**HugeInteger Report**

**Description of Data Structures and Algorithms:**

Data Structure:

The data structure used to store all HugeIntegers is an array of Integers. Storing the number as an array of integers is beneficial because I don’t have to keep switching from strings to integers and I can deal directly with the contents of the array when carrying out each operation. Each HugeInteger has an integer array called hugeIntArray which is used to store the digits of the number. Each HugeInteger also has a Boolean, negativeMarker, that acts as a negative flag. negativeMarker will be true if the HugeInteger is negative and false if the HugeInteger is positive. Finally, each HugeInteger has an integer, arrayLength, that stores the length of hugeIntArray and, consequently, the length of the HugeInteger.

Constructors:

To create HugeIntegers, I defined 3 constructors that are used for different cases. If HugeInteger() is called and no argument is passed, the constructor just sets the negativeMarker to false and leaves the hugeIntArray and the arrayLength undefined. Every time HugeInteger() is called, the hugeIntArray and the arrayLength need to be defined before the HugeInteger can be used.

The second constructor, HugeInteger(String val), takes a string as an argument. It then determines whether the number will be negative or not (by checking if the first character of the string is a negative sign) and sets the negativeMarker accordingly. After that, I loop through the string, checking that every character is a digit then adding it to the hugeIntArray as I go.

The third constructor, HugeInteger(int n), takes an integer as an argument and creates a random HugeInteger with n digits. I accomplish this by generating a random integer from 1 to 9 and making that random number the first digit of the HugeInteger. I then loop through the remaining indices of the hugeIntArray and assign each of them a random integer from 0 to 9. Exceptions will be thrown if anything is out of the ordinary (like n not being an integer.)

Add:

To add, I first define an integer, addCarry, to store the carry from each addition of 2 digits. I also defined an integer, tempNum, to store the result of the addition of 2 digits. To store the answer of the addition of the 2 HugeIntegers, I defined a new HugeInteger called addAnswer.

Addition is broken down to 4 cases depending on if there are negative numbers or not. If we had a.add(b), we would look at the negativeMarker of both a and b to determine which of the 4 cases we are dealing with: a and b are both positive, a and b are both negative, a is positive and b is negative, or a is negative and b is positive. In the cases where a and b are both positive or negative, we add the numbers and, in the case of a and b both being negative, set the negativeMarker to true. Addition is accomplished by doing basic addition and having a carry. We start from the least significant digit by accessing the digit at the last index of the hugeIntArray of both a and b and setting tempNum to the sum of said digits and addCarry. We then set the least significant digit of the hugeIntArray of addAnswer to: tempNum modulo (%) 10. Finally, we set the carry to tempNum/10 before moving on to the next set of digits.

In the remaining cases, I use the algorithm for subtraction to carry out the addition of 2 HugeIntegers where one of them is negative. I also call the absCompareTo method when subtracting to determine which number needs to be subtracted from the other. The absCompareTo method compares the absolute value of the 2 numbers passed to it. If we call a.absCompareTo(b), the method will return 1 if the absolute value of a is greater than the absolute value of b, 0 if they are equal, and -1 if the absolute value of a is less than the absolute value of b.

Subtract:

To subtract 2 HugeIntegers, I first had to define an integer, subCarry, to store the carry from each subtraction of 2 digits. I also defined an integer, tempNum, to store the result of the subtraction of 2 digits. To store the answer of the subtraction of the 2 HugeIntegers, I defined a new HugeInteger called subAnswer.

Like addition, subtraction is broken down into 4 cases depending of the sign of a and b in a.subtract(b). Looking at the negativeMarker of a and b, we would determine which of the 4 cases applies: a and b are both positive, a and b are both negative, a is positive and b is negative, or a is negative and b is positive. In the cases when a and b are both positive, we subtract the 2 numbers and set the answer negative if applicable. Subtraction is accomplished by doing basic subtraction and having a carry. If a is greater than b, we calculate a-b, but if b is greater than a, we calculate b-a and flip the answer’s sign. We start from the least significant digit by accessing the digit at the last index of the hugeIntArray of both a and b and setting tempNum to the difference between a and b. We then add subCarry to tempNum. If tempNum is greater than or equal to 10, we set subCarry to 1 and reduce the value of tempNum by 10. If tempNum is less than 0, we set subCarry to -1 and add 10 to tempNum. Otherwise, tempNum is kept the same and subCarry is set to 0. We then set the least significant digit of subAnswer’s hugeIntArray to the value of tempNum and move on the next set of digits.

