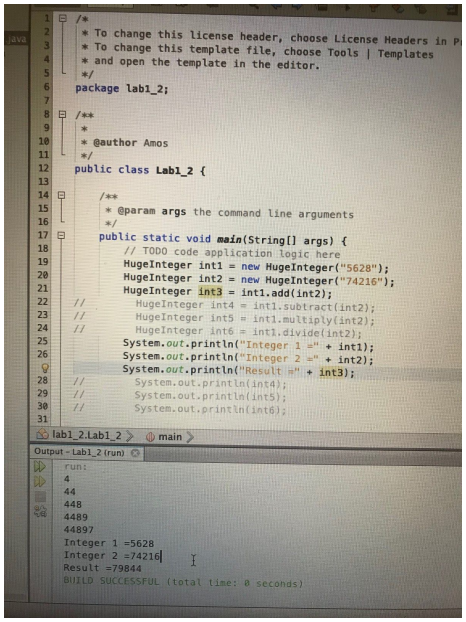
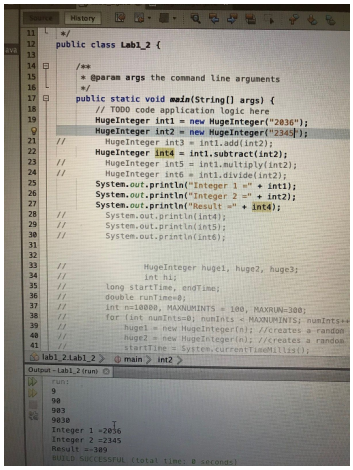
Description of Data Structure and Algorithms

I used strings to store the integers value. I picked this because it's the easiest way to store and also make changes. Also, because of the fact that we are already inputting a string to the constructor, it's easier to just keep it as a string rather than storing it as an array or linked list. For my HugeInteger class, I have three private members called hugeInt, size, and sign. HugeInt is a string of the integer inputted, excluding any 0 at the start before a non-zero integer and the negative sign if the number is negative. Size is how many digit the number is. In my constructor, I update the size of the number by going through the inputted string. For every 0 before a non-zero integer and a possible negative sign, I decrease the size by one so the size is a true representation of the length of the integer. Sign is a boolean value that defines if it's a positive or negative number. For positive, it is stored as true, and false for negative. So when I do my operations, I can just call sign and figure out the sign of the integer and result.

For addition, I did a basic addition digit by digit, so that no matter how big the integers are, they won't be out of range. Before actually doing addition, I first assume the second integer is bigger in size and store it in a variable call big, while also storing the first integer to variable small. But in order to make sure the second integer is actually bigger in size, I used the compare method to check. When the result of compare to is 1, which means the first integer is bigger, I will switch the value of big and small so that the variables are right. Then I start from the digit all the way from the right, and from there the addition of that one digit is executed. If the result of the digit addition is bigger than 10, then I subtract 10 away from it and put it into a variable ”carry”, which will be added to the next digit. In order to make it easier, I used the reverse function from stringbuilder so that instead of going through the integer from the back to front, I just compute it from left to right, and then reverse it back after addition is done. For the result, it was stored in a new string and goes through the HugeInteger constructor to make sure the outcome is a valid number without any problem. There are three special cases, which are adding a positive to a negative number, adding a negative to a positive number, or adding two negative numbers. For adding a positive to a negative number and adding a negative to a positive number, it is basically the positive number subtract the absolute value of the negative number, so I call the subtract function. But because it's a negative number, I have an absolute function that returns the absolute value of the number. And for adding two negative numbers, it is just two addition of the absolute value of the numbers but then setting the value of the result to be negative. Here is an example of 5628+74216. As it is shown in the result, the addition was done one by one starting from the right, but because it was reversed at the start, the value shown before actually printing the result is 44897, which is backward of the result 79844.

