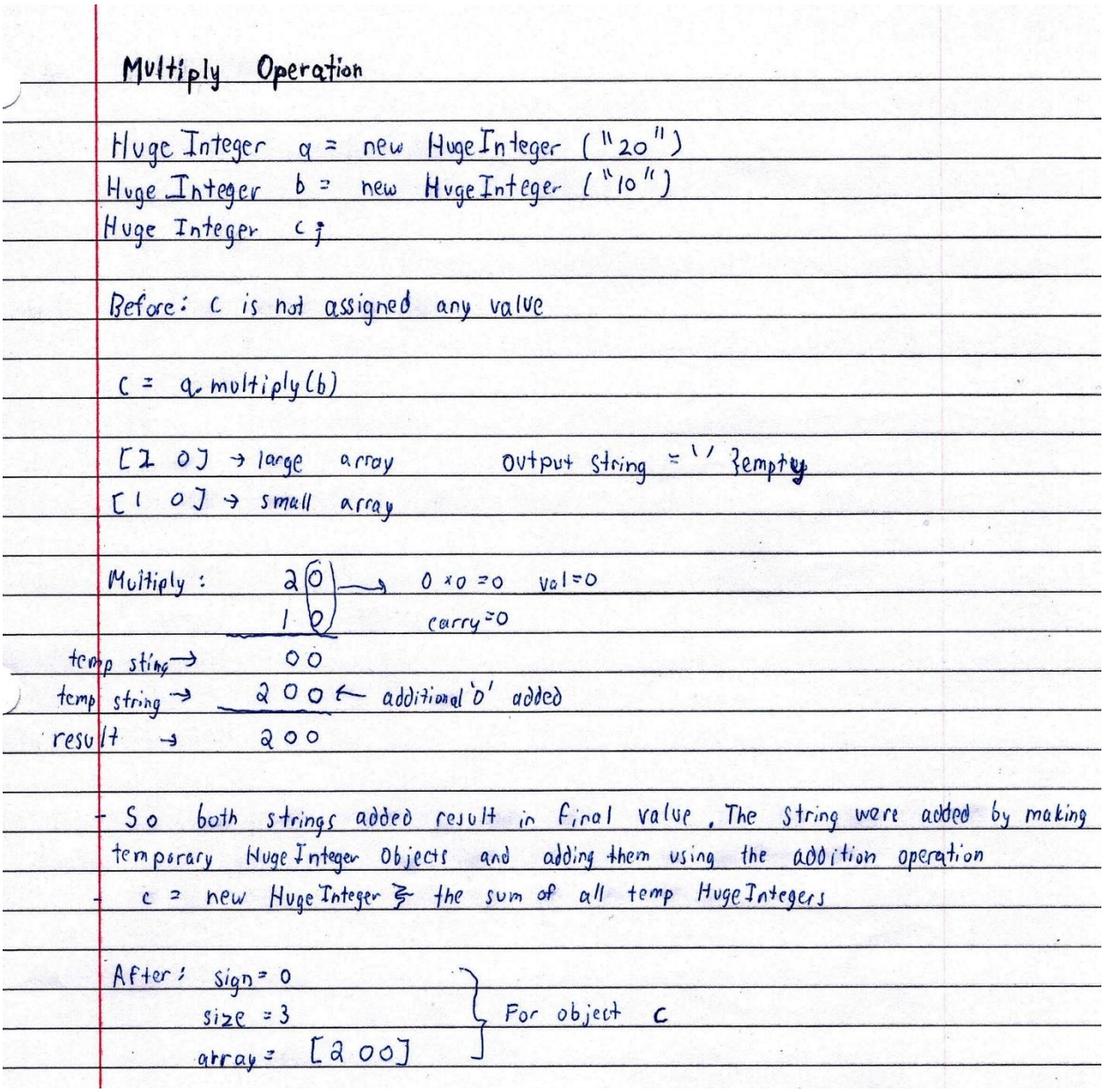
function is called such that if the first Huge Integer is negative, then the second Huge Integer's sign is change to 1. Then the addition function is called and the new Huge Integer returned is also returned by the subtraction function. If the second Huge Integer is negative, then its sign is changed to 0 and the addition function is called. Before the new Huge Integer returned by the addition function is returned by the subtraction function, any sign fields for any of the Huge Integers are set back to the original values. If both the Huge Integers are negative, both their signs are change to 0 and the subtraction function is called again. The new Huge Integer returned is returned by the subtraction function after all the sign fields are set back to their original values. If both the Huge Integers are positive, then the subtraction begins. Similar to the addition operation, the size of both Huge Integers are checked and based on that their arrays storing the digits are copied to temporary arrays that distinguish between the larger and smaller arrays. If both Huge Integers are the same, it returns a new Huge Integer with a parameter of a string 0. If both are the same size, it uses the compareTo operation to distinguish between the larger and smaller Huge Integer. If the first Huge Integer is smaller than the second Huge Integer, a flag is set to 1. After the large and small arrays are assigned then the subtraction begins. Using a loop to start from the back of both arrays and moving towards the front, if the corresponding digit from the larger array subtracted from the corresponding digit of the smaller array plus the carry value (which is initially 0) is less than 0, then the value stored in the string which represents the subtraction of 2 Huge Integers is equivalent to 10 plus the corresponding digit from the larger array subtracted by the corresponding digit of the smaller array plus the carry value. Then the carry value is set to -1. Otherwise the value stored in the string is equivalent to the corresponding digit from the larger array subtracted by the corresponding digit of the smaller array plus the carry value. The carry value is then set to 0. As this process is continued, each final value is added to the front of the string which represents the subtraction. This loop stops when we reach the front of the smaller array. Afterwards, a loop which continues through the rest of the larger array computes the digits of the subtraction as follows. If the corresponding digit from the larger array plus the carry value is less than zero, then the value stored is the corresponding digit from the larger array plus the carry value. The carry value is then set to -1. Otherwise, the value stored is the corresponding digit from the larger array plus the carry value. Then the carry value is set to 0. The final values are added to front of the string. After the loop is finished, the flag is checked. If it's equal to 1, then a minus sign is added to the front of the string. A new Huge Integer is created with the string as a parameter. The new Huge Integer is then returned. For example:



Multiply Operation

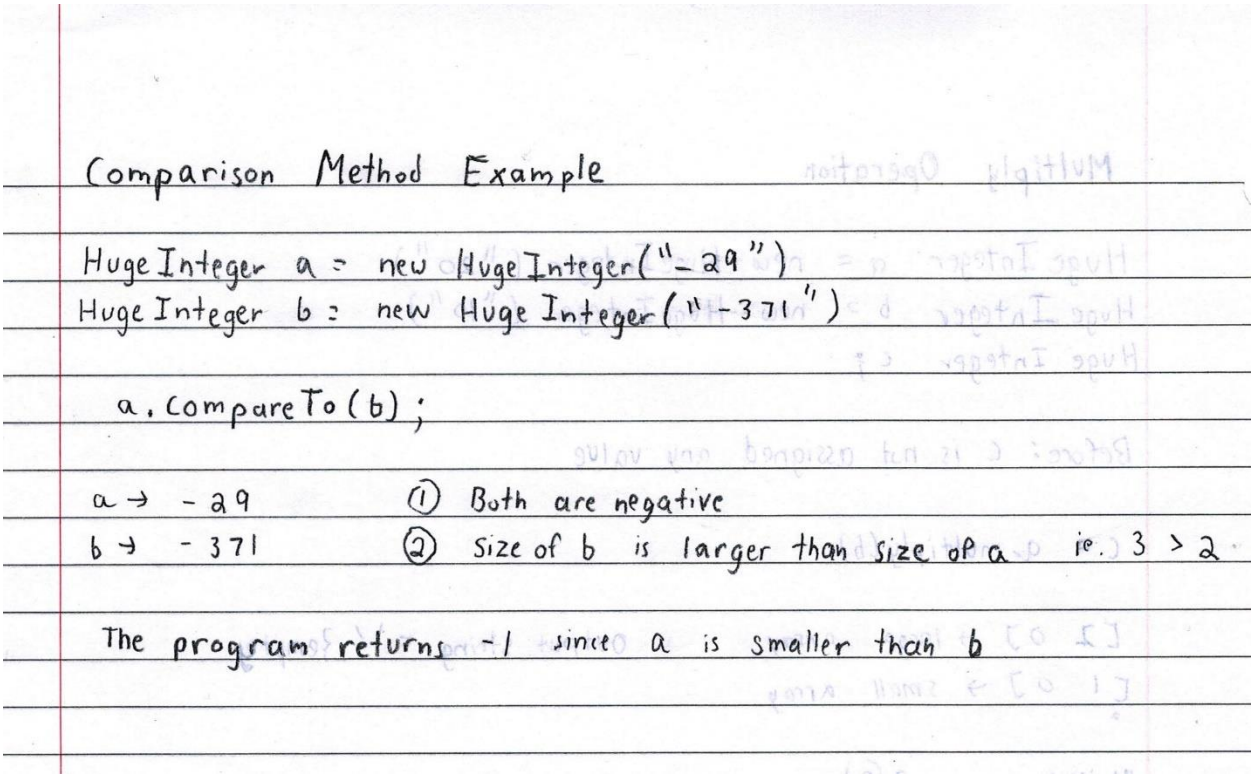
For the multiply operation, the algorithm is similar to the multiplication done by hand. First, the signs of both Huge Integers are compared. If any one of them have a sign = 1, then a flag is set to 1 to indicate the final huge integer to be negative. If both are positive or negative, then the flag is set to 0. The larger and smaller temporary arrays are assigned the integer arrays storing the digits of each Huge Integer based on which Huge Integer is larger and smaller, respectively. Then, a nested loop is used to compute the product of the 2 Huge Integers. While going through the smaller array from back to front, we set the string that stores the multiplication of that digits with the larger array to 0. Based on the value of the iteration variable, figure out how many zeroes need to be added to the empty string in advance. The carry value is set to 0 and we loop through the larger array from back to front. For each corresponding digit, the product is found by multiplying the corresponding digit in the smaller array by the corresponding digit in the larger array plus the carry value. The value stored in the string which represent the multiplication of the 2 Huge Integers depends on where we are in the larger array. If we are at the front of the larger array, then the value stored is equivalent to the product calculated above.

Otherwise elsewhere in the larger array, the value stored is the remainder of the product/10 i.e. product%10. The value is added to the front of the string. After going through the larger array and exiting the loop, a new temporary Huge Integer is created with the string as a parameter. The temporary Huge Integer is added to the final Huge Integer which stores the multiplication of the 2 Huge Integers. This is done through the addition operation. This process is repeated until we reach the front of the smaller array. Afterwards, the loop is exited and the flag is checked. If it's equal to 1, then the sign of the final Huge Integer is set to 1. The final Huge Integer is then returned. For example:



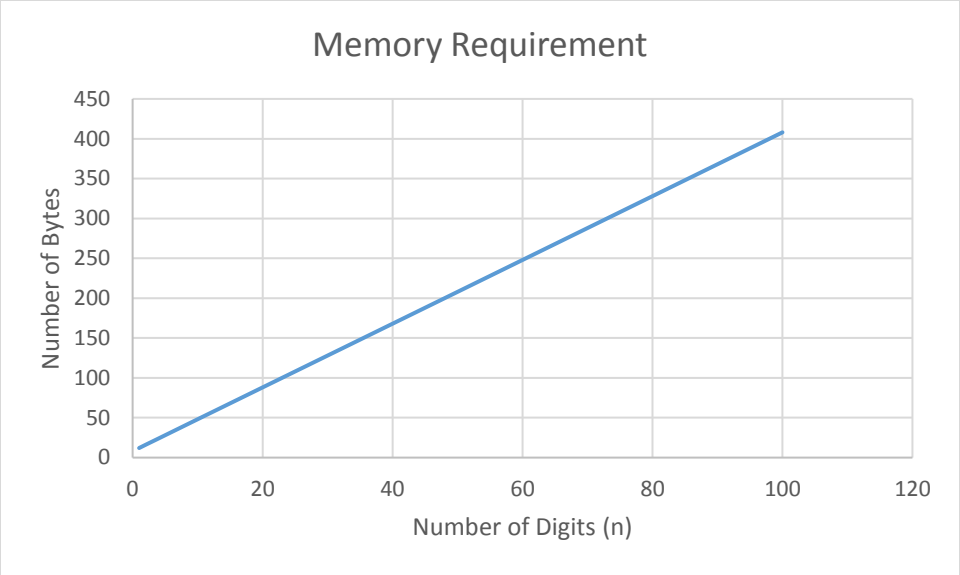
Comparison Operation

For the comparison operation, we begin by checking the signs of the 2 Huge Integers. If the first Huge Integer is negative and the second is positive, we return -1. If the first Huge Integer is positive and the second is negative, then we return 1. If both are positive then we check size of both Huge Integers. If the first Huge Integer is larger than the second, we return 1. Otherwise if the first Huge Integer is smaller than the second, we return -1. If both are the same size, then we iterate through both Huge Integer arrays and compare values. If one corresponding digit in the first array is larger than the corresponding digit in the second array, return 1. If it's the other way around, return -1. If you go through both arrays and find the digits to be equal, return 0. Similarly if both Huge Integers are negative, we compare their sizes. If the first Huge Integer has a size greater than the second Huge Integer, return -1. If it's the other way around, return 1. If both are the same size then we iterate through both Huge Integer arrays and compare values. If one corresponding digit in the first array is larger than the corresponding digit in the second array, return -1. If it's the other way around, return 1. If you go through both arrays and find the digits to be equal, return 0. For example:



Theoretical Analysis of Running Time and Memory Requirement

The amount of memory required to store an integer of n digits in my Huge Integer class is 4*n +8 bytes. This is because the array used to store the digits is an integer array which uses 4 bytes of memory. So 4*n represents the memory used by the array. The 8 extra bytes come from the 2 fields that store both the sign and the size of the Huge Integer.



Addition Operation:

Worst and Average Case Run Time: θ (n)

Extra Memory Requirement: θ (n)

The operation contains the many assignment and comparison statements but there are also 2 separate loops which result in the run time for both the average and worst case to be proportional to n (number of digits in the Huge Integer) since they always run regardless of how large n is. When computing the addition of the 2 Huge Integers, a new array is allocated in the constructor which allocates memory proportional to n.

Subtraction Operation:

Worst and Average Case Run Time: θ (n)

Extra Memory Requirement: θ (n)

Similar to the addition operation, this operation also contains many assignment and comparison statements. But the 2 separate loops account for the worst and average case runtime to be proportional to n since these loops are always executed regardless of how large n is. When computing the subtraction of the 2 Huge Integers, a new array is allocated in the constructor which allocates memory proportional to n.

Multiplication Operation

Worst and Average Case Run Time: θ (n^2)

Extra Memory Requirement: θ (n)

Aside from all the comparison and assignment statements, there is a nested while loop which accounts for the average and worst case run times to be proportional to n^2 since these loops are always executed regardless of how large n is. Since only the outer loop of the nested loop allocates memory to create a temporary object, this results in the extra memory requirement to be proportional to n.

Comparison Operation

Worst Case Run Time: θ (n)

Average Case Run Time: θ (1)

Extra Memory Requirement: θ (1)

This operation is mostly conditional statements except for a loop that runs only if the size of both Huge Integers are the same. This is why its worst is are the same being proportional to n since we may have to loop through and compare each element in each array. The average case is generally involving 2 Huge Integers that are not the same size, which is why it is proportional to a constant. The operation does not allocate any additional memory thus its extra memory requirement is proportional to a constant.

