COMPENG 2SI4 Lab 1 & 2 Report

Aaron Pinto
400190637
pintoa9

Description of Data Structures and Algorithms

In my implementation of the HugeInteger class, I used an int array with 1 base-10 digit per array element to store the digits of the entire number, and I used an int to store the signum of the number, +1 for positive, 0 for 0, and -1 for negative.

I have a private constructor to create a new HugeInteger from an int array and a sign, that I use in my operations. It wasn't part of the lab spec, but I created it because it was useful to me.

```
private HugeInteger(int[] digits, int signum) {
    this.digits = digits;
    this.signum = signum;
}
```

Addition

1) I check if the other number is 0, and if it is I return the current one

```
if (h.signum == 0) { return this; }
```

2) I check if this number is 0, and if it is I return the other number

```
if (signum == 0) { return h; }
```

3) I check if they have the same signs, and if they do then I return a new HugeInteger with the sum of the absolute values and the same sign as this current one (or the other one, it doesn't matter because they have the same sign)

```
if (signum == h.signum) { return new HugeInteger(add(digits, h.digits), signum); }
```

4) If #3 is false (they have different signs), I compare the 2 numbers, this being the 1st one and the other being the 2nd num

```
int compare = compareDigits(h);
```

5) If the result is 0 (they are the same magnitude), I return a new HugeInteger representing 0

```
if (compare == 0) { return new HugeInteger(new int[0],  signum: 0); }
```

6) If the 1st one is bigger, I subtract the 2nd one from the 1st one, otherwise I subtract the 1st from the 2nd.

```
int[] result = compare == 1 ? subtract(digits, h.digits) : subtract(h.digits, digits);
```

7) I return a new HugeInteger with the result of the subtraction, and a sign of 1 if the 1st number had the same sign as the comparison result, otherwise -1.

```
return new HugeInteger(result, compare == signum ? 1 : -1);
```

In order to sum the absolute values, I check the length of both numbers and make one of them my smaller one and the other the bigger one.

```
// Make digits be the small one by swapping
if (digits.length > hDigits.length) {
    int[] tmp = hDigits;
    hDigits = digits;
    digits = tmp;
}
```

I then create a boolean to handle carries, indices for the big and small numbers starting at their lengths respectively, and the result array and initialize it to the same length as the bigger number.

```
boolean carry = false;
int smallIndex = digits.length, bigIndex = hDigits.length;
int[] result = new int[bigIndex];
```

I iterate through the smaller number and add the corresponding pair of elements with the optional carry of 1. If the sum is greater than 10, I set the carry flag to true and subtract 10 from the sum, otherwise I set the carry flag to false. I then assign the element in the result array at the big number index to the value of the sum.

```
// Start from the end and sum each pair of elements with the optional carry, until small number is done
while (smallIndex > 0) {
    int sum = digits[--smallIndex] + hDigits[--bigIndex] + (carry ? 1 : 0);
    if (sum > 9) { // Can only have 1 digit per element so handle if the sum > 9
        carry = true;
        sum -= 10;
    } else {
        carry = false;
    }
    result[bigIndex] = sum;
}
```

I iterate through the remaining elements in the bigger number, and sum the corresponding digit with the carry, and do the same “sum greater than 10” check as above.

```
// Sum remaining big number elements with optional carry
while (bigIndex > 0) {
    int sum = hDigits[--bigIndex] + (carry ? 1 : 0);
    if (sum > 9) {
        carry = true;
        sum -= 10;
    } else {
        carry = false;
    }
    result[bigIndex] = sum;
}
```

If there's still a carry left over, I create a new array of size result array + 1, copy the result array elements to this new one starting at the new array's 2nd element, and set the 1st element, which represents the most significant digit, to 1. I return the new array.

```
// Grow result depending on possible last carry
if (carry) {
    int[] grown = new int[result.length + 1];
    System.arraycopy(result,  srcPos: 0, grown,  destPos: 1, result.length);
    grown[0] = 1;
    return grown;
}
```

