SUMMARY AND GENERAL OBSERVATIONS

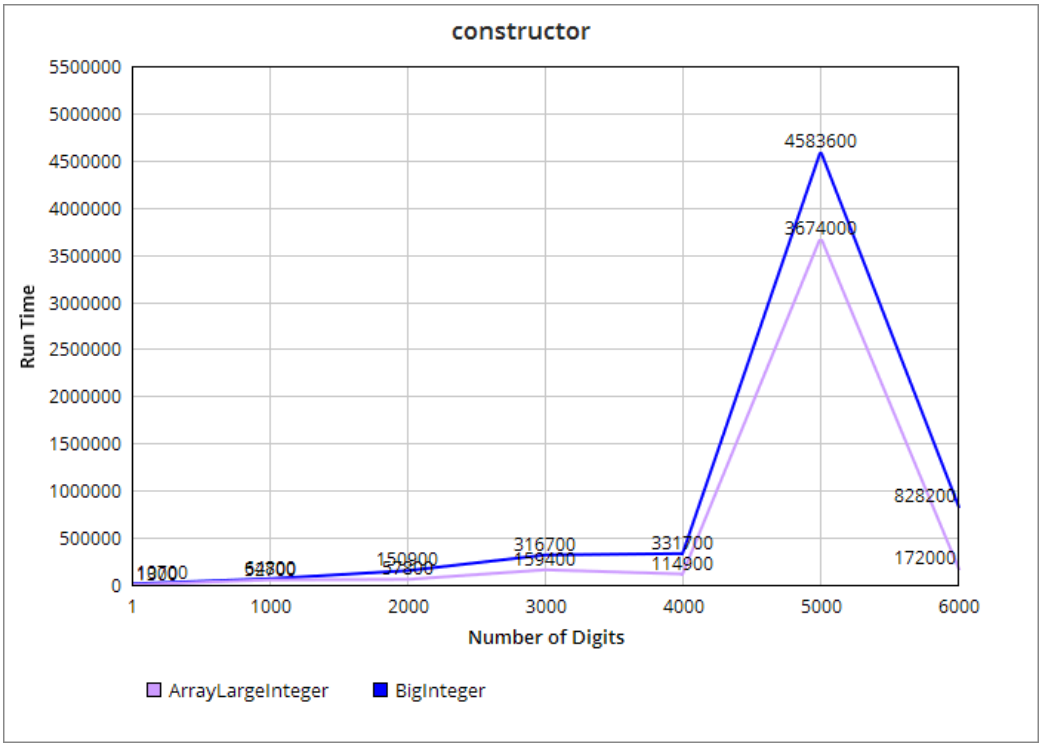
Overall, in most scenarios BigInteger (BI) was able to visibly outperform ArrayLargeInteger (ALI) in terms of nanoseconds elapsed. .multiply() is the most dramatic example of this. However, this was not *always* the case. For methods in ALI that have relatively constant big-oh complexity, ALI performs equally or better than BI, an example of this can be seen with . .toString(). The reason that this is possible is because ALI is able to use the original String parameter from the constructor, whereas I believe BI's implementation creates a new String representation. According to my interpretation of the LargeInteger interface's specifications, none of the provided methods should modify the object they are called on, thus the original string should work perfectly to return.

It seems that methods that depend heavily on the length of the given number and/or call the constructor once/more than once, as .multiply() does, have the highest disparity between ArrayLargeInteger and BigInteger.

COMMENTARY ON EACH METHOD

ArrayLargeInteger()

Below is a graph of ALI and BI's constructor's nanoseconds for numbers of various sizes: 1 digit, 1,000 digits, 2,000 digits, 3,000 digits, 4,000 digits, 5,000 digits, and 6,000 digits.



Big-oh Complexity Worst case $O(n^2)$, Average case $O(n)$:

At worst (when we have to trim a bunch of leading zeroes) new ArrayLargeInteger is

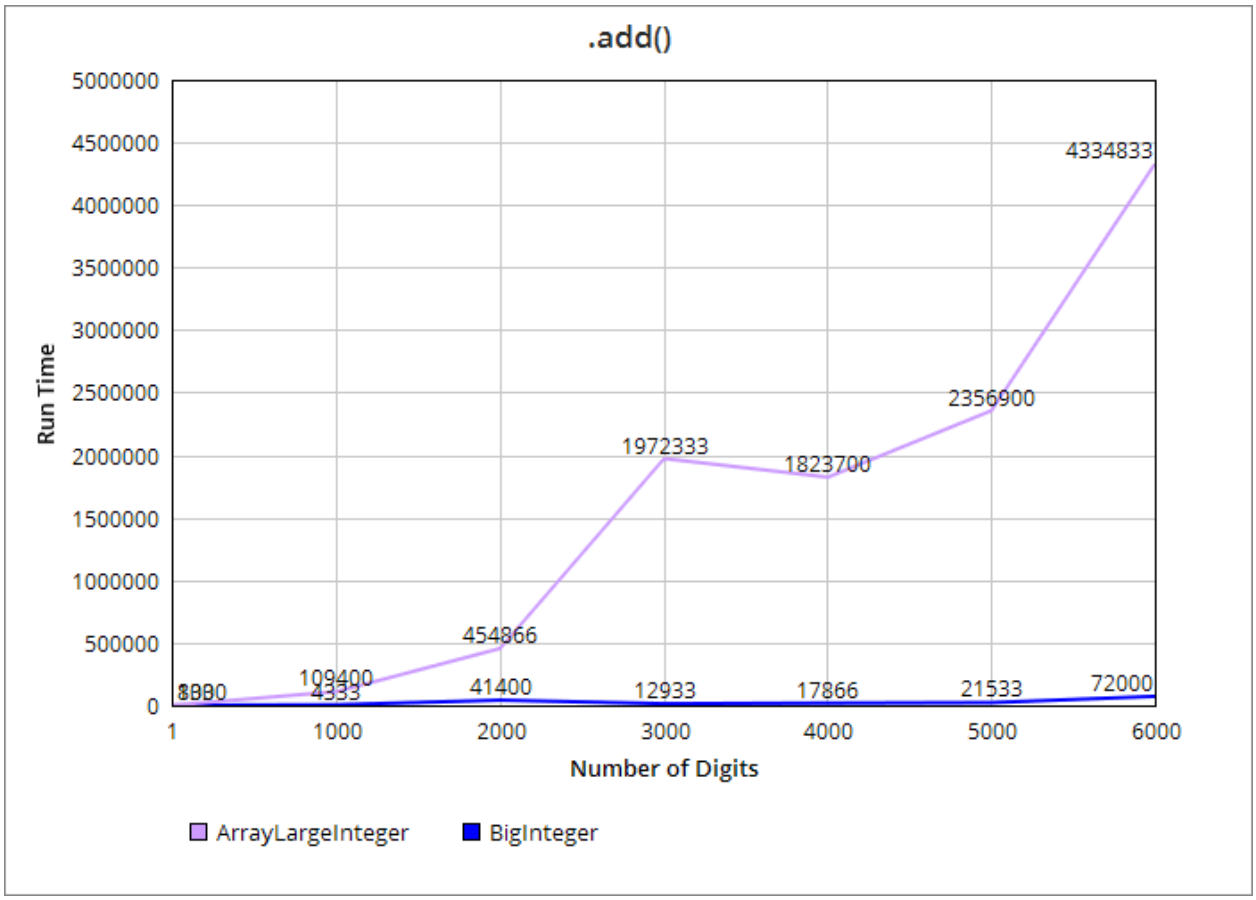
- Before trim zeroes (worst case: it passes every if statement): = O(n)
    - If (equals "-") substring = O(1) because the length of that substring is 1, and equals is O(n)
        - Substring = n
    - Equals = n
    - Substring = n
- Trim zeroes (worst case, many zeroes): = n*n = $O(n^2)$
    - n loop for tons of zeroes = n
        - n substring that many times (would decrease n each time) = n
- Re-adding the negative = 2n = O(n)
    - If (Equals) =O(n)
        - Concatenation = O(n)
- After trim, saving the digits = n(1) + 2n (worst case) = 3n = O(n)
    - While loop with length (n) iterations = n
        - Substring = O(1), because its of size 1
        - Equals = O(1), because its of size 1
        - parseInt = O(1)
    - If (equals zero) = O(n)

