Comp Eng 2SI3
Lab 1 & 2 - Report

Instructor: Dr. Thamas

Stefan Tosti - L06 - Tostis - 400367761

As a future member of the engineering profession, the student is responsible for performing the required work in an honest manner, without plagiarism and cheating. Submitting this work with my name and student number is a statement and understanding that this work is my own and adheres to the Academic Integrity Policy of McMaster University and the Code of Conduct of the Professional Engineers of Ontario. Submitted by
[Stefan Tosti, Tostis, 400367761]

Description of Data Structures and Algorithm

My HugeInteger class implements huge integers with the use of the Vector library. Vectors in C++ are just like arrays, except their size is mutable, meaning it can be changed throughout the runtime of the code. The vector itself holds Int data types, each digit in the desired HugeInt is a single index within the vector. In addition to this, the sign of the hugeInt is kept track of using a Public Integer variable called isNegative. This variable is 0 if the value is positive, and 1 if the value is negative. For example, if the desired huge integer to be created was -543678 then my class would store this as 2 components - the vector would look like <5 4 3 6 7 8> and the corresponding sign would be stored in isNegative = 1.

The only additions made to the hugeInteger class are the aforementioned addition of the isNegative public variable, as well as another constructor. This constructor accepts a pointer to a vector of integers, and turns that vector into a huge integer. The reason that this is needed is because in my methods, the final thing to be returned by the method is often a locally created vector, and thus I need to convert these vectors into hugeInteger's before returning them to the caller.

The toString method is implemented pretty simply. An empty string is first initialized, then the hugeInt is checked for negative or non-negativeness. If the hugeInt is negative, then the first thing appended to the string is a negative sign. After this, I simply iterate through each index of the hugeInt, appending each item to the string.

compareTo is also a relatively simple implementation. The first thing I do is check if the two hugeInts are the same thing, by using the toString method and checking equality. If this is met, then I return 0. Next, I check if one and only one of the numbers is negative. If this is the case, then it is easy to conclude that the non-negative hugeInt is the larger one, and thus an appropriate indicator is returned. The next case is that both of the hugeInts are non-negative. If they are both non-negative, then we compare their sizes and return an appropriate value based on which one is longer. If they are the same size, then we can simply check their first digit to determine which one is larger in magnitude. This algorithm is the exact same for two negative hugeInts, except the return values are flipped, since a negative number that is larger in magnitude is actually smaller in value.

Addition is the first major method of the class. Addition basically consists of two major sections. The first section is if both of the hugeInts are negative, or both are positive. These two are grouped together because they employ the same kind of operation. The only difference between these 2 cases is that when adding 2 negative numbers, you have to make sure to set isNegative equal to 1 for the final vector, but for 2 non negative numbers, isNegative is equal to 0. The logic in this method is akin to manually performed, long addition. We iterate through the smaller of the 2 hugeInts, each time adding corresponding indices of the smaller hugeInt to the larger hugeInt.

The sum of each addition is split into a sum and a carry. The sum is what actually gets added to the final sum vector, and the carry is used to carry over into the next addition of indices. Once we have iterated completely through the smaller vector, the remaining values are moved from the larger vector to the sum vector, still taking into account a carry from the last addition step. The second section is when we add a positive and a negative hugeInt together. This section is actually quite simple, because I can use my subtraction method to my advantage, and simply call that instead of re-implementing negative addition.

Subtraction is implemented with very similarity. I begin by checking cases that will simplify my work. The case scenario where we have $+ Huge - (- H)$ we can simply call our addition function. Conversely, the case where we have $- Huge - (+ H)$ can be simplified to $(Huge + H) \cdot (- 1)$. As can be seen, these two cases can pretty easily be seen as addition, and thus my addition method was used as implementation. If none of these cases are true, then a subtraction algorithm must be implemented. Similarly to addition, I begin by iterating through the smaller hugeInt. I define a temporary difference as the difference in corresponding indices of the two hugeInts. If this temporary difference is negative, then I know I will need to borrow from the next number, thus adding 10 to the given index in the larger hugeInt. Once we have gotten to the end of the smaller hugeInt, we transfer over all of the remaining values from the larger hugeInt into the final difference vector.

The final method implemented was Multiply. Much like subtraction and addition, this was implemented based on the algorithm for long multiplication. For each index in the “bottom number” we iterate through each index in the “top number”, which makes multiplication far more computationally intensive for extremely large numbers. Multiplication is based around a nested for loop, which lets us iterate through each top index, for each individual bottom index. At each step, we multiply the corresponding indices together. This is then appended to our final product vector, taking into consideration any carry.

