An analysis of Multiplication Functions -Programming 1-

By: Thomas Kwashnak

Emily Balboni

Priscilla Esteves

December 10th, 2021

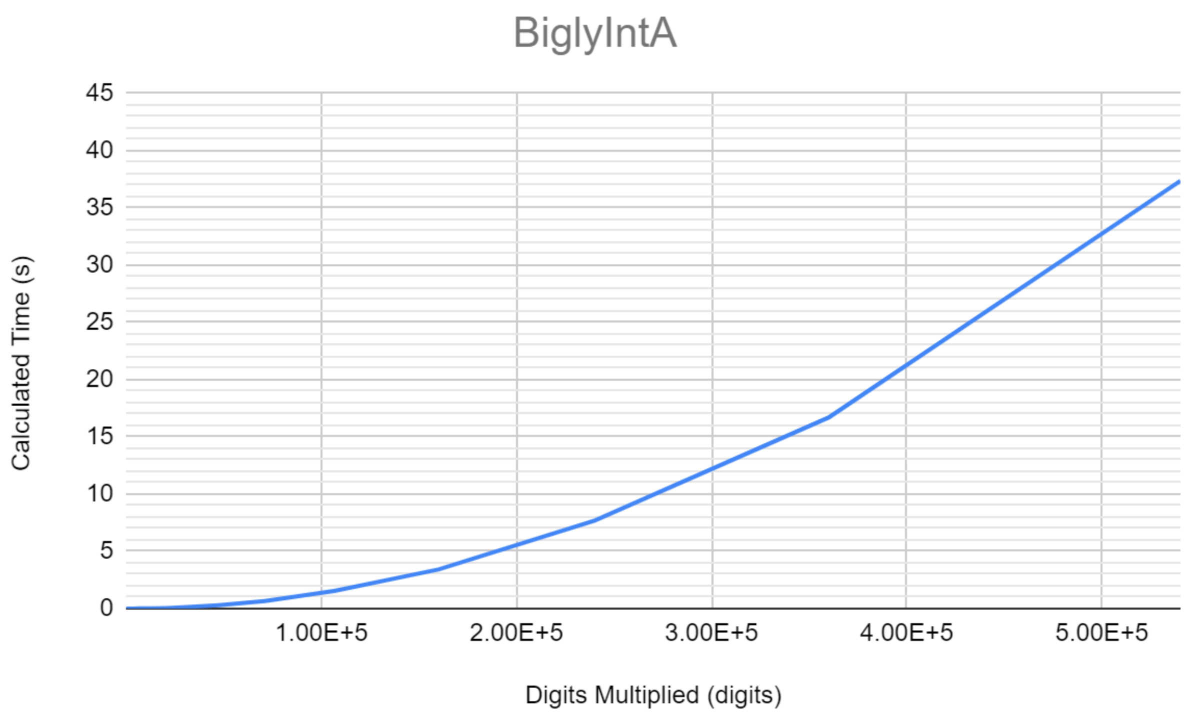
The Problem

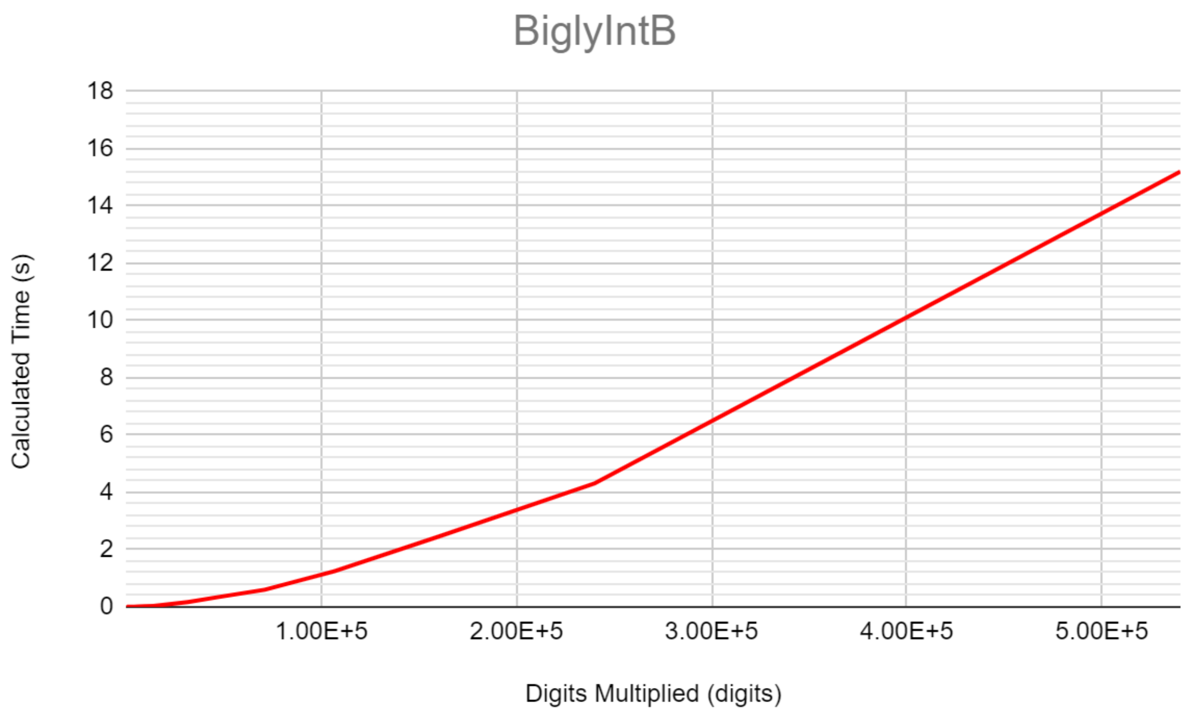
The traditional multiplication algorithm takes two different numbers and multiplies the first by the second number. In doing this, the run time of the algorithm is n multiplied by n, for n representing the length of each number. Therefore, the run time of the original multiplication algorithm takes Θ (n^2) time. But is this the most efficient method of multiplication? During the 1960s Russian mathematician Karatsuba derived a multiplication algorithm in Θ ($n^{\log_2 3}$) time. The algorithm splits the first number in half recursively which can be represented as a and b. The algorithm then splits the second number in half recursively that can be represented as c and d. Karatsuba then multiplies a and c, then a with d. Then, the technique multiplies the inner two numbers b and c. Lastly, the algorithm multiplies b and d. To complete the computation the algorithm adds the results together, maintaining decimal place, and returns the product. Karatsuba is intended to be faster than the original multiplication algorithm. However, is Karatsuba's technique more efficient than the original multiplication technique? Karatsuba's algorithm is also used in Java's BigInteger class yet before the BigInteger class completes the computation it switches to use a different, more efficient, algorithm. Therefore, the purpose of this experiment is to compare Karatsuba's algorithm with the original multiplication of technique.