In the worst case scenario, which is extremely unlikely to happen, this method is $O(n^2)$. This is extremely unlikely because the chance that a number passed in has almost as many leading zeroes as it does digits is very slim, but it still represents the worst case big-oh at $O(n^2)$. When there are few, if any, zeroes to be trimmed, which would be the average case, the time complexity is O(n), because all of the statements inside of the while loop are O(1), due to the fact that they (they being substring and equals specifically) are of length 1.

Of course, the numbers used for the graph most likely do not follow the worst case scenario, and rather represent something much more average. Although I do not know what caused n = 5,000 to take so much runtime for both ALI and BI. Additionally, it is important to mention that any methods that call the constructor again will most likely *not* be the worst case scenario because it is very unlikely that the object used to call the method which is now calling the constructor, still has many, if any, leading zeroes.

.add()

The ArrayLargeInteger solution to add large numbers is, of course, less efficient than BigInteger's implementation. At larger number sizes, such as numbers with 6,000 digits, ALI performs much worse than BI. Below is a graph of ALI and BI's .add()'s time in nanoseconds for numbers of various sizes: 1 digit, 1,000 digits, 2,000 digits, 3,000 digits, 4,000 digits, 5,000 digits and 6,000 digits.



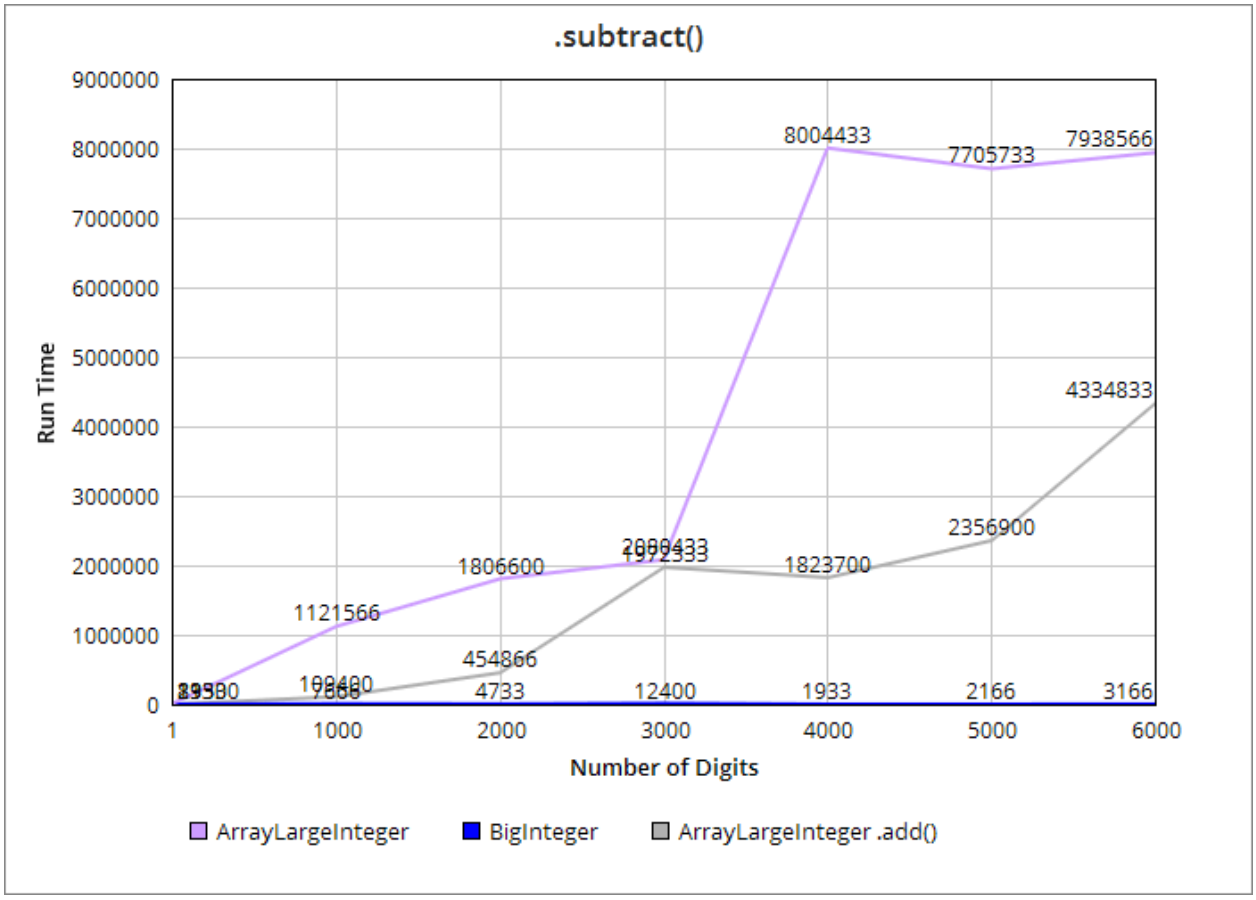
Big-oh complexity, $O(n^2)$:

- calls string.equals() method twice O(n)
- At some point returns a newly constructed ArrayLargeInteger Average O(n), Worst Case, O(n2)
- Matching signs (n, where n = longer number's size) = O(n) * O(n) = $O(n^2)$
    - While loop dependant on n, where n is the number of digits in the larger of the two given numbers
    - Concatenation O(n) (the actual complexity of the method depends on the length that the string currently is, so this would not be O(n) for every iteration of the while loop, but it will be by the end, which is what matters the most)
- OR not matching signs
    - Negate = Average O(n), Worst Case, O(n2)
    - Subtract

For .add() calls where the numbers have matching signs, the method continues to actually do the addition, the method makes two calls to String.equals() which are O(n) each, and then runs through a while loop that has a number of iterations equal to the number of digits in the larger number. This loop, the String concatenation in it, and the call to the constructor are what dictate this method's big-oh. In any case, the time complexity will be $O(n^2)$.