Theoretical Analysis of Running Time and Memory Requirements

If we think about the construction of each hugeInt, the total amount of memory required is dependent on the number of integers, *n*, as well as other constant-valued additions. Each element of the array is a single Int, which takes up 4-bytes in C++. In addition to this, each hugeInt has another Int, *isNegative*, associated with it that indicates the sign. Since the hugeInt’s are implemented through the use of a vector, there is also an inherent pointer that C++ stores, pointing to the head of the vector. A pointer does not have a fixed size in C++, though it is widely agreed upon that 4-bytes is an accurate representation of pointer size. This means that for an *n* digit hugeInt, we can represent the size, *S(n)*, as…

S(n) *= 4n + 4 + 4 =* ***4n + 8 bytes***

The worst case run time of Add is $\Theta(n)$. This is because the function independently iterates through the largest hugeInts digits, and does not include nested loops. The maximum number of times that this function will run is *n* times, when *n* is the number of digits in that larger hugeInt being given to us. Thus, the maximum run time occurs when *n* is at its maximum value. The average run time of this function is also $\Theta(n)$, in that it is linearly related to n. In terms of extra memory, Add implements a new vector of Int's which we know required *4n* bytes of data. This vector is then taken, and turned into a hugeInt with my new constructor. Additionally, there are 6 new integers defined in this function, for a total of 6 * 4 = 24 bytes. Thus, the total extra memory for Add is… ***S(n)*** *= (4n + 8) + (4n + 4) + 24 =* ***8n + 36 bytes***

The worst case run time of Subtract is $\Theta(n)$. This is because the function independently iterates through the largest hugeInts digits, and does not include nested loops. The maximum number of times this function will run is *n* times, when *n* represents the size of the larger hugeInt. The average run time of this function is also $\Theta(n)$, since it is also linearly related to n. In terms of extra memory, Subtract implements a new hugeInt, 3 new integer vectors, and 4 new integer constants. Thus, the total extra memory for Subtract is… ***S(n)*** *= (4n + 8) +3(4n+4) + 3(4) =* ***14n + 32 bytes***

The worst case run time of Multiply is $\Theta(n^2)$. This is because the function iterates through each index of one hugeInt, and for each index iterates through every index in the other hugeInt. This means that, in an implementation sense, the code includes a nested for loop, with each for loop having linear complexity. Thus, the total complexity of the multiply function would be *n * n* or n^2. The average run time of the function is also $\Theta(n^2)$, which is also as a result of two linear for loops, making up a single nested for loop. The multiply function employs an extra vector, an extra hugeInt, 6 extra Integers. Thus, the total extra memory for Multiply is ***S(n)*** *= 2(4n + 4) + (4n + 8) + 6(4) =* ***12n + 40 bytes***

The worst case complexity of compareTo is $\Theta(1)$. The worst case scenario would occur when the two numbers have the same sign, and the same number of digits. In this case, rather than iterating through one of the hugeInts, I simply check the first index and compare their values. In C++, this operation has constant complexity. The average run time would also be $\Theta(1)$ for this function, for the reasons previously explained. This method does not create any new vectors or hugeInts, and just uses 3 new Integers values, Thus, ***S(n)*** *= 3(4) =* ***12 bytes***

Test Procedure

To test the string input constructor, I will use the following inputs…

- Empty String: (“ ”)
- Leading Zeros: (“0000789”)
- Invalid Characters: (“GHJ@@34$”)
- Positive and Negative Number: (“-100”) & (“100”)

To test the integer input constructor, I will use the following inputs…

- Less than 0: (-3)
- Equal to 0: (0)
- Greater than 0: (32)

To test the Add method, I will use the following inputs…

- Two negatives
- HugeInt = negative, h = positive
- HugeInt = positive, h = negative
- HugeInt = 0, h = positive or negative
- HugeInt = positive or negative, h = 0
- HugeInt and h require a final carry digit (extra index for last addition)
- HugeInt and h are the same number

To test the Subtract method, I will use the following inputs…

- Two negatives
- HugeInt = negative, h = positive
- HugeInt = positive, h = negative
- HugeInt = 0, h = positive or negative
- HugeInt = positive or negative, h = 0
- HugeInt and h require a final carry digit (extra index for last addition)
- HugeInt and h are the same number

To test the multiplication method, I will use the following inputs…

- Two negatives
- HugeInt = negative, h = positive
- HugeInt = positive, h = negative
- HugeInt = 0, h = positive or negative
- HugeInt = positive or negative, h = 0
- HugeInt and h require a final carry digit (extra index for last addition)
- HugeInt and h are the same number