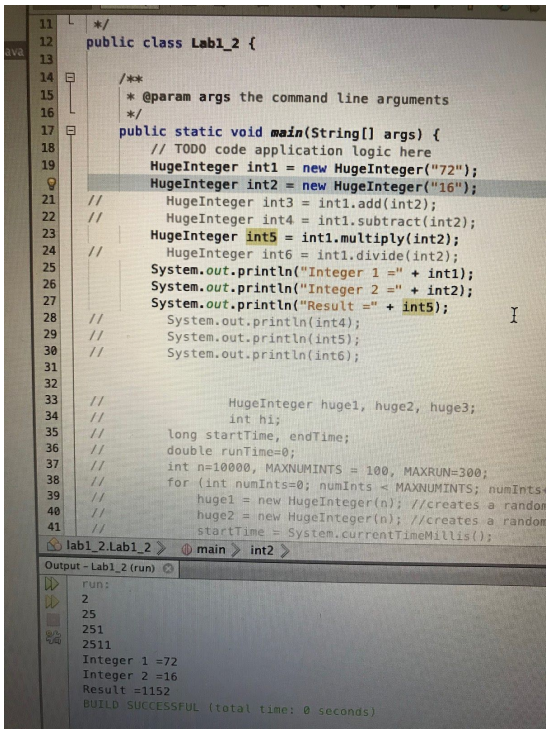


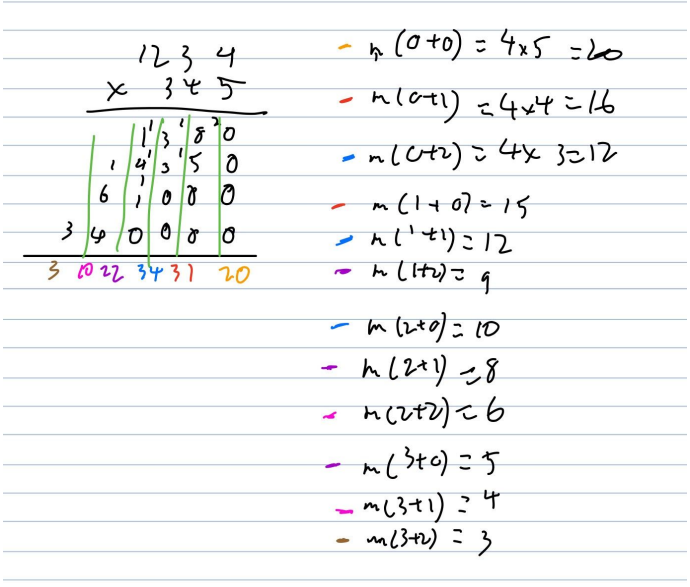
For subtraction, it is the same basic idea from addition. I do subtraction of each digit starting from the right, and if the value is negative, I add ten, which is the complement of the result, and will output the correct value at that digit. And if ten was added to the digit, I subtract one from the next digit, so that the “borrowing” in math operation is done. For the special cases of subtracting a positive to a negative number and subtracting a negative to a positive number, it is actually the positive number adding the absolute value of the negative number, so addition is called instead to do the operation. And for subtracting two negative numbers, it is just the the absolute value of the second number subtracting the absolute value of the first number. So here is an example of 2036-2345. As it shows, the result prints it digit and reverse it back and add a negative sign because the first integer is smaller than the second one.



For multiplication, I ran a nested for loops. I realize at the start the doing repeated addition will work, but it is not efficient at all. How the nested for loop works is that I realize when you do multiplication, we basically take the digit all the way on the right of the first, multiply it with the

each digit of the second integer, then add them together. And once you went through the first time and move to the next digit of the first number, the second round will be the same but multiplied by 10, or moved one digit to the left and added a zero there. So I can run a nested for loop, store all the values in an array, and then do addition of each row once the operation is done. And when the loop is done running and all addition is done, I check if any of the values are bigger than 10. If they are bigger, I find the remainder after dividing by 10, and store that to the digit of that position. Also, for the value at that position, I do an integer division by 10, store that as “carry”, and add it to the next digit I look at. The special case here is the signs of the integers. I use if statements to check the signs of the two integers, and return the correct sign. If they are the same sign(positive and negative), the result is positives. If they are a different sign, the result is negative. Here is an example of 72x16, and as it show, it do digit by digit, then gives the result of the actual answer at the end.





Theoretical Run Time Results & Memory Requirement

In the two constructors, the memory required to store the values are n. The memory required just purely depends on the length of the string. The run time for the two constructors are n, as it contains a for loop to generate random numbers according to the amount of numbers inputted or runs through to check for if the characters are valid numbers.

In addition, the memory required is n. The amount of memory it will take to store the two numbers for addition is equal to the number of digits of each number. And if the result of addition creates an extra digit, the memory required is n +1. And for run time, the operation

goes through a bunch of if statements to check for signs and carries then a for loop to add each number, so the run time should be n.