.subtract()

.subtract() is much more time intensive than .add() for the ALI implementation. This is most likely due to the need to call max, when continuing on to perform the subtraction, or the need to call .abs() or .negate() when rewriting the equation into an addition problem. This is most likely especially the case for when two negative numbers are meant to be subtracted because .subtract() will invoke .add() on those numbers, which then invokes .subtract() on those numbers in a way that allows the method to go through with the subtraction now that the equation has been rewritten. Below is a graph of ALI and BI's .subtract()'s time in nanoseconds for numbers of various sizes: 1 digit, 1,000 digits, 2,000 digits, 3,000 digits, 4,000 digits, 5,000 digits and 6,000 digits. The data for .add() has been included as a point of reference for the time difference between the two



Big-oh Complexity $O(n^2)$:

- For matching sign
  - Calls max = O(n)
    - Worst case $O(n^2)$, where you have to look through all of the digits, and due to the constructor's worst case
    - Best case O(1), where they are opposite signs

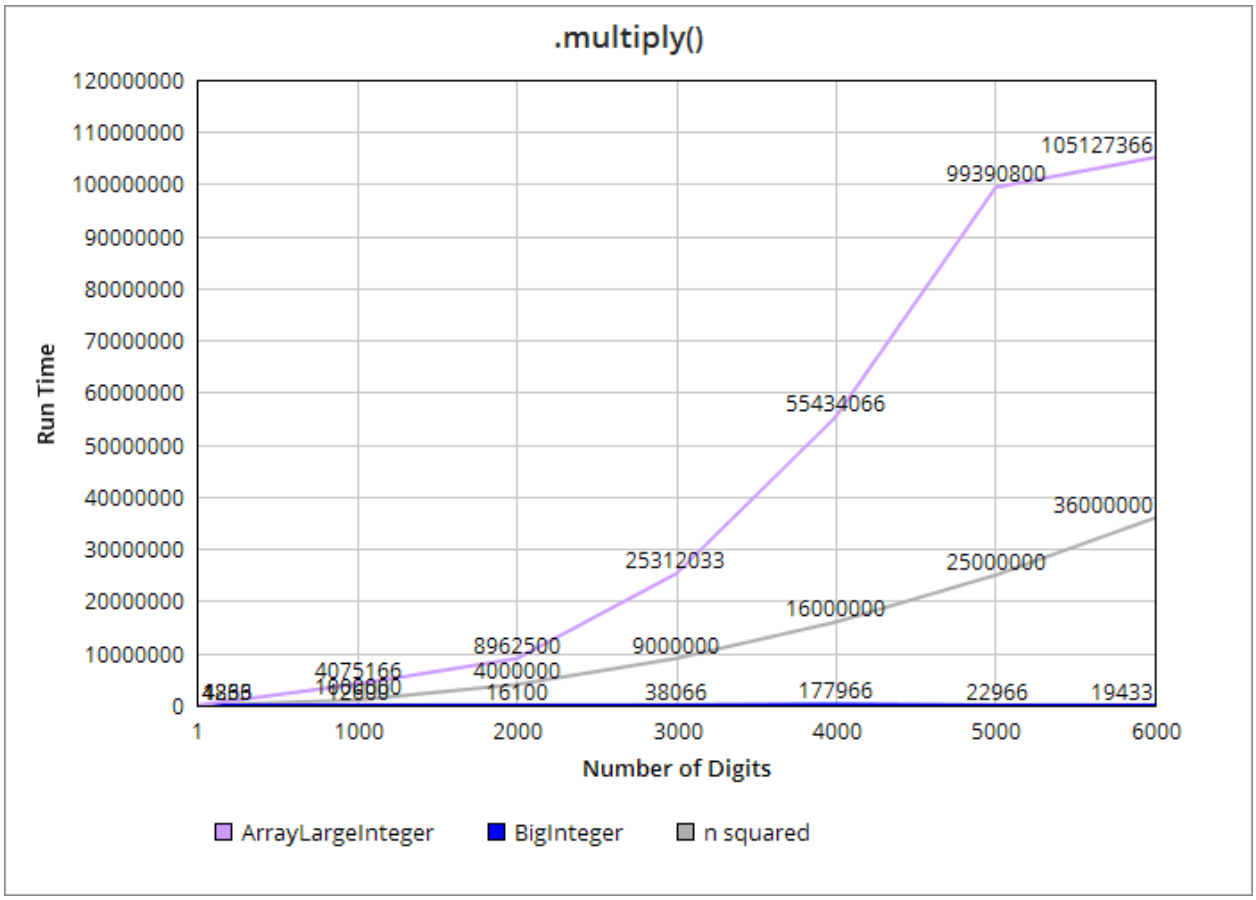
    - Calls min = O(n)
        - Worst case $O(n^2)$, where you have to look through all of the digits, and due to the constructor's worst case
    - The while loop is dependent on the size of the larger number O(n)
        - Concatenation O(n) (the actual complexity of the method depends on the length that the string currently is, so this would not be O(n) for every iteration of the while loop, but it will be by the end, which is what matters the most)
    - Make the new ArrayLargeInteger, Average O(n), Worst Case, O(n2)
    - Potential to have to negate, Average O(n), Worst Case, O(n2)
- OR Non matching signs
    - Calls add and abs
        - Abs calls ArrayLargeInteger, Average O(n), Worst Case, O(n2)
            - Potential to call substring, O(n)
        - .add() $O(n^2)$
    - OR Calls add and negate
        - Negate calls ArrayLargeInteger, Average O(n), Worst Case, O(n2)
            - Potential to call substring, O(n)
        - .add() $O(n^2)$

For .subtract() calls that actually go through with the subtraction, when the numbers are both positive, the method calls .max() and .min() which are (on average) O(n) each, and then runs through a while loop that has a number of iterations equal to the number of digits in the larger number. This loop, the concatenation within, and the call to the constructor are what dictate this method's big-oh. It is $O(n^2)$ in both the average and worst case.

When both numbers are not positive, .subtract() will call .add() and either .abs() or .negate(), if this is the case .subtract()'s complexity will be the same as .add()'s. The worst case of this is when the two given numbers are both negative, because the invocation of .add() will reinvoke .subtract(), which will then proceed to do the subtraction, whose time complexity is discussed in the paragraph above this, even in this case the time complexity still falls within $O(n^2)$.

.multiply()

At very small sizes of numbers, the difference in time elapsed is already noticeable. However at larger number sizes, ALI performs *much worse* than BI's implementation. This is most likely because of the number of times that ALI's constructor is invoked during its implementation of .multiply(), which is equal to the number of digits in the second term, and because of the utilization of nested loops, which are equal to the size of the two terms. Below is a graph of ALI .multiply()'s time in nanoseconds for numbers of various sizes. A plot of n^2 has been included as a point of reference.



Big-oh Complexity $O(n^3)$:

- Calls String.equals twice O(n)
- Calls constructor, Average O(n), Worst Case, $O(n^2)$
- Outer while loop, dependent on second term's size O(n) * $O(n^2)$ ← from the inner loop, or constructor, or .add() = $O(n^3)$
    - Inner while loop, dependent on first term's size O(n) * O(n) = $O(n^2)$
        - Concatenation is O(n)
    - Calls ArrayLargeInteger[1], Average O(n), Worst Case, $O(n^2)$
        - Potential to call substring, O(n)
    - Calls .add()[1] $O(n^2)$
- Potential to have to negate O(n)