To test the compareTo method, I will use the following inputs…

- One of HugeInt or h are zero
- HugeInt = negative, h = positive
- HugeInt = positive, h = negative
- Both positive
- Both negative
- HugeInt and h are the same number PAST the first digit

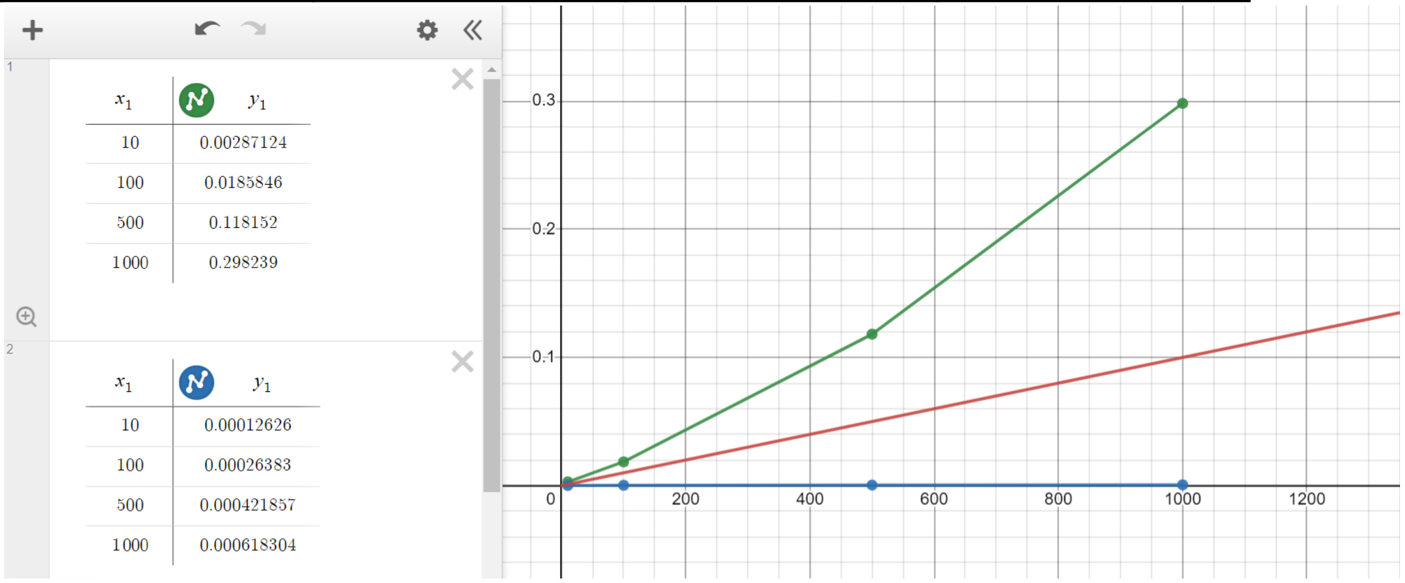
Given the above test case procedure, my code passed almost all of the tests, with the exception of one! The one test that it doesn't pass is using the compareTo function for HugeInt and h are the same number PAST the first digit. Despite this, I know exactly why this fails. This is because in my implementation, I only compare the first digit of two numbers that are the same length. I did this to save on computation time, and also because it passes all of the test cases. In order to fix this I would need to iterate through, and compare each digit of HugeInt and h, until I get a different number. The greater of these two numbers will belong to the larger of the two hugeInts. Additionally, my code does not pass the test case for "Addition in same size and different sign without borrow or carry". This was one of the test cases used to test in the lab, and despite many attempts to fix this issue, I could not resolve it in time for the interview.

In debugging my code, one thing that I noticed was that a lot of the time the numbers that I was getting as output were being computed correctly, but were often in reverse order as to what I was expecting. This was relatively simple to debug, since the console shows the output for each function when it compares it to the desired output.