If there's no carry I return the result array.

```
return result;
```

Subtraction

1) I check if the other number is 0, and if it is I return the current one

```
if (h.signum == 0) { return this; }
```

2) I check if this number is 0, and if it is I return the other number with a flipped sign

```
if (signum == 0) { return new HugeInteger(h.digits, -signum); }
```

3) I check if they have different signs, and if they do then I return a new HugeInteger with the sum of the absolute values and the same sign as this current one (because the result will always have the same sign as the current one)

```
if (h.signum != signum) { return new HugeInteger(add(digits, h.digits), signum); }
```

Steps 4-7 are an exact duplicate of the Addition operation.

In order to subtract the absolute values, I pass in the bigger magnitude first, then the smaller one.

```
subtract(int[] big, int[] small)
```

I then create a boolean to handle borrows, indices for the big and small numbers starting at their lengths respectively, and the result array and initialize it to the same length as the bigger number.

```
boolean borrow = false;
int bigIndex = big.length, smallIndex = small.length;
int[] result = new int[bigIndex];
```

I iterate through the smaller number and subtract the corresponding pair of elements with the optional borrow of 1 (big - small - optional borrow). If the difference is less than 0, I set the borrow flag to true and add 10 to the difference, otherwise I set the borrow flag to false. I then assign the element in the result array at the big number index to the value of the difference.

```
// Start from the end and subtract each pair of elements with the optional borrow, until small number is done
while (smallIndex > 0) {
    int difference = big[--bigIndex] - small[--smallIndex] - (borrow ? 1 : 0);
    if (difference < 0) { // Can only have 1 digit per element so handle if the difference < 0
        borrow = true;
        difference += 10;
    } else {
        borrow = false;
    }
    result[bigIndex] = difference;
}
```

I iterate through the remaining elements in the bigger number, and subtract the corresponding digit with the borrow, and do the same “difference less than 0” check as above.

```
// Subtract remaining big number elements with optional borrow
while (bigIndex > 0) {
    int difference = big[--bigIndex] - (borrow ? 1 : 0);
    if (difference < 0) {
        borrow = true;
        difference += 10;
    } else {
        borrow = false;
    }
    result[bigIndex] = difference;
}
```

I return the result array stripped of any leading 0’s.

```
return stripLeadingZeroes(result);
```

Comparison

1) I check if the signs are the same, and if they are I check if this number’s sign is 1, -1 or 0. If it’s 1, I return the result of the comparison of this number’s digits with the other number’s digits.

If it's -1, I return the result of the comparison of the other number's digits with this number's digits. If it's 0, I return 0 because both numbers are the same (0).

```
// Compare signs first, then digits
if (signum == h.signum) {
    switch (signum) {
        case 1:
            return compareDigits(h);
        case -1:
            return h.compareDigits( h: this);
        default:
            return 0;
    }
}
```

2) If #1 is false (they have different signs), I check if this number's sign is greater than the other number's, and if it is I return 1, otherwise -1

```
return signum > h.signum ? 1 : -1;
```

In order to compare the digits, I check the length of both numbers and return -1 if this number is shorter than the other one, or 1 if this number is longer than the other number.

```
// Compare length first then each digit pair
int len1 = digits.length, len2 = h.digits.length;

if (len1 < len2) { return -1; }
if (len1 > len2) { return 1; }
```

I then iterate through the length of this number (or the other, it doesn't matter because they have the same length) and check if any digit is not equal to its corresponding one (same index) in the other number. If they are not the same, I return -1 if the digit in this number is smaller than the other one, otherwise 1.

```
// Iterate through both digits and compare them
for (int i = 0; i < len1; i++) {
    int a = digits[i], b = h.digits[i];
    if (a != b) {
        return a < b ? -1 : 1;
    }
}
```

If I iterate through the numbers fully, then they must be the same so I return 0.

```
return 0;
```

Multiplication

I implemented Karatsuba's multiplication algorithm for my multiplication operation. Karatsuba's algorithm is a recursive divide-and-conquer algorithm which is more efficient for large numbers than the typical algorithm used in multiplication. If the numbers to be multiplied have length n, the typical algorithm has an asymptotic complexity of O(n^2). In contrast, the Karatsuba algorithm has complexity of O(n^(log2(3))), or O(n^1.585). It achieves this increased performance by doing 3 multiplies instead of 4 when evaluating the product.