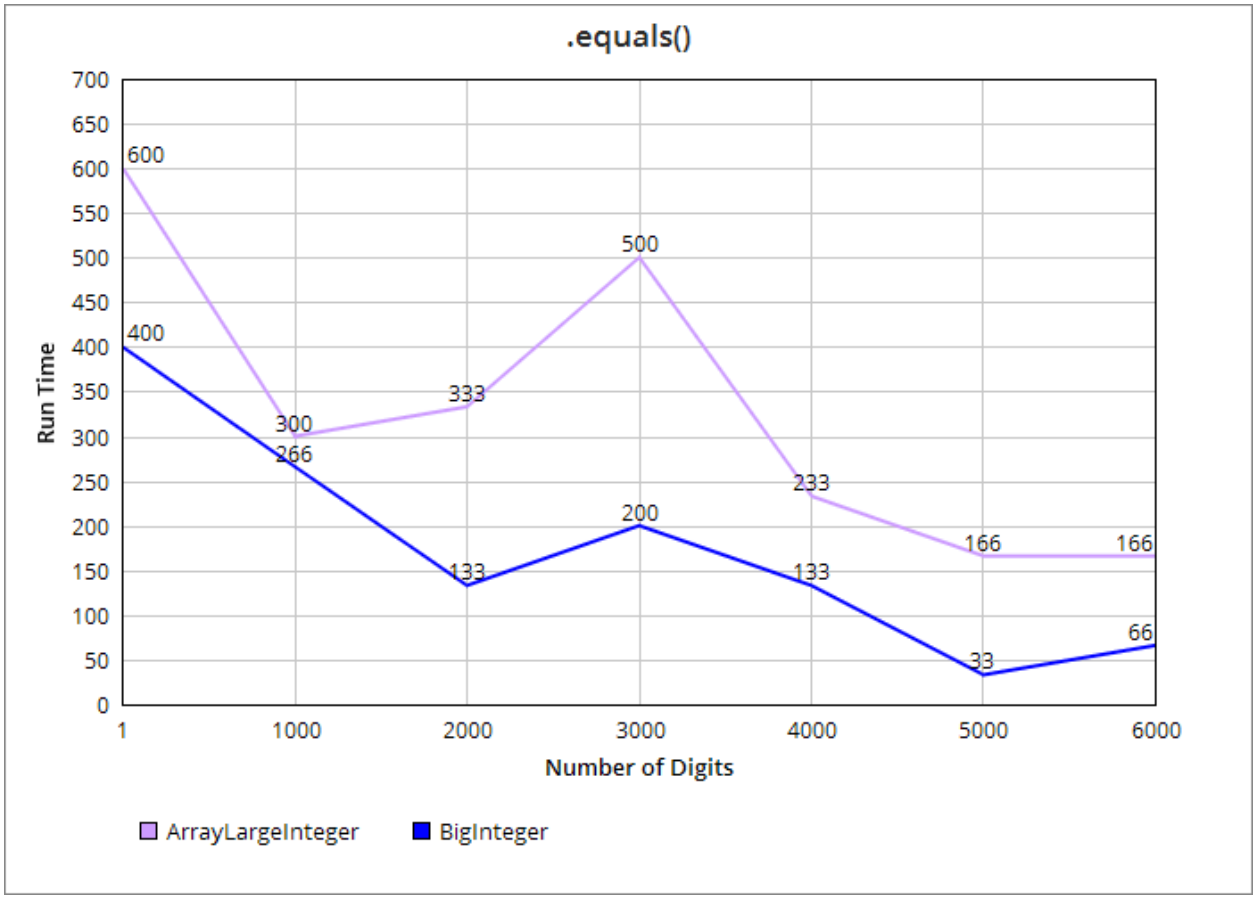
Multiply gets particularly large because of the nested loops and the repeated concatenation and calls to the constructor and to .add(). The outer loop iterates n times, executing the inner loop m times each time, where m = the number of digits in the first term[2], and calling the constructor and

[1] This is why .multiply() is so bad
[2] I'll assume they're equal size for convenience

.add() once each iteration of the outer loop. Making the time complexity of .multiply() $O(n^3)$ in the worst case.

.equals()

ArrayLargeInteger's implementation of .equals() performs similarly to BigInteger's implementation at both small sizes and large sizes. It is interesting to note the downwards trend of both ALI and BI on the graph. Additionally note that the range of runtime is comparatively small, between 166 and 600 for ALI and 33 and 400 for BI. Below is a graph of ALI and BI's .equals()'s time in nanoseconds for numbers of various sizes[3].



Big-oh Complexity O(n):

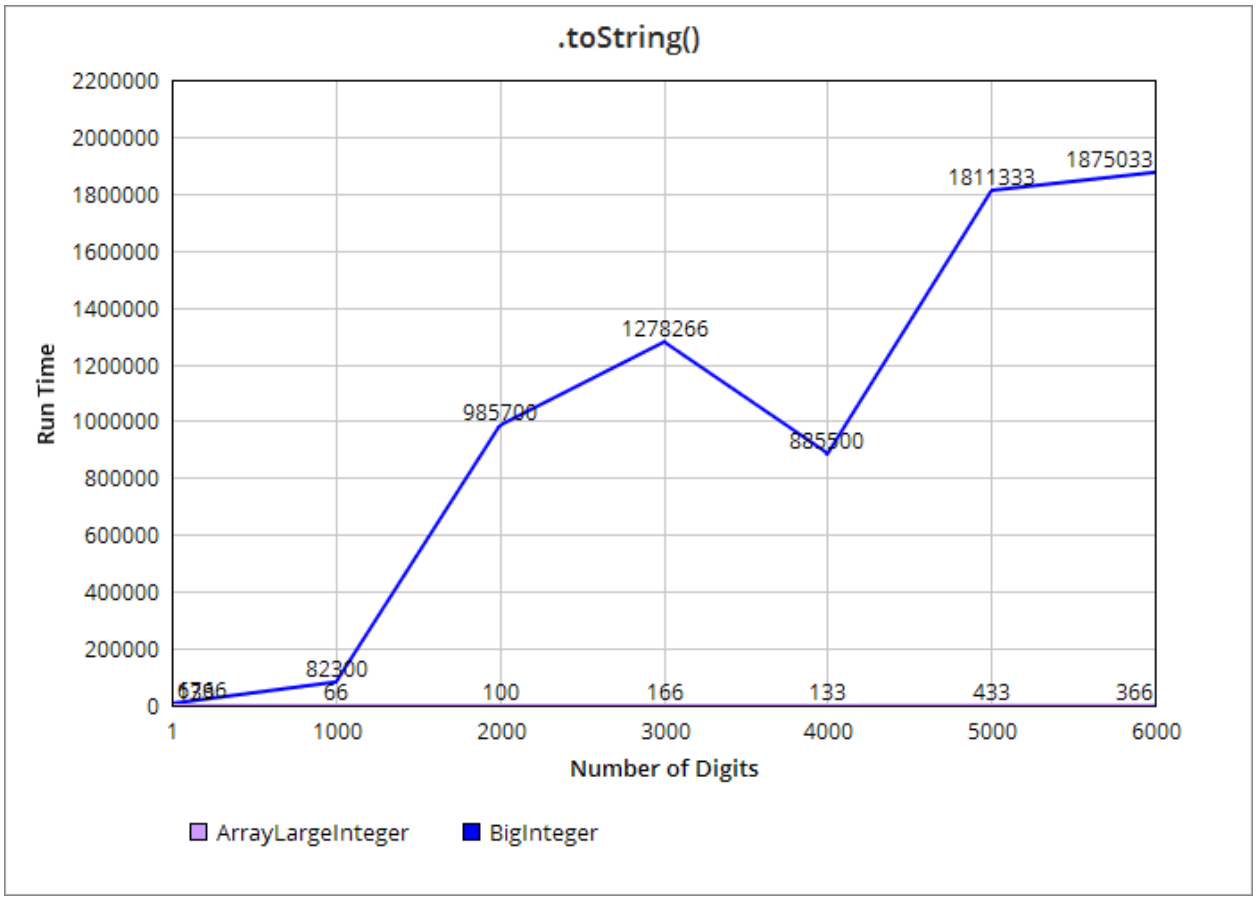
- Calls ArrayLargeInteger.toString() twice, but this is just O(1)
- Then uses String.equals once O(n)

According to the analysis of the method, the ArrayLargeInteger implementation of .equals() should be O(n), however, this does not seem to be the case, as both ALI and BI .equals()'s runtime generally trend downwards as the size of n increases. I do not know why this is the case.

[3] Why is it so common that the first call takes so much longer, even though it's a smaller number?