Experimental Measurement, Comparison, and Discussion

Add

n	My function (ms)	Boost (ms)	Experimental
10	0.00287124	0.00012626	$\Theta(n)$
100	0.0185846	0.00026383	$\Theta(n)$
500	0.118152	0.000421857	$\Theta(n)$
1000	0.298239	0.000618304	$\Theta(n)$

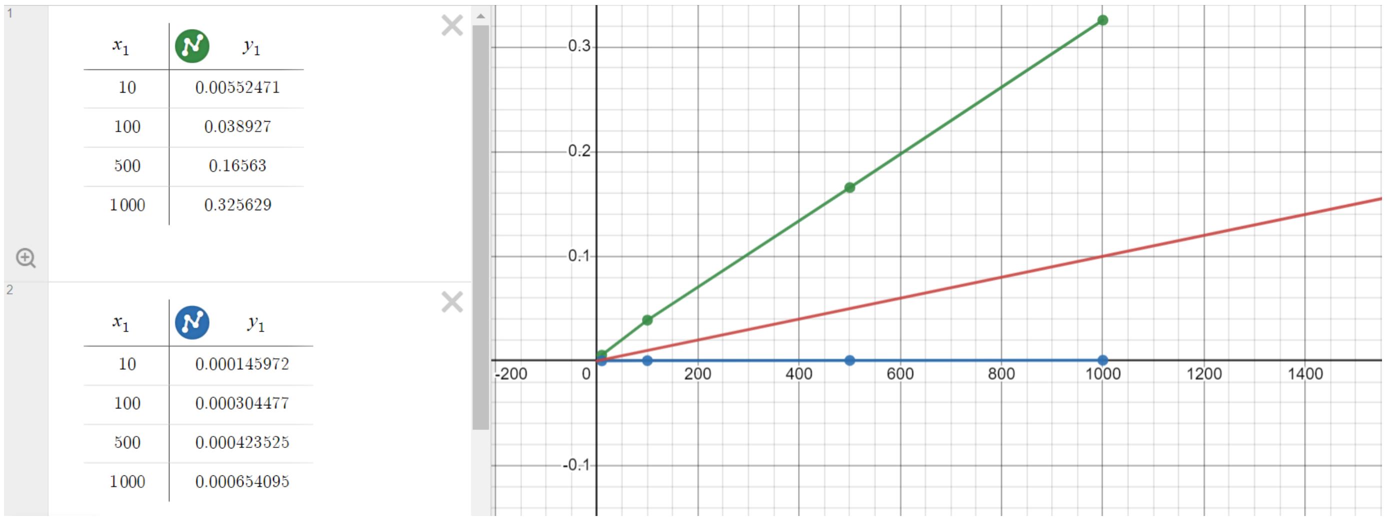


x_1	y_1
10	0.00287124
100	0.0185846
500	0.118152
1000	0.298239

x_1	y_1
10	0.00012626
100	0.00026383
500	0.000421857
1000	0.000618304

Subtract

n	My function (ms)	Boost (ms)	Experimental
10	0.00552471	0.000145972	$\Theta(n)$
100	0.038927	0.000304477	$\Theta(n)$
500	0.16563	0.000423525	$\Theta(n)$
1000	0.325629	0.000654095	$\Theta(n)$