In subtraction, it is also n for memory required. The amount of memory to store the two numbers is the number of digits of each number. During the operation, it is doing digit by digit subtraction, and the memory required to store it is n -1. And for run time, the operation goes through a bunch of if statements to check for signs and borrow then a for loop to subtract each number, so the run time should be n.

In multiplication, the memory required to store each number is n. But after the operation, the memory required to store the product is 2n as the amount of numbers can be doubled. And for run time, it takes n^2 as the operation runs through a nested for loop to do the digit by digit multiplication and then add them up after.

Test Procedure

Here are the results of the test cases from the announcements and their results.

```
    public static void main(String[] args) {
        // TODO code application logic here
        HugeInteger int1 = new HugeInteger("99999");
        HugeInteger int2 = new HugeInteger("99998");
        HugeInteger int3 = new HugeInteger("-99999");
        HugeInteger int4 = new HugeInteger("0");
        HugeInteger int5 = int1.add(int3);
        HugeInteger int6 = int1.subtract(int2);
        HugeInteger int7 = int3.multiply(int4);
//        HugeInteger int6 = int1.divide(int2);
        System.out.println("Integer 1 = " + int1);
        System.out.println("Integer 2 = " + int2);
        System.out.println("Integer 3 = " + int3);
        System.out.println("Integer 4 = " + int4);
        System.out.println("Result of 99999 + (-99999) = " + int5);
        System.out.println("Result of 99999 - 99998 = " + int6);
        System.out.println("Result of -99999 *0 = " + int7);
//        System.out.println(int4);
//        System.out.println(int5);
//        System.out.println(int6);


//                HugeInteger huge1, huge2, huge3;
//                int hi;
```

```
ab1_2.Lab1_2 > main >
itput - Lab1_2 (run)
run:
Integer 1 = 99999
Integer 2 = 99998
Integer 3 = -99999
Integer 4 = 0
Result of 99999 + (-99999) = 0
Result of 99999 - 99998 = 1
Result of -99999 *0 = 0
BUILD SUCCESSFUL (total time: 0 seconds)
```

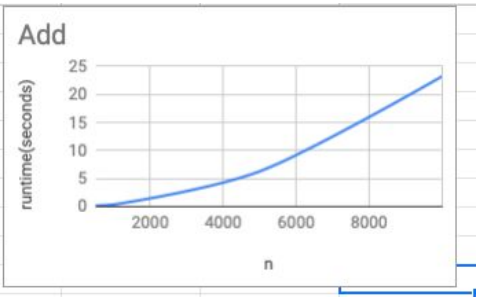
Experimental measurements, comparison and discussion

So, basically I used the code given for the run time analysis to calculate the run time required for each operation. And in order to graph and find the trend of each operation, I ran the program

with 6 different number of random digits so that I can see the relationship of the run time and the number of random digits, which provides me with a run time result.
HugeInteger:

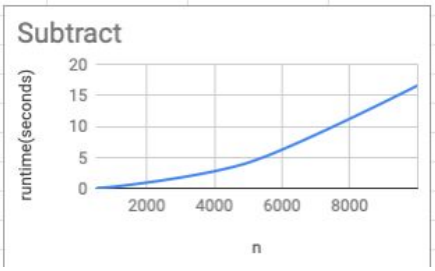
Add

n	maxnumints	maxrun	runtime(seconds)
10	100	300	0.01513333333
100	100	300	0.03806666667
500	100	300	0.1435333333
1000	100	300	0.3456666667
5000	100	300	6.2675
10000	100	300	23.1988



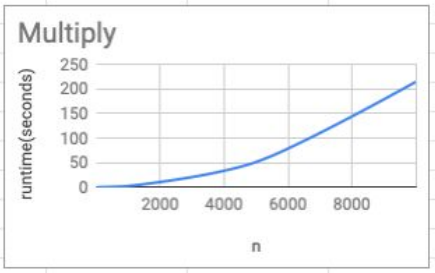
Subtract

n	maxnumints	maxrun	runtime(seconds)
10	100	300	0.009033333333
100	100	300	0.0375
500	100	300	0.1378
1000	100	300	0.3152333333
5000	100	300	4.213033333
10000	100	300	16.61196667