.toString()

ArrayLargeInteger is able to use the String that was originally given in the constructor, because none of the ALI methods allow the original number to be modified. Due to this fact, ALI's .toString() method outperforms BigInteger's. ALI's runtime stays generally constant, despite the increasing size of n, whereas BI's does not. Below is a graph of ALI and BI's .toString()'s time in nanoseconds for numbers of various sizes: 1 digit, 1,000 digits, 2,000 digits, 3,000 digits, 4,000 digits, 5,000 digits, and 6,000 digits.

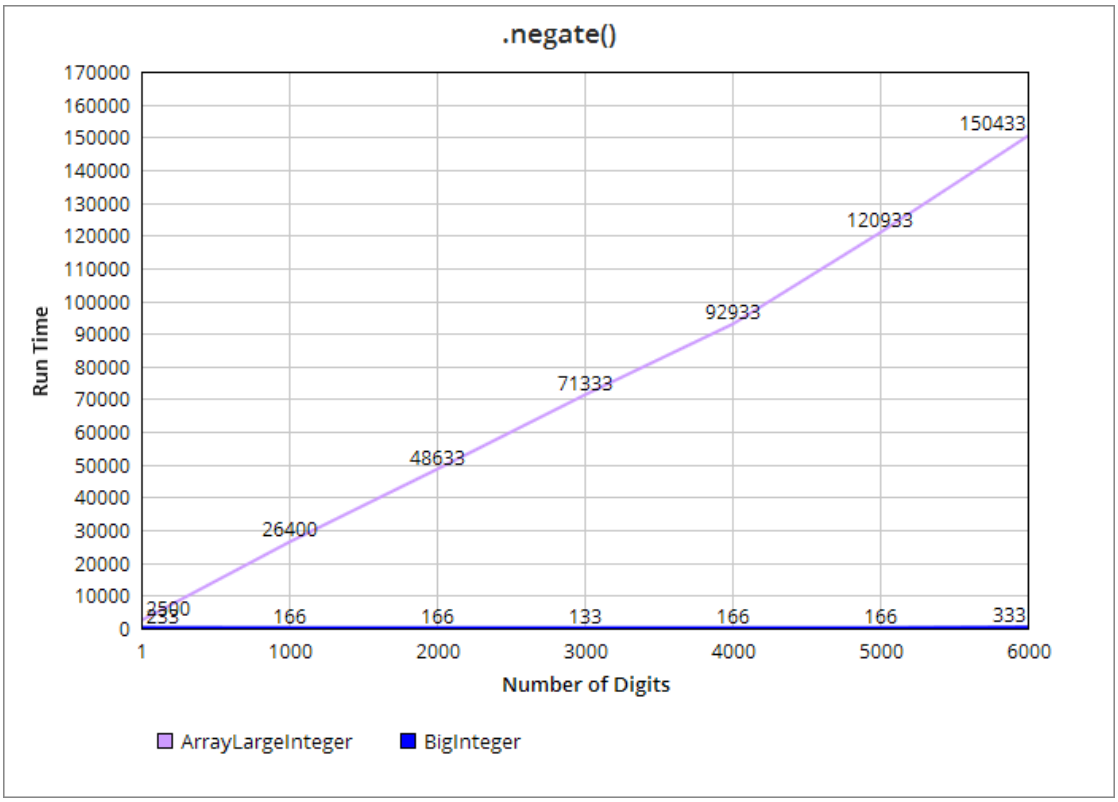


Big-oh Complexity O(1):

This method is O(1). The time complexity is constant, because it simply returns a pre-existing instance variable. The graph of ArrayLargeInteger's implementation agrees with this statement.

.negate()

ArrayLargeInteger takes noticeably longer to negate a given number than BigInteger does. This is most likely due to the fact that ALI's implementation creates a new ArrayLargeInteger that is the negation of the object used to call the method rather than simply returning the object negated. Below is a graph of ALI and BI's .negate()'s time in nanoseconds for numbers of various sizes: 1 digit, 1,000 digits, 2,000 digits, 3,000 digits, 4,000 digits, 5,000 digits, and 6,000 digits



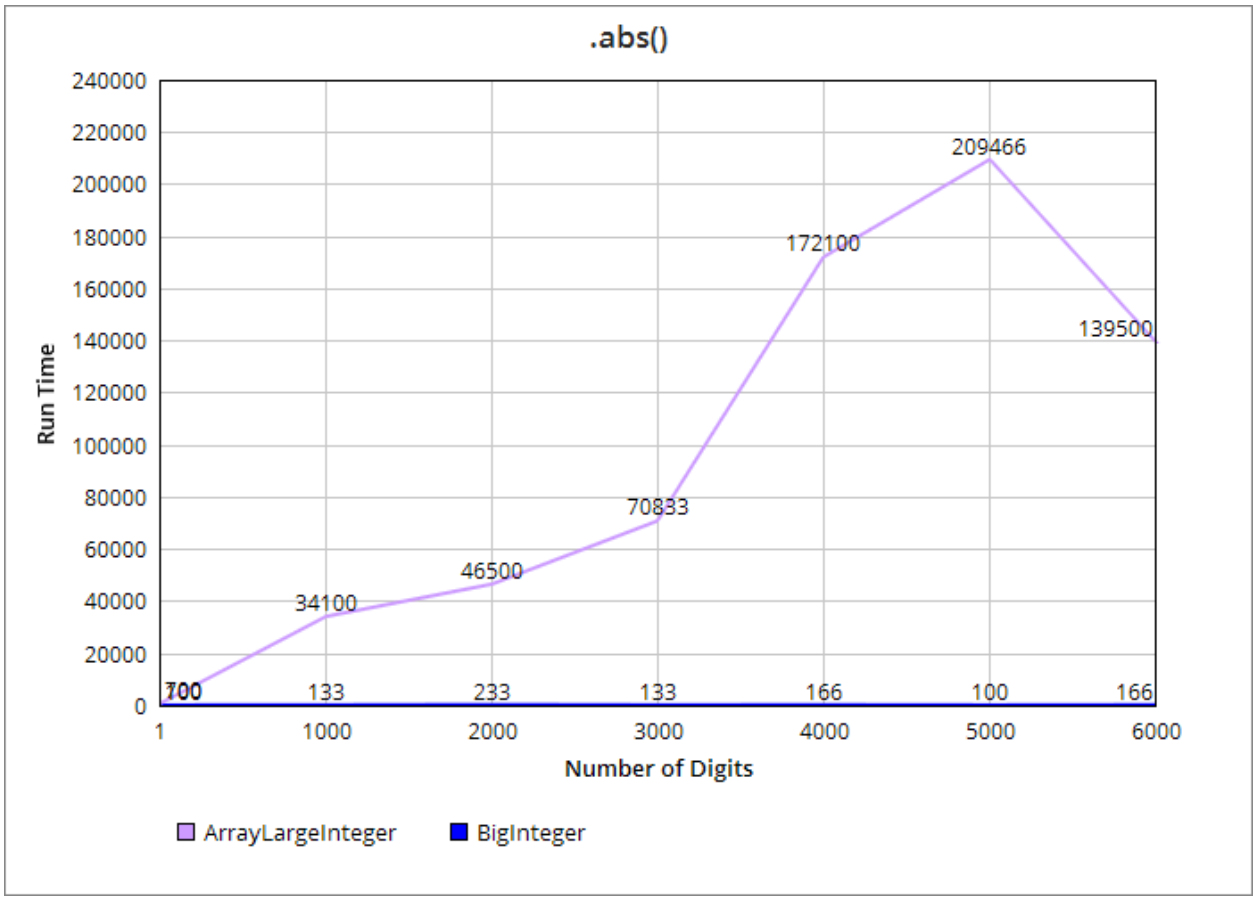
Big-oh Complexity Average O(n), Worst Case, O(n^2):

- calls substring O(n) OR concatenates O(n)
- Calls ArrayLargeInteger Worst case O(n^2), average case O(n)

Due to the conditional statement, one way or the other, the method will execute a statement that is O(n), and then call the constructor. The call to new ArrayLargeInteger() is the main thing of note. Due to how linear ALI's graph seems to be, this lends some credibility to the notion that in an average case ALI's constructor is some type of O(n). It is reasonable to state that .negate() has a similar time complexity to the constructor.