1) I check if any number is 0, and if it is I return 0.

```
if (signum == 0 || h.signum == 0) {
    return new HugeInteger(new int[0],  signum: 0);
}
```

2) I calculate the result sign (if the 2 numbers have the same sign its 1, otherwise -1) and check if any number has a length of 1. If this number does, I return a new HugeInteger representing the other number multiplied by this number's digit, with the result sign. If the other number does, I return a new HugeInteger representing this number multiplied by the other number's digit, with the result sign.

```
int resSignum = signum == h.signum ? 1 : -1;
int len = digits.length, hLen = h.digits.length;
if (len == 1) {
    return multiplyByDigit(h, digits[0], resSignum);
}
if (hLen == 1) {
    return multiplyByDigit( x: this, h.digits[0], resSignum);
}
```

multiplyByDigit() is an O(n) function because it only iterates through the number of length n once, it only does O(1) operation inside the loop, and it does an optional O(n) copy at the end. Therefore this function does not increase the asymptotic complexity of my multiplication operation.

3) If none of the above base conditions are met, I return the value of the karatsuba algorithm, which recursively calls the multiplication function, which is why I have those base conditions above.

```
return karatsuba( x: this, h);
```

The main step of Karatsuba's algorithm computes the product of two numbers a and b using three multiplications of smaller numbers with about half as many digits as a or b, and some sums, subtractions and digit shifts.

In my implementation, I get the length of both numbers and then find the midpoint of the largest one.

```
int xLen = x.digits.length, yLen = y.digits.length;

int mid = (Math.max(xLen, yLen) + 1) / 2;
```

I then divide both numbers into their upper and lower halves.

```
// Divide both numbers into their upper and lower parts
HugeInteger xUp = x.getUpper(mid), xLow = x.getLower(mid), yUp = y.getUpper(mid), yLow = y.getLower(mid);
```

I then compute the 3 partial results that are used to compute the final result.

```
HugeInteger z0 = xLow.multiply(yLow);
HugeInteger z1 = xUp.multiply(yUp);
HugeInteger z2 = xLow.add(xUp).multiply(yLow.add(yUp));
```

I then compute the final result, using private methods that are O(n), which again doesn't change the asymptotic time complexity.

```
HugeInteger result = z1.shiftLeft(mid).add(z2.subtract(z1).subtract(z0)).shiftLeft(mid).add(z0);
```

I check if the signs of both numbers are not equal to each other, and if so I return a new HugeInteger with the result's digits and the negative of the result's sign, otherwise I return the result.

```
if (x.signum != y.signum) {
    return new HugeInteger(result.digits, -result.signum);
} else {
    return result;
}
```

Theoretical Analysis of Running Time and Memory Requirement

An int is 4 bytes, and each digit is stored in an int in the int array, so each HugeInteger requires 4 bytes * n digits + 4 bytes for the signum, of memory. In a simple formula, y = 4n + 4, where y is the amount of memory required in bytes and n is the number of digits of the number.
Memory required graph

The size of a boolean in Java is VM dependent so I assumed it was also 4 bytes, to make my calculations easier.

Addition

The worst and average case big-Theta time complexity of the addition operation is Θ(n), where n is the number of digits the larger number has. If both numbers are the same length then n is the number of digits any of them have. The algorithm has 2 while loops with O(1) operations inside of them, and both while loops combined iterate n times. The 1st one iterates n - (n - m) times, where m is the number of digits of the smaller number, and the 2nd one iterates n - m times, for a total of n times. It also has an O(n) array copy, but that doesn't affect the asymptotic time complexity.