In the remaining cases, we just add a and b and set the negativeMarker accordingly.

Multiply:

To multiply 2 HugeIntegers, I first had to define an integer, mulCarry, to store the carry from each multiplication of 2 digits. I also defined an integer, tempNum, to store the result of the multiplication of 2 digits and the addition of the carry. I also defined 2 HugeIntegers, mulAnswer and tempHugeInt, to store the product of the multiplication and to store the addition of the multiplication of each digit with the leading number respectively (this will be explained in more detail below.)

If a.multiply(b) was called, we’d start off by flipping a and b and saving them to the HugeIntegers thisFlipped and hFlipped respectively. We accomplish this by looping through a and b backwards and adding each digit (from least significant to most significant) to the HugeIntArray’s of thisFlipped and hFlipped respectively. After a and b are flipped (from now on, we will call them aFlipped and bFlipped,) we start to multiply. The approach that I took uses step by step long multiplication to find the product of a and b. We use nested loops to loop through a and b simultaneously. Every time we multiply a digit from b with the whole of a, we need to shift the next multiplication to account for the degree of the number being multiplied. By using nested for loops, we can keep track of the degree of the number we are multiplying with. The outer loop has index j that is incremented by 1 each time and loops through bFlipped. The inner loop has index i (also incremented by 1 each time) and loops through aFlipped. We then set tempNum to equal aFlipped[i] times bFlipped[j] plus what already exists at tempHugeInt’s hugeIntArray[i+j] and then we also add the mulCarry. This means that we take the least significant digit of b and multiply it with each digit of a and add the carry from the multiplication of each 2 digits. We then store that product that needs to be added to all the other products in tempHugeInt. By adding i to j, we make the first multiplication digit shift each time we complete the multiplication of a digit from b with a. tempNum modulo(%) 10 is then stored at the index i+j of tempHugeInt’s hugeIntArray. Finally, mulCarry is to tempNum/10 and added to the tempHugeInt’s most significant index (after the indexes that were used by the prior multiplication) before being reset to 0.

To retrieve the answer from tempHugeInt, we flip it by looping through it backwards and setting the values equal to the indexes of mulAnswer. Finally, we set the negativeMarker of the answer as follows: if both a and b are positive or negative, the negativeMarker is set to false, but if either a or b are negative, the negativeMarker is set to true.

Divide:

To divide, we first eliminate the simple cases. If we have a.divide(b), and b is zero, an ArithmeticException is thrown. We use absCompareTo to determine which of a or b’s absolute values are greater. If the absolute value of b is greater than the absolute value of a, we return a HugeInteger, zeroAnswer, that equals 0. If both a and b are positive or negative and equal to each other, we return a HugeInteger, oneAnswer, that equals one. Otherwise, if one of a or b is negative and their absolute values are equal, we set oneAnswer’s negative marker to true and return it, which equals -1.

Beyond that, my general approach was to count how many times I can subtract b from a using a while loop. Inside the while loop, we call the subtract() method to subtract the 2 HugeIntegers, a and b, from each other. We then increment a HugeInteger, divAnswer, by one each time and return it at the end of the function.

Finally, we set the negativeMarker of the answer as follows: if both a and b are positive or negative, the negativeMarker is set to false, but if either a or b are negative, the negativeMarker is set to true.