.abs()

ArrayLargeInteger takes noticeably longer to provide the absolute value of a given number than BigInteger does. This is most likely due to the fact that ALI's implementation creates a new ArrayLargeInteger that is the absolute value of the object used to call the method rather than simply returning the object as a positive number. Below is a graph of ALI and BI's .abs()'s time in nanoseconds for numbers of various sizes: 1 digit, 1,000 digits, 2,000 digits, 3,000 digits, 4,000 digits, 5,000 digits, and 6,000 digits.



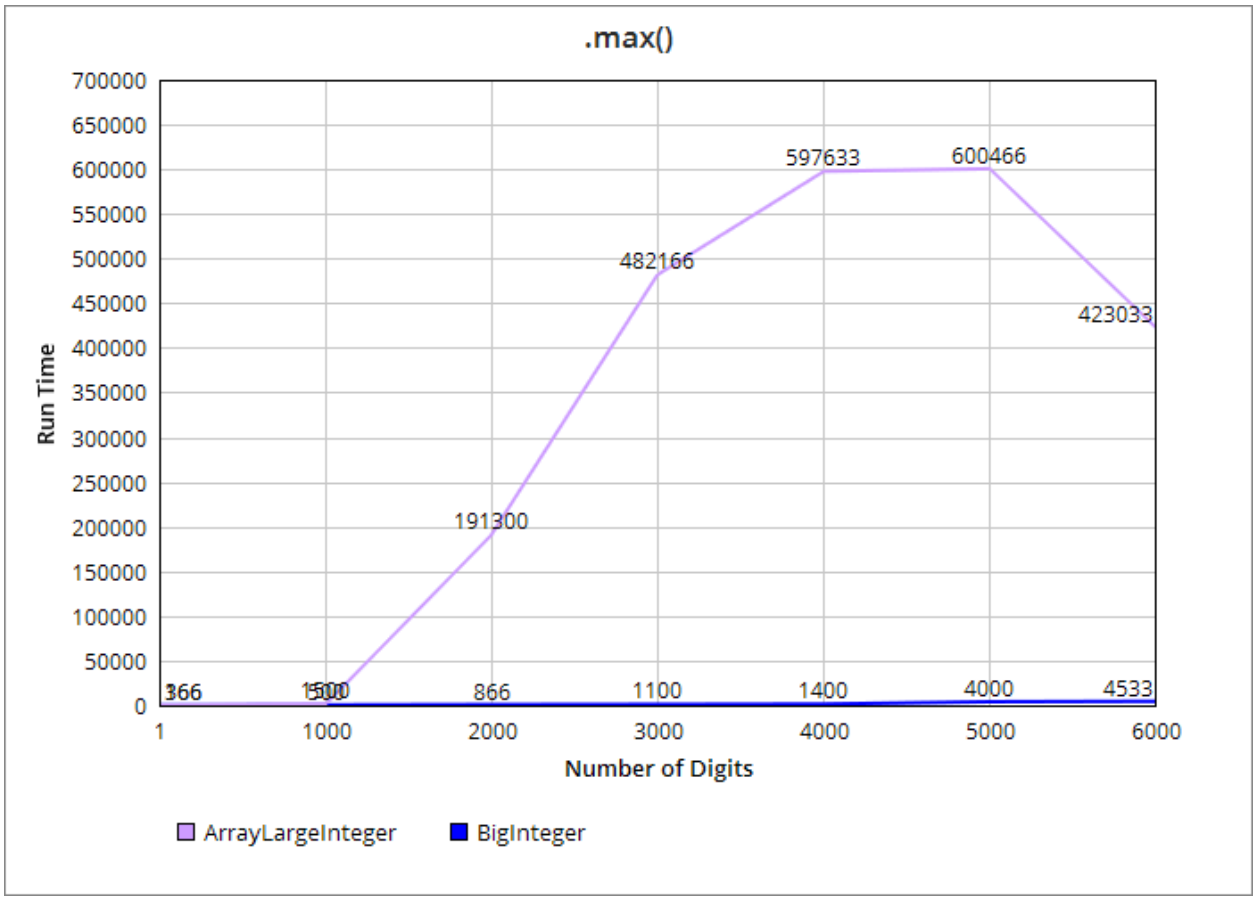
Big-oh Complexity Average O(n), Worst Case, $O(n^2)$:

- Could call substring O(n)
- Calls ArrayLargeInteger constructor Worst case $O(n^2)$, average case O(n)

At worst case, when this number is negative, String.substring() must be called, otherwise the only thing of note is the call to new ArrayLargeInteger(). Overall it could be reasonable to state that abs() has a similar time complexity to the constructor, since it is the largest statement of note in the method.

.max()

Below is a graph of ALI and BI's .max()'s time in nanoseconds for numbers of various sizes: 1 digit, 1,000 digits, 2,000 digits, 3,000 digits, 4,000 digits, 5,000 digits, and 6,000 digits. *It is imperative to note that this graph shows the near worst case for ALI,* because these results were produced by calling the method to compare numbers that were exactly the same, and no initial check was performed to see if the numbers were equal. This was done to emulate the worst case where the the terms evaluated were near identical, and only varied towards the end, in the ones or tens place for example.