The extra space needed is up to 8 + (2n)*4 + 3(4(n + 1)) bytes for all the ints, int arrays and booleans, because I have some ints defined for comparison results and indices, ints redefined inside O(n) while loops, and int arrays to store a temp swap, the result and the grown int array.

Subtraction

The worst and average case big-Theta time complexity of the subtraction operation is Θ(n), where n is the number of digits the larger number has. If both numbers are the same length then n is the number of digits any of them have. The algorithm has 2 while loops with O(1) operations inside of them, and both while loops combined iterate n times. The 1st one iterates n - (n - m) times, where m is the number of digits of the smaller number, and the 2nd one iterates n - m times, for a total of n times. It also has an O(n) function call (stripLeadingZeroes()), but that doesn't change the asymptotic time complexity.

The extra space needed is up to 8 + (2n)*4 + 2(4(n + 1)) bytes for all the ints, int arrays and booleans, because I have some ints defined for comparison results and indices, ints redefined inside O(n) while loops, and int arrays to store the result and the stripped int array.

Comparison

The average case big-Theta time complexity of the comparison operation is Θ(1). The worst case time complexity is Θ(n). This happens when both numbers are the exact same, because then my

comparison algorithm has to iterate through the length of one of them to make sure that both digits at each index are the same. In this case, n represents the number of digits of any number.

The extra space needed is up to 8 + 2 * 4n bytes for all the ints because I have some ints defined for the lengths of the numbers and 2 ints redefined inside an O(n) for loop.

Multiplication

The worst and average case big-Theta time complexity of the multiplication operation is $\Theta(n^{\log2(3)})$ or approximately $\Theta(n^{1.585})$, where n is the number of digits the larger number has. If both numbers are the same length then n is the number of digits any of them have.

For one function call the amount of memory required as a function of n, where n is the number of both digits, S(n) = 9 * sizeof(HugeInteger) + 8 * 4n + 2 * sizeof(addition operation) + 2 * sizeof(subtraction operation) + 3 * 4 + S(n/2). According to the geometric series formula, the result should be 2 * S(n), and therefore the space complexity is O(n).

Test Procedure

The point of unit testing, or creating tests in general is so that every single path in your code is tested. Exhaustive testing is nearly impossible. Therefore, my test class only contains enough test cases to test every branch of the code for every function defined in the lab spec and the 2 constructors.

Possible test cases for the string constructor

- Input is null
- Input is an empty string (“”)
- Input is multiple - (“--”)
- Input is - (“-”)
- Input is negative 0 (“-0”)
- Input is 0 (“0”)
- Input is number with illegal digit (“3523534-34568374”)
- Input is valid (“23453”)

Possible test cases for the random constructor

- Input is less than 1 (0)
- Input is valid

Possible test cases for addition

- Other number is 0
- This number is 0

- Both numbers have the same sign
- Numbers have opposite sign and are the same magnitude
- Numbers have opposite sign and different magnitude

Possible test cases for subtraction
- Other number is 0
- This number is 0
- Both numbers have opposite signs
- Numbers have same sign and are the same magnitude
- Numbers have same sign and different magnitude

Possible test cases for multiply
- Other number is 0
- Numbers have same sign
- Numbers have opposite sign
- This number is 1 digit long
- Other number is 1 digit long

Possible test cases for comparison
- Numbers have same positive sign
- Numbers have same negative sign
- Numbers are 0
- This number is longer than the other, with same sign
- Other number is longer than this, with same sign
- Numbers are same length, same sign but different magnitude
- Numbers are the exact same

My output meets the specification of the lab, and I did not have any difficulties debugging my code. An input condition that I could not check is if the input string, or the number of digits is extremely large, as in 2^{64} digits long.