Comparison:

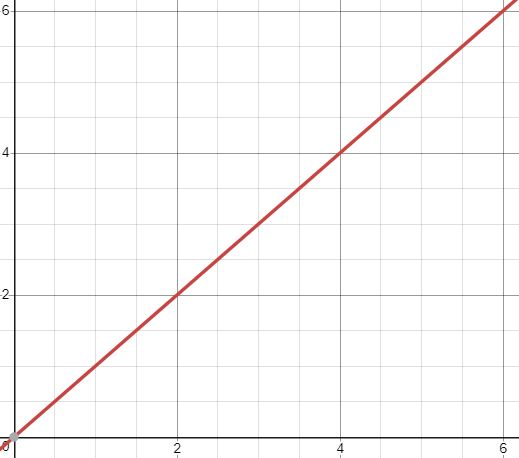
If we are comparing HugeIntegers a and b, I first compare the negativeMarkers of a and b. If a is positive and b is negative, we return 1. Otherwise, if a is negative and b is positive, we return -1. Secondly, if a and b are both positive or negative, I compare their lengths. If the length of a is greater than the length of b, we return 1. On the other hand, if the length of b is greater than the length of a, we return -1. If the length of a and b are equal, we loop through a and b (from the most significant digit to the least significant digit) and compare each digit. As soon as we find a digit that is larger than the other, we return the appropriate result. If we get to the end of a and b and both are the same, we return 0.

**Theoretical Analysis of Running Time and Memory Requirement:**

For all of the following calculations, I am assuming that for a call, a.method(b), a and b are HugeIntegers of the same length n.

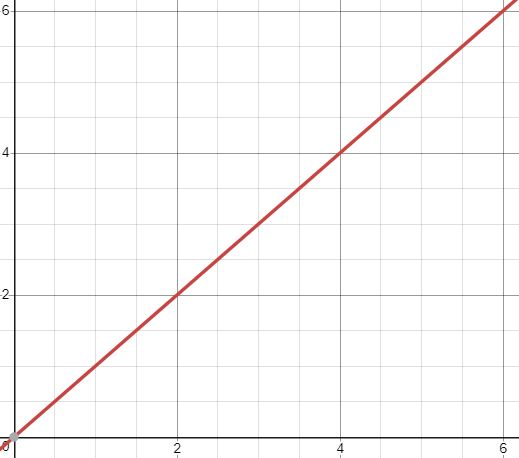
Memory to Store HugeInteger with n Digits:

Each HugeInteger uses at least 33 bits, or about 4 bytes, even if n is 0. hugeIntArray is an array of integers and therefor has the space complexity of 4n since each integer takes up 4 bytes. Therefore, the space complexity equation is M(n) = 4n + 4. We can also estimate that M(n) = Θ(n) (Big-theta of n.)

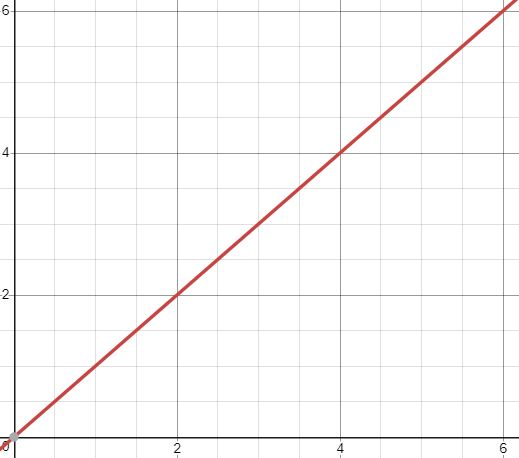


Time and Space Complexity for Addition:

The time complexity equation for addition is T(n) = 3n + 8 because there are 8 commands outside the for loop and the worst case inside the for loop has 3 commands. The for loop runs n times. Since there isn’t much variation in the code of add, the worst case and the average case are the same. Therefore, T(n) = O(n) = Θ(n).