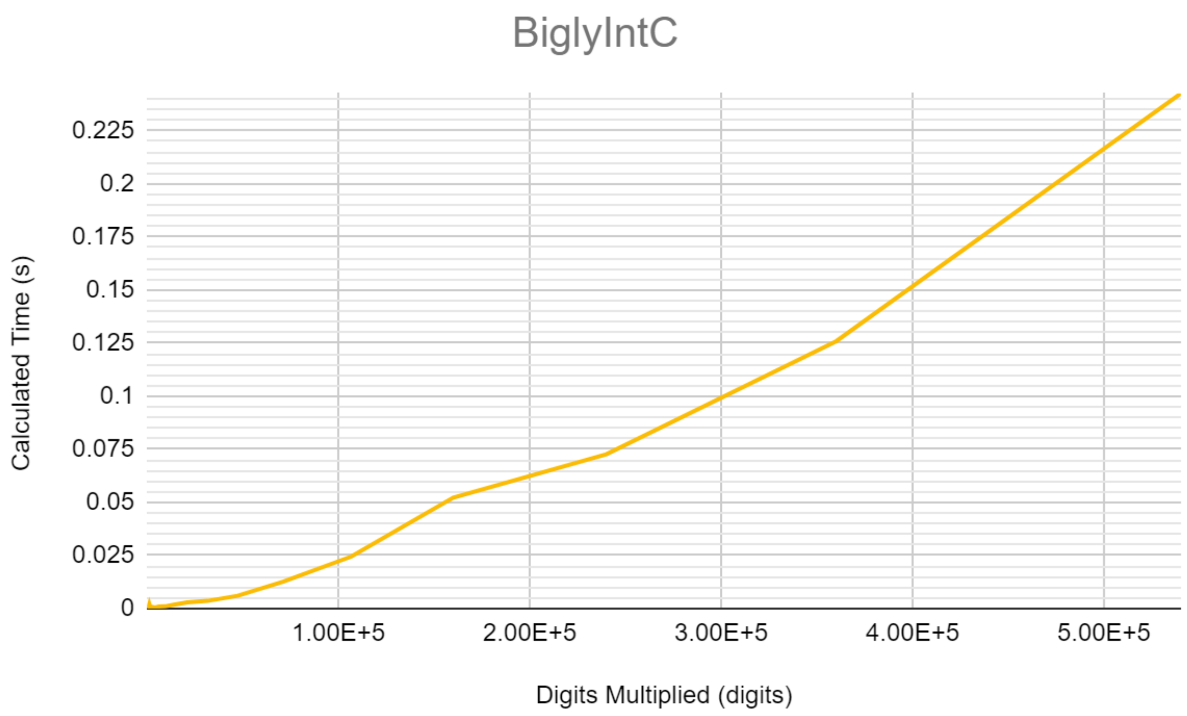
The Experiment

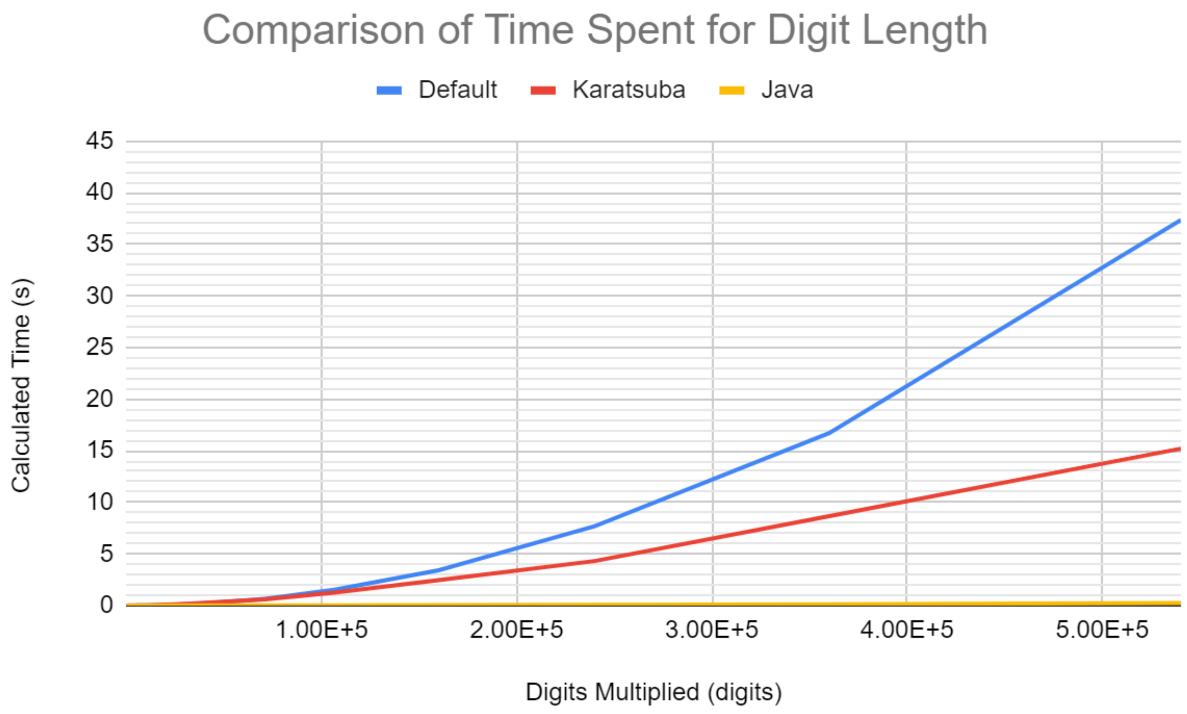
Before starting the experiment, our group created two randomly generated numbers to be multiplied together through using strings. To begin, we created fifteen strings of length ten and did not use a time limit. Instead of utilizing our instructors code, to save time generating numbers, we reused the strings from previous trials. We did this by adding random numbers to the end of the strings until the needed digit length. Each trial we increased the length by 50% until they reached 500,000 digits. To determine which strings out of the 15 created were multiplied with each other, we used an n choose n technique to avoid the same numbers being multiplied together for every trial. Although reusing previous strings has no effect on the test results, it prevented the need of repetitive string concatenations. Through doing this we were also able to use one hundred repetitions significantly faster than the original program. Statistically, using one hundred repetitions made our numbers more reliable. The multiplication functions investigated were BiglyIntA, BiglyIntB, and BiglyIntC. In each of these functions, at each trial the numbers were converted to the specific type and then were multiplied using either the regular multiplication technique implemented in BiglyIntA, Karatsuba implemented in BiglyIntB, or Java’s BigInteger class implemented in BiglyIntC. Our group decided to optimize this code as well to explore parallel computations. To do this, Thomas rewrote the experiment portion of the code to allow for multiple threads when converting the 15 strings to each BiglyInt type and performing the multiplication. The number of threads being dependent on the number of logical cores in the CPU. This empirical test was conducted on a laptop PC running Kubuntu 21.10 x86_64 operating system. The laptop ran on an Intel I7-1065G7 with “8 Cores” at 3.9GHz (ran at 2.2GHz during the experiment). The laptop had 16GB of RAM at speed 3200 Mhz.