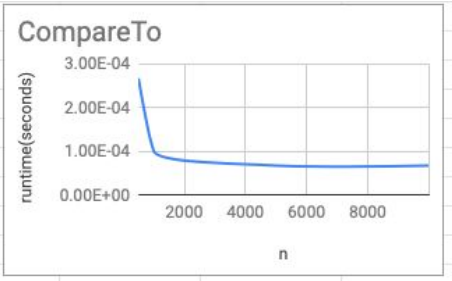
Multiply

n	maxnumints	maxrun	runtime(seconds)
10	100	300	0.01456666667
100	100	300	0.05973333333
500	100	300	0.6352
1000	100	300	2.234833333
5000	100	300	51.1015
10000	100	300	214.3349667



CompareTo

n	maxnumints	maxrun	runtime(seconds)
10	100	300	1.33E-04
100	100	300	6.67E-05
500	100	300	2.67E-04
1000	100	300	1.00E-04
5000	100	300	6.67E-05
10000	100	300	6.67E-05

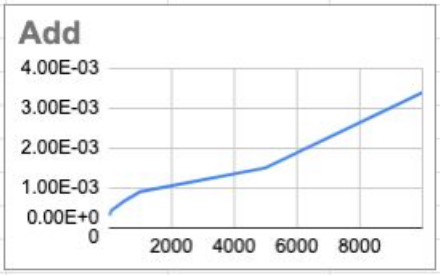


As it shows in the runtime result, all operations have a close to linear runtime.
There was one small problem during the runtime process, and that is the long build time. When doing multiply on 10000 digits, it took a total of 105 minutes to build after the laptop crashed on the previous try.

BigInterger:

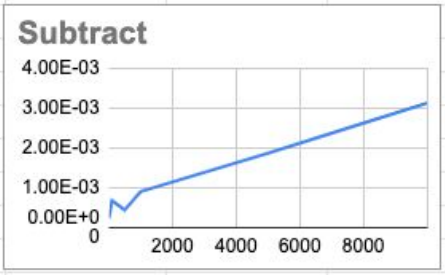
Add

n	maxnumints	maxrun	runtime (seconds)
10	100	300	3.00E-04
100	100	300	4.33E-04
500	100	300	6.67E-04
1000	100	300	9.00E-04
5000	100	300	0.0015
10000	100	300	0.0034



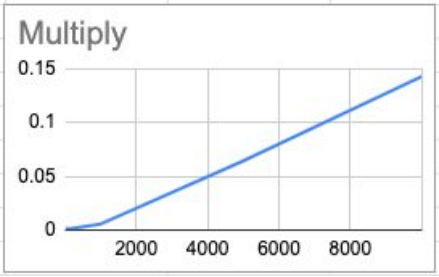
Subtract

n	maxnumints	maxrun	runtime (seconds)
10	100	300	2.33E-04
100	100	300	6.67E-04
500	100	300	4.33E-04
1000	100	300	9.00E-04
5000	100	300	0.001866666667
10000	100	300	0.003133333333



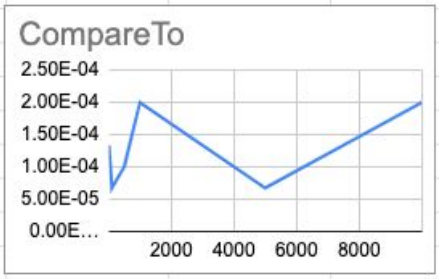
Multiply

n	maxnumints	maxrun	runtime (seconds)
10	100	300	0.0012
100	100	300	6.33E-04
500	100	300	0.002566666667
1000	100	300	0.005
5000	100	300	0.0643
10000	100	300	0.1433333333



CompareTo

n	maxnumints	maxrun	runtime (seconds)
10	100	300	1.33E-04
100	100	300	6.67E-05
500	100	300	1.00E-04
1000	100	300	2.00E-04
5000	100	300	6.67E-05
10000	100	300	2.00E-04



Compared to BigInteger, the run time of HugeInteger is way longer. BigInteger has a really linear run time, no matter how large n is, while HugeInteger run time looks more like an exponential graph. From this, I realize that there are more improvements that can be done to the code, which might be adding other functions to do error checking or a faster way to do the operations, especially for bigger numbers.

Discussion

In the first lab, I actually attempted to implement the lab in an array, but it wasn't a good choice as resizing an array is impossible, which makes it harder to create the functions as more has to

be taken into account. Because of this, I changed my implementation method and used strings to store my integers and do operations on. If extra time was given, I would implement my divide function in another way. The current way of division is the easiest to implement, but the runtime is really slow as it has to keep calling the subtract function and do the operation one by one. I would have implemented it similar to multiply, by looking at digits one by one and do operations on it.