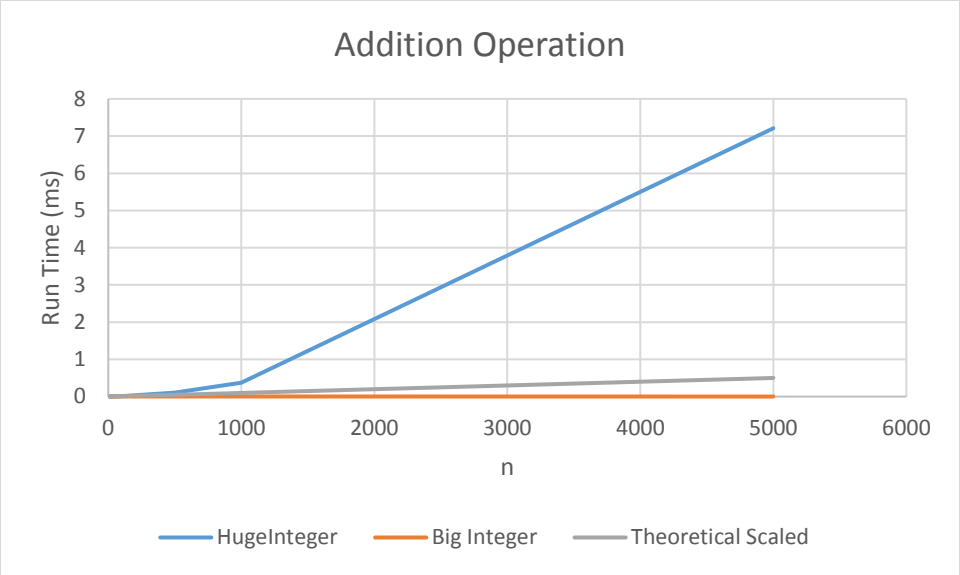
Test Procedure

For the test procedure, there are many possible cases that can arise. The methods have all be checked by many different test cases and the program does not fail. The inputs I will use are inputs that are handled by the method itself. All possible inputs have been considered and tested and the output results are correct. For example, the addition method was checked with a combination of positive and negative numbers that vary in size. The method successfully produced the correct output each time. All the outputs meet specifications. There were some difficulties debugging the code for inputs that were non-trivial but the problems were solved. I was able to check all input conditions and the program successfully produces a output, whether that be a result or exception.

Experimental Measurement, Comparison and Discussion

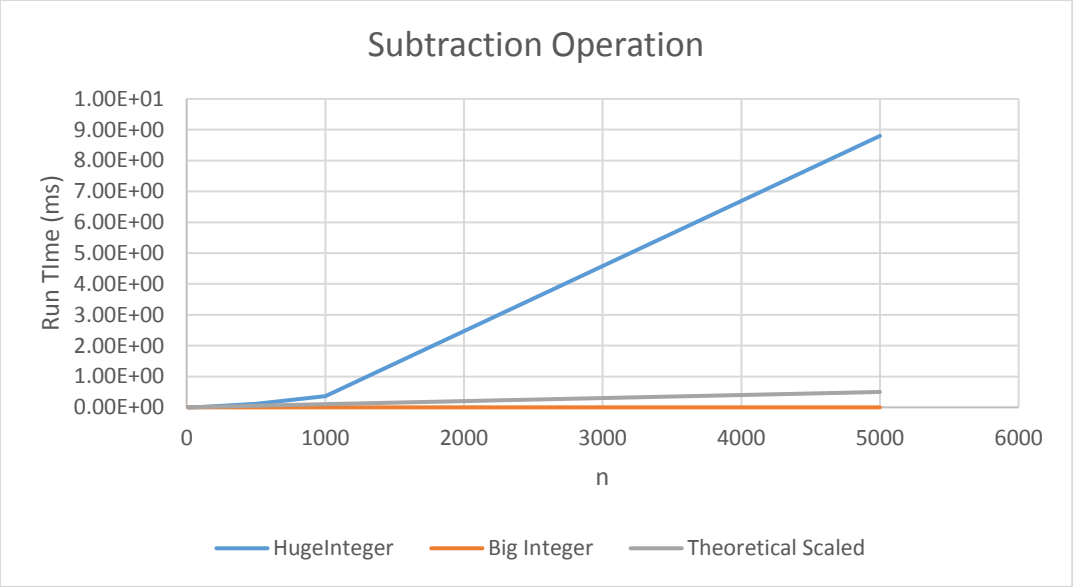
Addition Operation

n	Huge Integer	Big Integer	Theoretical Scaled
10	0.00114 [600000]	8.92E-5 [10000000]	0.001
100	0.00978 [60000]	1.2E-4 [4900000]	0.01
500	0.1041 [6000]	1.7E-4 [4500000]	0.05
1000	0.36874 [1500]	2.49E-4 [2000000]	0.1
5000	7.21 [70]	0.000786 [900000]	0.5