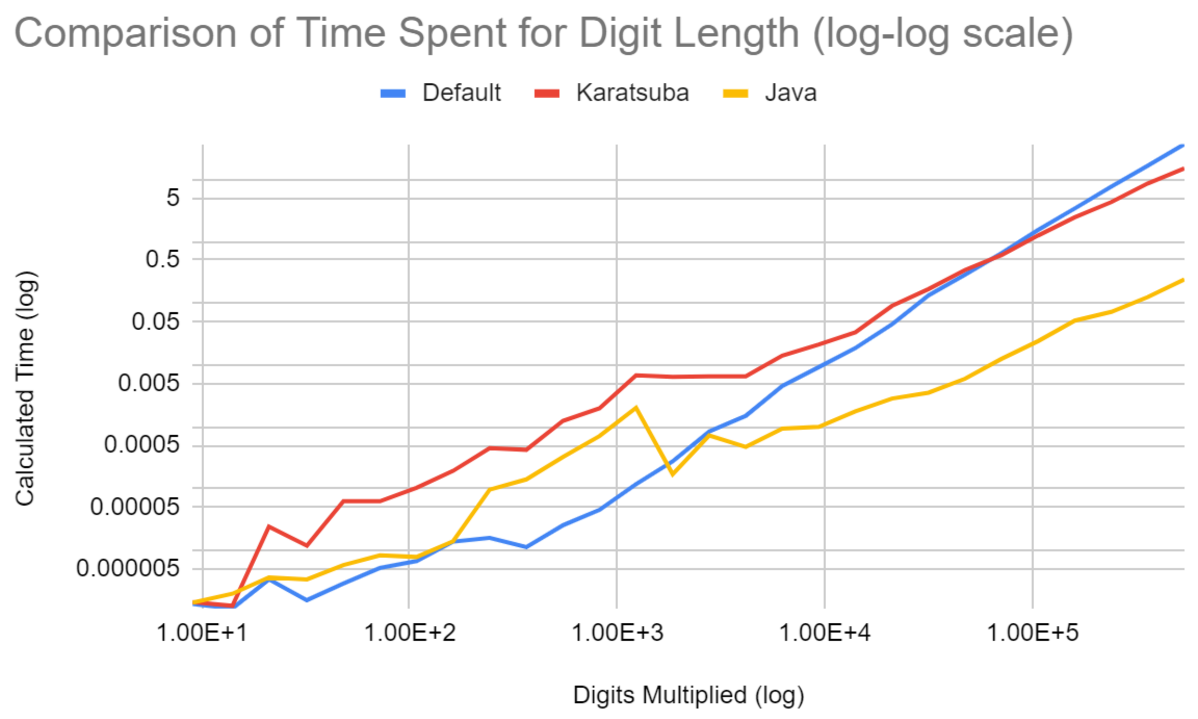
Graphs


BiglyIntA
Calculated Time (s)
45
40
35
30
25
20
15
10
5
0
1.00E+5
2.00E+5
3.00E+5
4.00E+5
5.00E+5
Digits Multiplied (digits)