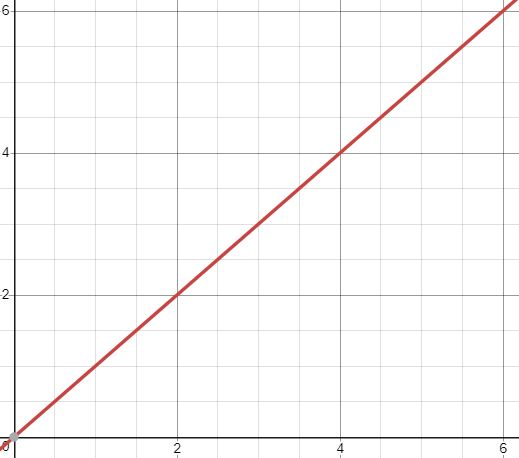


The space complexity for addition starts with the 2 integers defined at the beginning taking up 8 bytes. Then, our answer is stored in a HugeInteger that uses 4n + 8 bytes. Therefore, the space complexity equation for addition is M(n) = 4n + 8 + 8 = 4n + 16. Again, since the code doesn’t vary from case to case, the worst case and the average case will be the same. M(n) = O(n) = Θ(n).



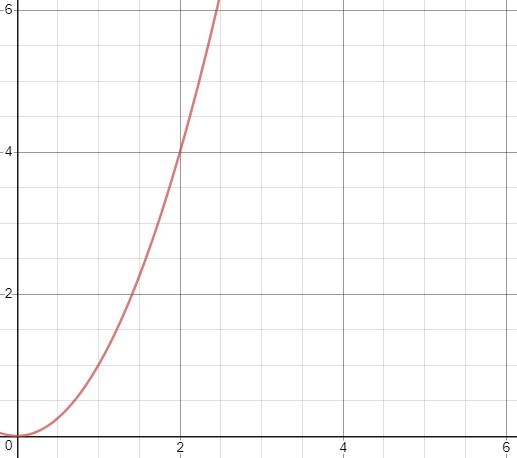
Time and Space Complexity for Subtraction:

Since subtraction has the same code as addition but the cases are flipped around, the space and time complexities will be the same. The time complexity equation will be T(n) = 3n + 8 🡺 T(n) = O(n) = Θ(n). The space complexity equation will be M(n) = 4n + 8 + 8 = 4n + 16 🡺 M(n) = O(n) = Θ(n). They also both have the same plot:

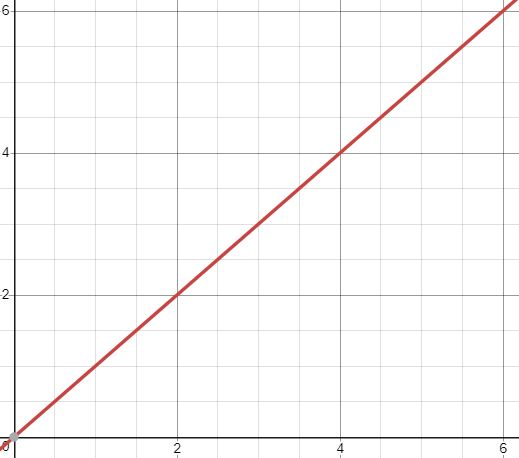


Time and Space Complexity for Multiplication:

The time complexity equation for multiplication is T(n) = 1n + 1n + (3n2 + 2n) + 1n + 17 = 3n2 + 5n + 17 because there are 17 commands outside of the for loops. There are 3 single for loops and 1 nested for loop. The single for loops have 1 command inside each. The nested for loop is made up of 2 for loops that run n times each. The inner one has 3 commands and the outer one has 2. Therefore, the worst case and the average case are: T(n) = Θ(3n2 + 5n + 17) = Θ(n2) = O(n2)

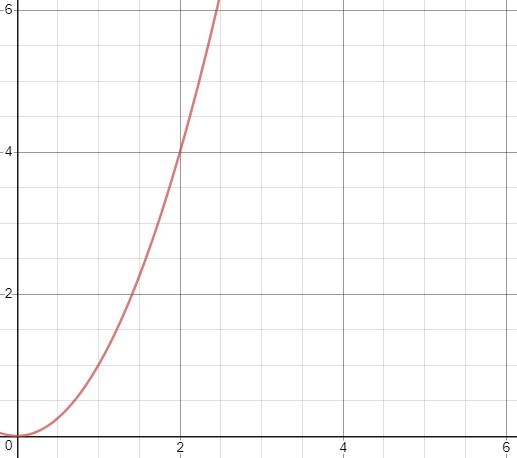


The space complexity for multiplication starts with 2 integers being defined which take up 8 bytes. We then define 4 HugeIntegers. 2 of the HugeIntegers take up space 8n + 4 and the remaining 2 take up 4n + 4. The space complexity equation will be M(n) = 8n+4 + 8n+4 + 4n+4 + 4n+4 + 8 = 24n + 24. The worst and average cases are: M(n) = O(n) = Θ(n).