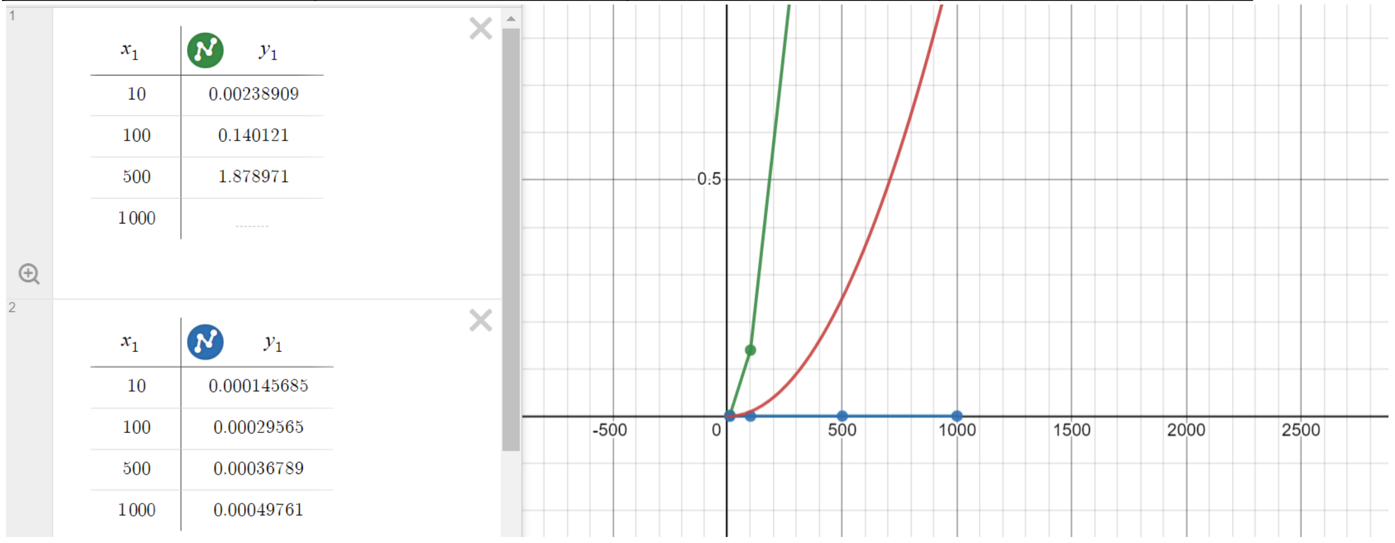


x_1	y_1
10	0.00552471
100	0.038927
500	0.16563
1000	0.325629

x_1	y_1
10	0.000145972
100	0.000304477
500	0.000423525
1000	0.000654095

Multiply

n	My function (ms)	Boost (ms)	Experimental
10	0.00238909	0.000145685	$\Theta(n^2)$
100	0.140121	0.00029565	$\Theta(n^2)$
500	1.878971	0.00036789	$\Theta(n^2)$
1000	D.N.F	0.00049761	$\Theta(n^2)$



1

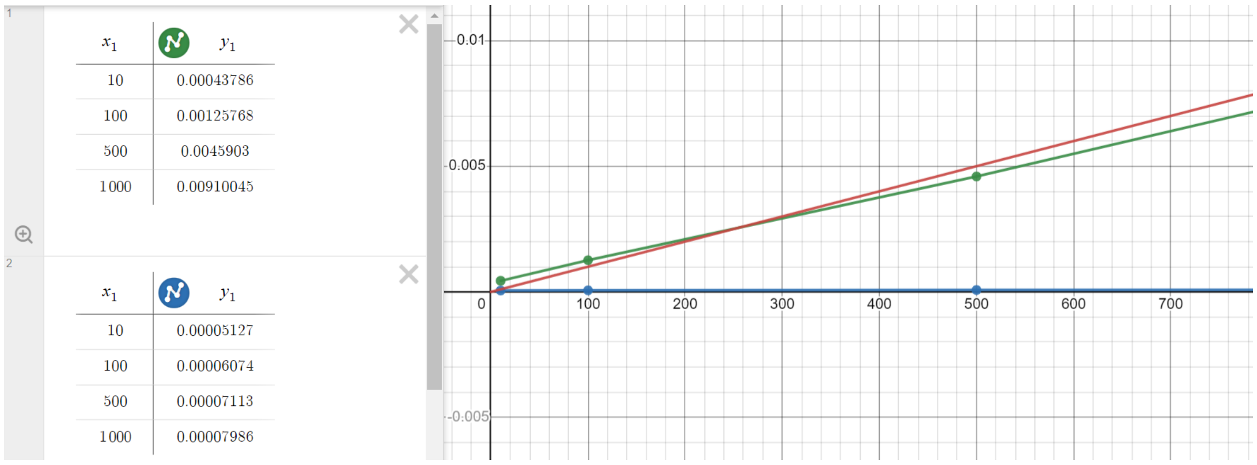
x_1	y_1
10	0.00238909
100	0.140121
500	1.878971
1000	

2

x_1	y_1
10	0.000145685
100	0.00029565
500	0.00036789
1000	0.00049761

compareTo

n	My function (ms)	Boost (ms)	Experimental
10	0.00043786	0.00005127	$\Theta(n)$
100	0.00125768	0.00006074	$\Theta(n)$
500	0.0045903	0.00007113	$\Theta(n)$
1000	0.00910045	0.00007986	$\Theta(n)$



x_1	y_1
10	0.00043786
100	0.00125768
500	0.0045903
1000	0.00910045

x_1	y_1
10	0.00005127
100	0.00006074
500	0.00007113
1000	0.00007986

Discussion of Results and Comparison

As can be seen from the data above, my theoretical time complexity was relatively close to the actual trend as seen in the performance of my implementation. CompareTo, Add, and Subtract are all relatively linear, but Multiply definitely follows a quadratic shape.

It is clear to see that my implementation is generally a lot worse when it comes to runtime than the Boost library. This is likely because the code that I created has not been completely optimized, and is lagging behind in speed when in comparison to boost. Some things in my code, like the creation of extra variables, or the copying of vectors was done to improve my understanding as a programmer, but could definitely have been left out if I was trying to completely prioritize run time. Most notably, the Karatsuba algorithm could have been implemented into the Multiply function in order to optimize run time, but I had a lot of trouble understanding the algorithm as is, and that made it very difficult to generalize, and implement into my code.