Experimental Measurement, Comparison and Discussion

My timing results

```
Addition runtime
runtime at n=10: 1.0026067272727271E-7
runtime at n=100: 2.643860606611569E-7
runtime at n=250: 5.249928005514652E-7
runtime at n=500: 1.2155958436413769E-6
runtime at n=1000: 2.662860125851285E-6
runtime at n=5000: 2.0249366546598648E-5
runtime at n=10000: 5.3766363023150894E-5

Subtraction runtime
runtime at n=10: 9.888545454545463E-8
runtime at n=100: 2.7333294049586764E-7
runtime at n=250: 5.399251358226894E-7
runtime at n=500: 1.2099292830529329E-6
runtime at n=1000: 3.3420006298459343E-6
runtime at n=5000: 2.872401198754404E-5
runtime at n=10000: 6.091181896352315E-5

Comparison runtime
runtime at n=10: 2.0209545454545452E-8
runtime at n=100: 6.7434867768595O4E-9
runtime at n=250: 1.4485722607062368E-8
runtime at n=500: 4.544324750973294E-9
runtime at n=1000: 4.256821134099757E-9
runtime at n=5000: 3.596589283037271E-9
runtime at n=10000: 3.4234235389367013E-9

Multiplication runtime
runtime at n=10: 6.371305454545454E-7
runtime at n=50: 7.165157004958679E-6
runtime at n=100: 2.7360715245499643E-5
runtime at n=250: 2.466538777749591E-4
runtime at n=500: 9.745410525252272E-4
```

BigInteger timing results

```
Addition runtime
runtime at n=10: 9.025647272727273E-8
runtime at n=100: 1.0155876793388426E-7
runtime at n=250: 7.297078879939894E-8
runtime at n=500: 7.736326171635814E-8
runtime at n=1000: 1.4787739328833056E-7
runtime at n=5000: 3.2437099448443926E-7
runtime at n=10000: 6.155200635862221E-7

Subtraction runtime
runtime at n=10: 1.0951674545454545E-7
runtime at n=100: 1.265845885950413E-7
runtime at n=250: 7.186187807813674E-8
runtime at n=500: 8.952978070980125E-8
runtime at n=1000: 1.4868897982463448E-7
runtime at n=5000: 3.162405907256786E-7
runtime at n=10000: 6.348691508247787E-7

Comparison runtime
runtime at n=10: 2.868643636363636E-8
runtime at n=100: 3.4167858512396706E-8
runtime at n=250: 1.860081689556724E-8
runtime at n=500: 1.0210643789959697E-8
runtime at n=1000: 1.4738751307181454E-8
runtime at n=5000: 1.0355170466428928E-8
runtime at n=10000: 1.1291337913331172E-8
```

```
Multiplication runtime
runtime at n=10: 1.4777450909090914E-7
runtime at n=100: 1.272781319008264E-7
runtime at n=250: 1.262970193809166E-7
runtime at n=500: 2.405383183580084E-7
runtime at n=1000: 5.345956392578E-7
runtime at n=5000: 8.933169669447802E-6
runtime at n=10000: 3.295559025154044E-5
```

Multiplication Results

Discussion of Results and Comparison

The experimental results do make sense because the growth rates (exponents) for all of the measurements were very close to the theoretical ones, but what really killed the performance was the scaling factors. As evident in the multiplication results especially, my scaling factor for karatsuba's algorithm was 5 orders of magnitude greater than the BigInteger's multiplication implementation, which uses standard $O(n^2)$ multiplication, karatsuba's algorithm at $O(n^{1.585})$, and the Toom-Cook algorithm at $O(n^{1.46})$, where n is the number of digits of both numbers.

All of my results were slower than the BigInteger class, for every operation, because the initial implementation of how I stored my number vs how the BigInteger class stores it's number is very different, and my implementation is much worse.

If I were to make any improvements, I would change how I store the HugeInteger to be more like the BigInteger's method, where the value of each element is up to the max value of an int in java, instead of just 1 digit per element. This would improve both the space and time complexities of the program. I would also look into implementing the Schönhage–Strassen multiplication algorithm, which has a time complexity of O(n log n log log n), for larger numbers than Toom-Cook.

Sources of Inspiration

The BigInteger class in Java 11.0.5
Raeed Hassan, 400188200, hassam41
Geometric Series Formula
Exhaustive Testing