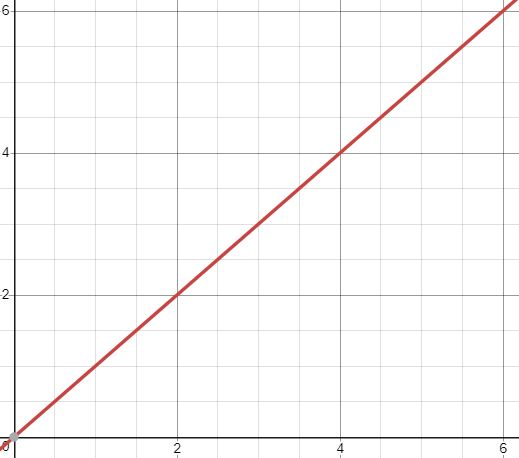


Time and Space Complexity for Division:

The time complexity equation for division is T(n) = n(3n+8 + 3n+8 + 1) + 15 = 6n2 + 17n + 15 because we have 15 commands outside of the while loop. The while loop will run a maximum of n times. Inside the while loop, we call both add and subtract which makes the run time of the n2 because we add the run times of add and subtract and multiply them by n. There is also another command inside the while loop. Therefore, the worst and averages cases are: T(n) = O(n2) = Θ(n2).



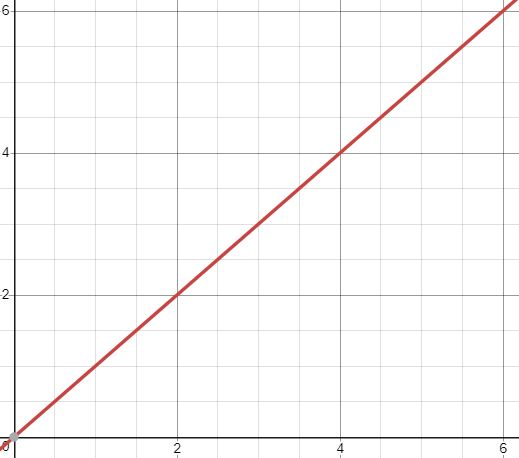
The space complexity of starts with the definition of 2 HugeIntegers with constant space 8 bytes for each or 16 bytes in total. We then define another 2 HugeIntegers that take up 4n+4 bytes each or 8n+8 bytes in total. We then define the HugeInteger, divAnswer, which also takes up 4n+4 bytes. Inside the while loop, subtract takes up 4n+16 and add takes up 4n+16. The space complexity equation is M(n) = 8n+32 + 4n+4 + 8n+8 + 16 = 20n + 60. Therefore, the average and worst cases are: M(n) = O(n) = Θ(n).



Time and Space Complexity for Comparison:

The equation for time complexity of comparison is T(n) = 4n + 2 because there are 2 commands outside the for loop. The for loop runs n times and has 4 commands inside it. Therefore, the for the worst case, T(n) = O(n). For the average case, T(n) = Θ(1) because a little more than half the cases will be eliminated either by length or by their sign.

Worst Case:



Average Case:



The space complexity is M(n) = 4 because we only define one integer that gets returned in the end. Therefore, the average and worst cases are: M(n) = O(1) = Θ(1).



**References:**

1. Raymond Lee, private communication, Jan. 2019.
2. Alex Hibbs, private communication, Jan. 2019.
3. Jesse Rowley, private communication, Jan. 2019.
4. Brandon Noble, private communication, Jan. 2019.
5. Abhay Patel, private communication, Jan. 2019.