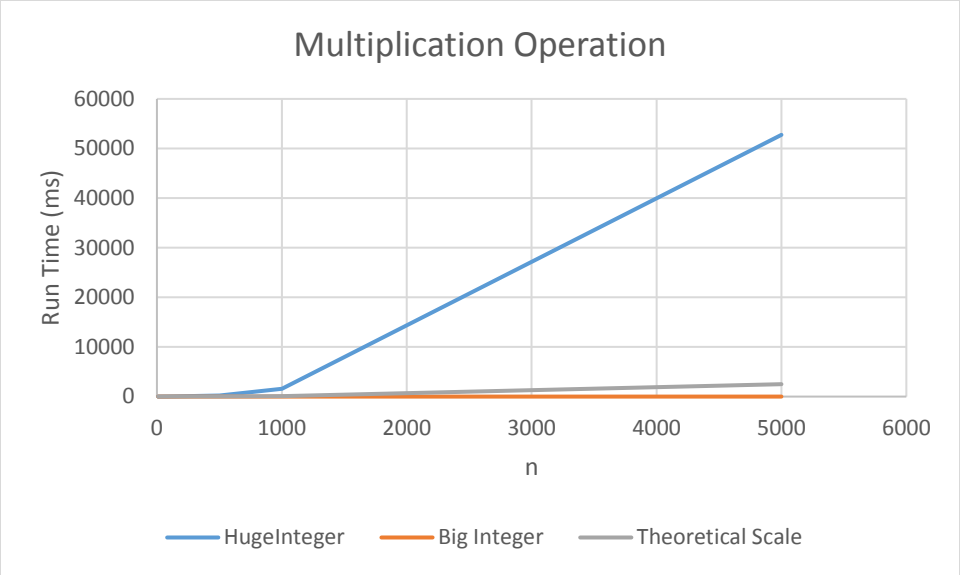
Subtraction Operation

n	HugeInteger	Big Integer	Theoretical Scaled
10	8.78E-4 [700000]	1.47E-5 [50000000]	0.001
100	0.00886 [65000]	2.3E-5 [30000000]	0.01
500	0.116 [6000]	3.63E-5[16000000]	0.05
1000	0.3676 [1500]	6.40E-5 [9000000]	0.1
5000	8.8 [70]	2.38E-4 [3500000]	0.5



Multiplication Operation

n	Huge Integer	Big Integer	Theoretical Scale
10	0.0257 [25000]	1.197E-4 [7790000]	0.01
100	3.57 [200]	2.14E-4 [4000000]	1
500	223.61 [3]	5.94E-4 [1000000]	25
1000	1560.24 [1]	0.00216 [300000]	100
5000	52753.69 [1]	0.042 [17000]	2500



Comparison Operation

n	Huge Integer	Big Integer	Theoretical Scaled
10	9.7875E-6 [80000000]	1.57E-6 [40000000]	0.000001
100	0.0000108 [80000000]	2.87E-6 [20000000]	0.000001
500	0.000012 [80000000]	4.45E-6 [15000000]	0.000001
1000	0.0000129 [80000000]	7.49E-6 [10000000]	0.000001
5000	0.000145 [80000000]	9.64E-6 [3000000]	0.000001

Obtaining the experimental data for large values of n in the multiply operation was a bit difficult as the run time was extremely long.

Discussion of Results and Comparison

The theoretical calculations were not exactly the same as the measured ones. They were in general much smaller than the actual measured calculations. Some of my experimental data makes sense, for example the addition and subtraction operations follow a linear trend which was predicted by their theoretical run times. The multiplication operation should follow a quadratic trend but it ended up following a linear trend that was quite deviated from the theoretical scaled trend. My implementation compared to the java class Big Integer is much slower in terms of run time. This is because my implementation is much more inefficient and a lot additional memory was allocated for each method. An improvement for my methods would be to implement a character array instead of an integer array, since they take up less memory. Also, the multiplication method can be done using Karatsuba's multiplication algorithm which would require recursion. This would reduce the theoretical run time of the multiplication method to be less than $\theta(n^2)$. Furthermore, I think if I were to implement the class using a linked list, it would make the class more memory and run time efficient.