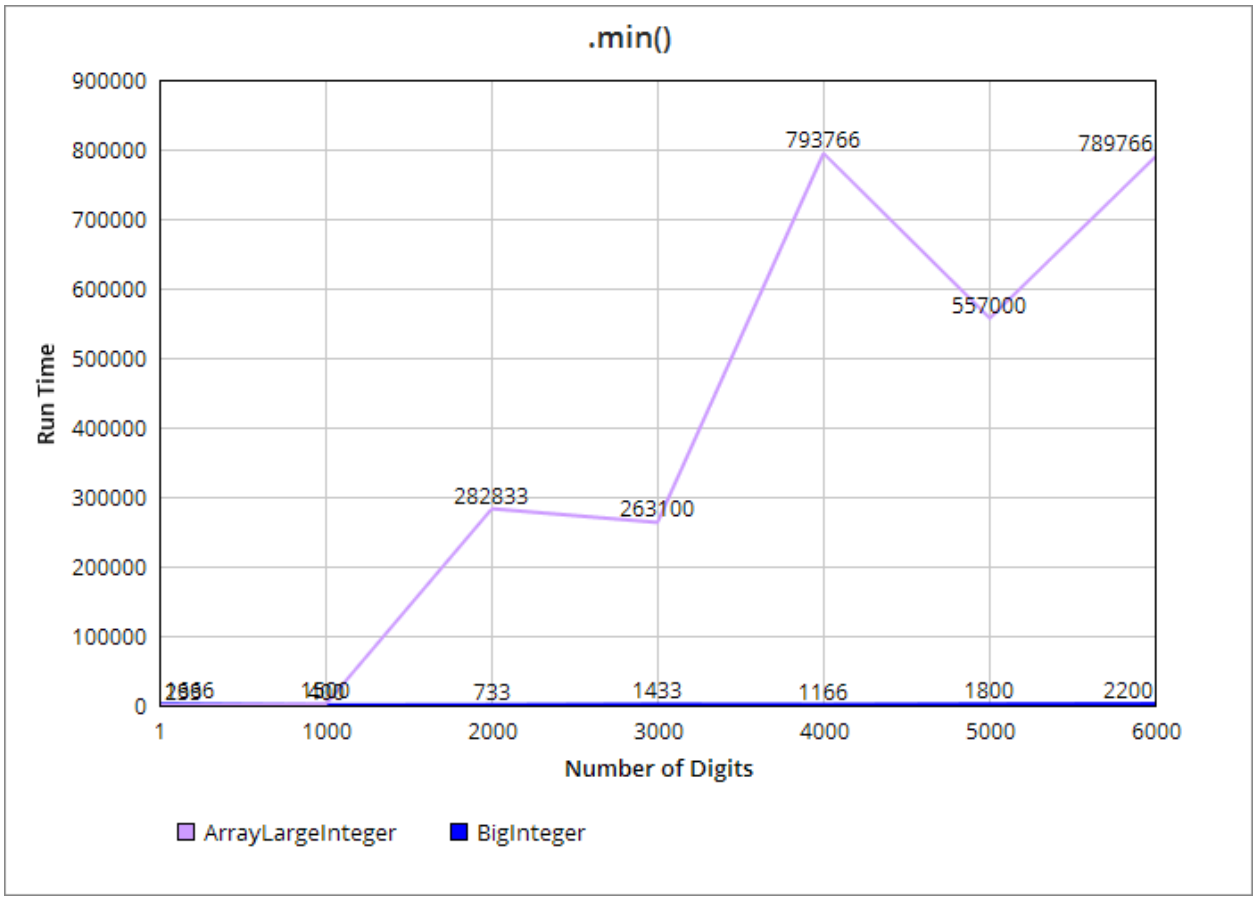
Big-oh Complexity Average O(n), Worst Case, O(n2):

- Call new ArrayLargeInteger, Average O(n), Worst Case, O(n2)
- Worst case: will have to iterate through all of the digits. O(n)
    - All statements in the for loop are O(1)

The worst case occurs when .max() must iterate through all of the digits in the given numbers. If this is the case, the method itself has a complexity of O(n) + the complexity of the constructor.

.min()

Much like was the case for max, the below graph of ALI and BI's .min()'s time in nanoseconds *represents the near worst case for ALI,* because these results were produced by calling the method to compare numbers that were exactly the same, and no initial check was performed to see if the numbers were equal. This was done to emulate the worst case where the the terms evaluated were near identical, and only varied towards the end, in the ones or tens place for example.



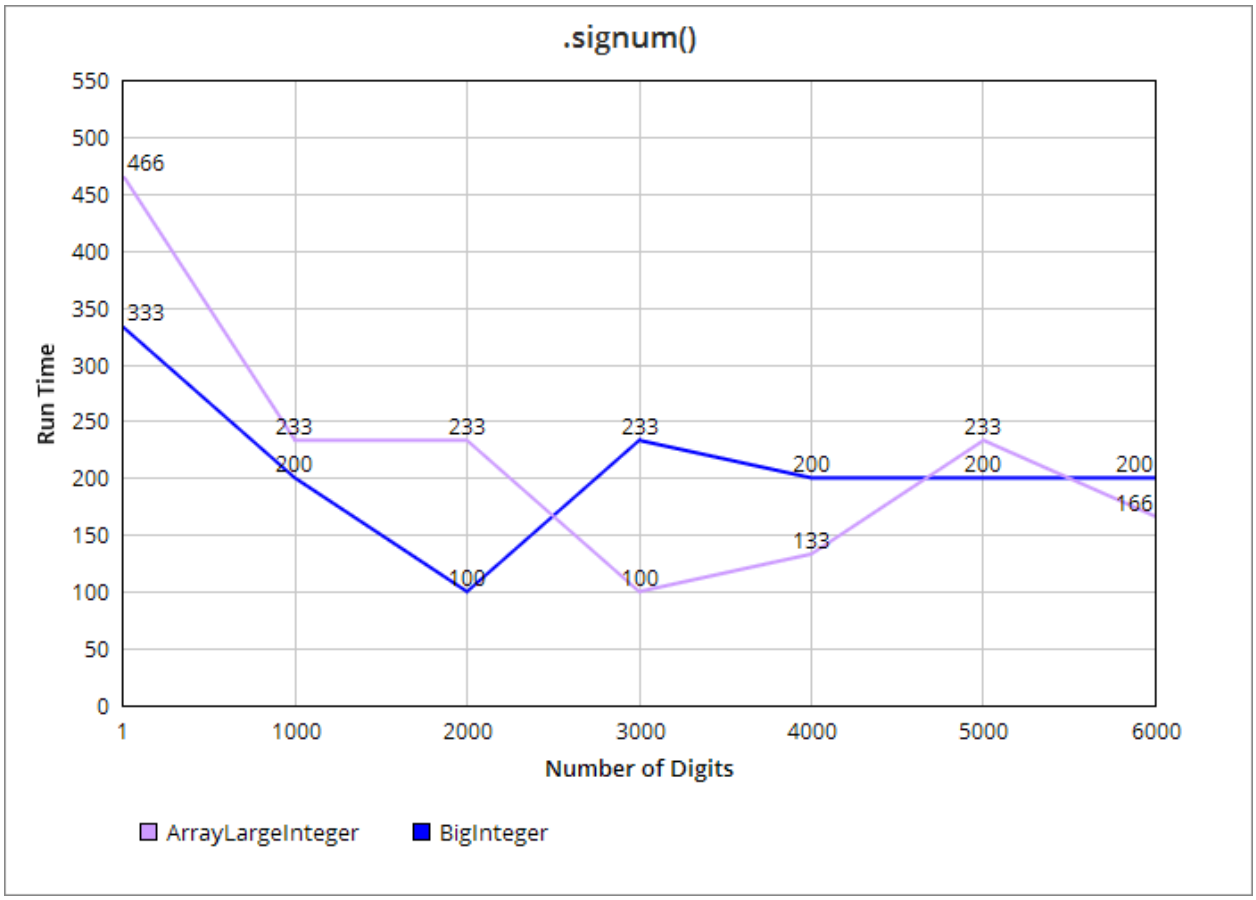
Big-oh Complexity Average O(n), Worst Case, O(n2):

- Call new ArrayLargeInteger, Average O(n), Worst Case, O(n2)
- Worst case: will have to iterate through all of the digits. O(n)

The worst case occurs when .max() must iterate through all of the digits in the given numbers. If this is the case, the method itself has a complexity of O(n) + the complexity of the constructor.

.signum()

ArrayLargeInteger's implementation performs equally as well as BigInteger's implementation of signum, with most calls to the method having a difference of around 100-200 nanoseconds. Below is a graph of ALI and BI's .signum()'s time in nanoseconds for numbers of various sizes: 1 digit, 1,000 digits, 2,000 digits, 3,000 digits, 4,000 digits, 5,000 digits, and 6,000 digits. Additionally note that the range of runtime is comparatively small, between 100 and ~500.