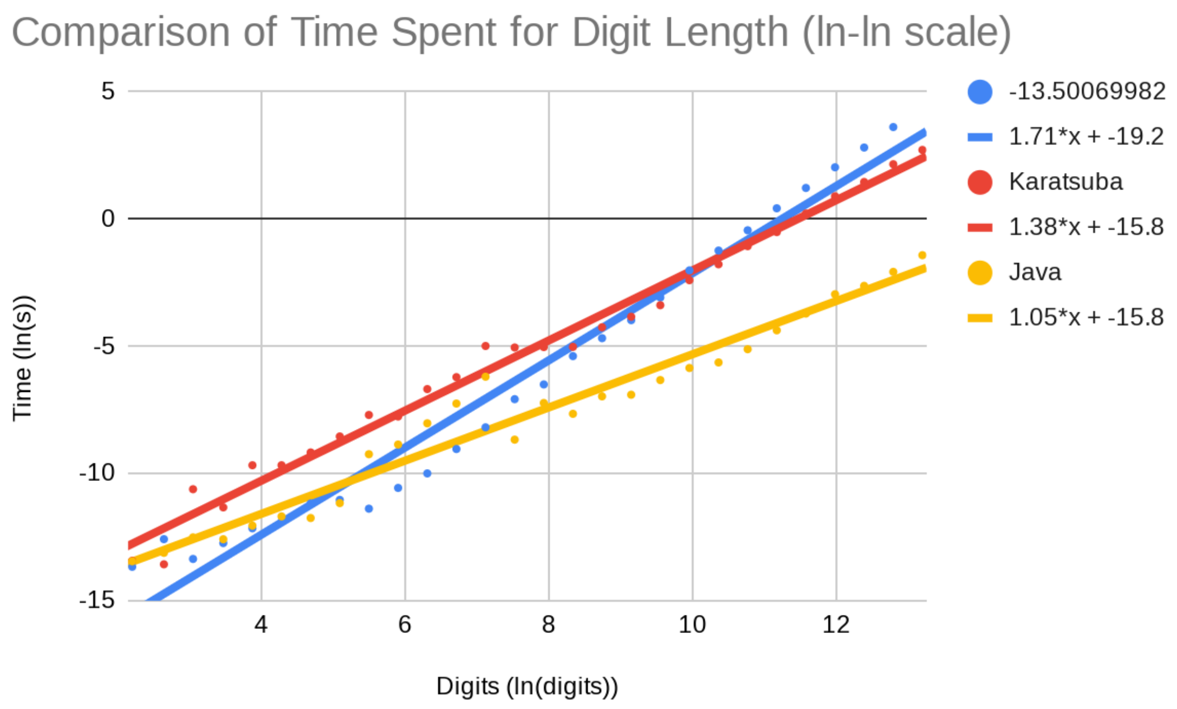

BiglyIntB
Calculated Time (s)
18
16
14
12
10
8
6
4
2
0
1.00E+5
2.00E+5
3.00E+5
4.00E+5
5.00E+5
Digits Multiplied (digits)


BiglyIntC
Calculated Time (s)
0.225
0.2
0.175
0.15
0.125
0.1
0.075
0.05
0.025
0
1.00E+5
2.00E+5
3.00E+5
4.00E+5
5.00E+5
Digits Multiplied (digits)


Comparison of Time Spent for Digit Length
Default
Karatsuba
Java
Calculated Time (s)
45
40
35
30
25
20
15
10
5
0
1.00E+5
2.00E+5
3.00E+5
4.00E+5
5.00E+5
Digits Multiplied (digits)



Included ln-ln Scale graph to provide a more accurate illustration


Comparison of Time Spent for Digit Length (ln-ln scale)
Time (ln(s))
Digits (ln(digits))
5
0
-5
-10
-15
4
6
8
10
12
-13.50069982
1.71*x + -19.2
Karatsuba
1.38*x + -15.8
Java
1.05*x + -15.8

Conclusion

After running the experiment and calculating the data, the method that performed the calculation in the best time was function BiglyIntC, which computed digits up to 500,000 in less than a second. Thus, showing that Java's implementation was the most efficient method. When presented with these results, our group looked through Java's BigInteger implementation. For the first 80 digits the traditional multiplication algorithm is used. Then from digits 80 to 240 Java switches to use Karatsuba algorithm. From 240 on Java then switches to implement the Toom-Cook-3 multiplication which computes at Θ ($n^{\log_3 5}$).

As illustrated above in the graph displaying the log-to-log scale there were a few significant time differences. About a quarter of the way between 1.00E+1 and 1.00E+2, Karatsuba takes a notable amount of time in comparison to Java's technique and the traditional algorithm. In addition, just after 1.00E+2, Java's technique and the traditional algorithm take almost the exact same time to calculate while Karatsuba continues to take significantly longer. Last, there was a significant difference in the time it took each algorithm at the digits leading up to 1.00E + 4 until the end of the experiment. There, it is clearly displayed that Java's BigInteger class is computing multiplication faster than the traditional multiplication algorithm and Karatsuba's algorithm.