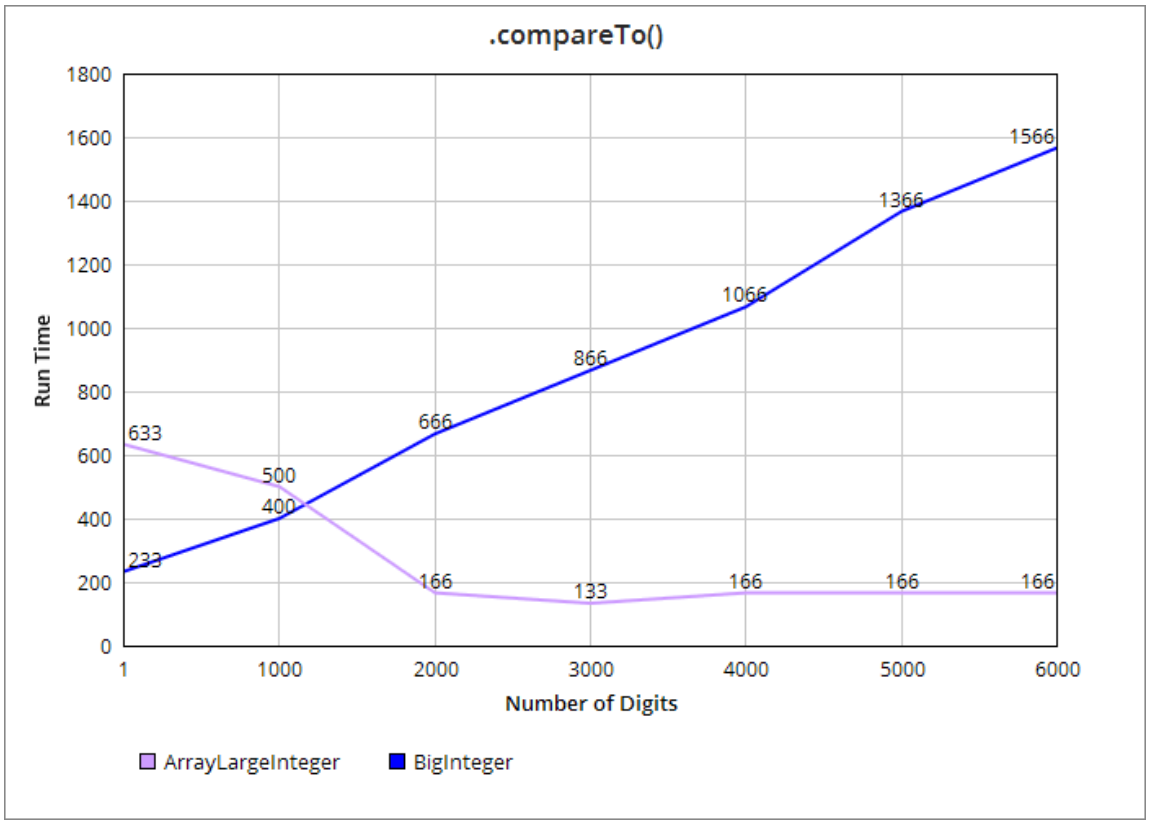
Big-oh Complexity O(1):

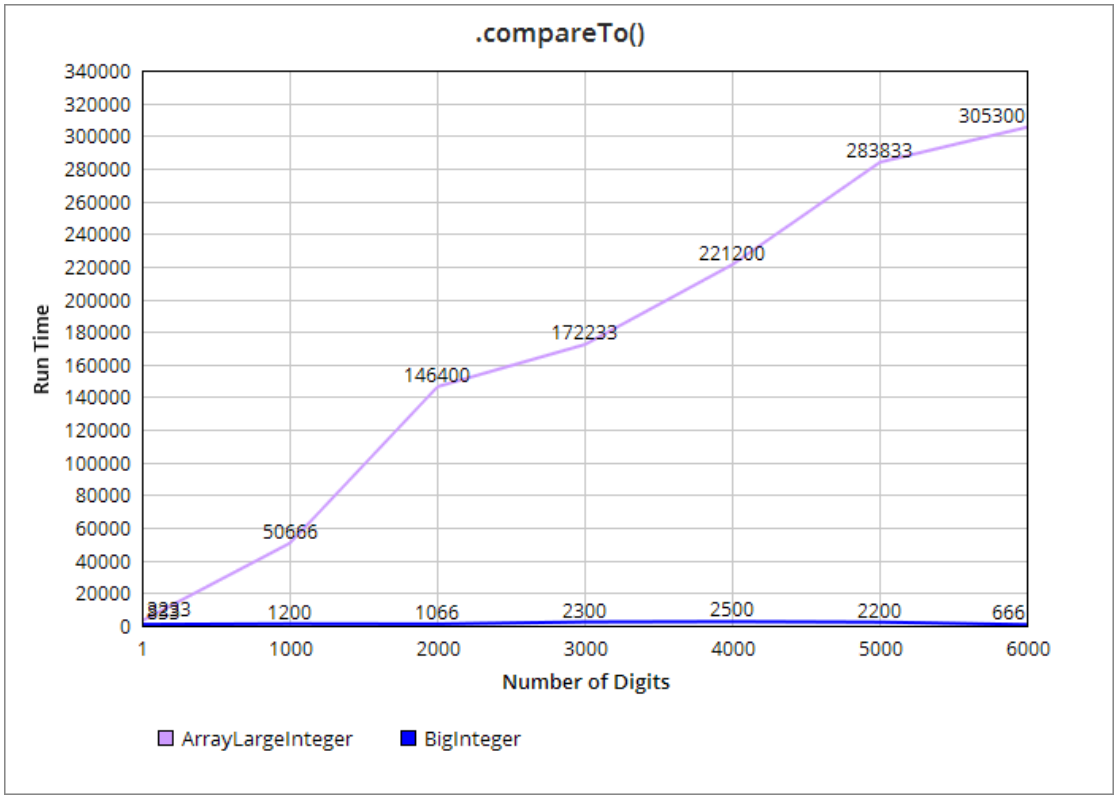
- If string is negative: O(1)
- If its positive,
    - Calls String.equals O(1), because the .equals() only has to check one character

According to the analysis, even in the worst case scenario, this method is O(1). This statement is more credible due to the fact that the graph shares a similarly constant run time.

.compareTo()

The first graph below shows .signum()'s time in nanoseconds for numbers of various sizes: 1 digit, 1,000 digits, 2,000 digits, 3,000 digits, 4,000 digits, 5,000 digits, and 6,000 digits, where the terms compared were equal. It is particularly interesting to observe that at larger numbers, BI's .compareTo() takes much more time than ALI's does. This is due to the fact that ALI's implementation checks for equality first, and can do it quite quickly. The second graph shows .signum()'s time in nanoseconds, where the terms compared were *not* equal. It is clear to see that ALI's implementation performs less well under these conditions. This is because the method must now call .max().


.compareTo()
Run Time
Number of Digits
1800
1600
1400
1200
1000
800
600
400
200
0
1
1000
2000
3000
4000
5000
6000
633
500
400
233
666
866
1066
1366
1566
166
133
166
166
166
ArrayLargeInteger
BigInteger


.compareTo()
Run Time
Number of Digits
340000
320000
300000
280000
260000
240000
220000
200000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
1
1000
2000
3000
4000
5000
6000
50666
146400
172233
221200
283833
305300
1200
1066
2300
2500
2200
666
ArrayLargeInteger
BigInteger

Big-oh Complexity Average O(n), Worst Case, O(n2):

- Calls .equals, which uses String comparison O(n)
- Calls max, Average O(n), Worst Case, O(n2) (due to the constructor)
- Calls .equals, which uses String comparison O(n)

In a best case scenario, where the numbers are equal, ALI's time complexity is practically constant, however in a more likely scenario where the two numbers are not equal, the time complexity is worse, though somewhat linear.

aliDataUsedForReport biDataUsedForReport