Fast Multiplication of Large-Base Integers

Travis Pressler
CSCI B403
Indiana University

April 24, 2014

Abstract

A common operation which needs to be performed in a wide variety of studies with universal application is the multiplication of two integers, but this operation can become slow for very large numbers. This paper describes an algorithm which will perform the multiplication operation on two positive integers of base 2^{32} in a timely and efficient manner.

Contents

1 Summary **2**

2 Control Test **3**
2.1 Control Test Hypothesis 4

3 Algorithm 1.8 - Small Integers **5**
3.1 Algorithm 1.8 Time Complexity Hypothesis 6
3.2 Algorithm 1.8 Test Analysis 7

4 Algorithm 5.2 - Large Integers **8**

5 Finding an Optimal Breakpoint **10**

6 Addition **13**
6.1 Addition Algorithm Time Complexity Hypothesis 14
6.2 Addition Algorithm Test Analysis 17

7 Subtraction **18**
7.1 Subtraction Algorithm Time Complexity Hypothesis 19
7.2 Modifying the Testing Framework 20
7.3 Subtraction Algorithm Test Analysis 22

8 Appendix **25**
8.1 Control Tests 25
8.2 Algorithm 1.8 Tests 26
8.3 Addition Tests 27
8.4 Subtraction Tests 29
8.5 Algorithm 5.2 Code 31

9 References **36**

1 Summary

The algorithm is given two numbers of base 2^{32} in two arrays as inputs where each number in the array represents a digit of each number, respectively. The algorithm produces a single array of integers where each entry in the array represents a digit of the final product. The final running time of this algorithm is $\Theta(n^{lg3})$. This running time is calculated in Section 4.

I used the C programming language for all computations. I used the GNU Compiler Collection(GCC) to compile my code, with the standard compilation options in the makefile which was provided for this assignment(-O2 and -m32). All tests were processed on the Linux machine named 'Hulk', which was accessed at Indiana University through remote login via SSH. The file *main32* was executed as a driver for all of the tests for this assignment, which generated two random input numbers of a specified width, then timed the calculations of the algorithms and checked the correctness of the numbers calculated. Both the time spent and the correctness of our outputted numbers were compared against the GNU Multiple Precision Arithmetic Library's[2] *mpz_mul*, *mpz_add*, and *mpz_sub* functions.

My code utilizes Algorithms 1.8 and 5.2 from *The Analysis of Algorithms*[1] (found on pages 22 and 213, respectively) in its execution. Algorithm 1.8 handles multiplication of integers when the smaller of the two integers has a length smaller than some breakpoint. The value for this breakpoint was determined through testing a range of digit lengths between 0 and 3000. All

numbers larger than the breakpoint utilize Algorithm 5.2, which can either subdivide the problem further, or invoke Algorithm 1.8. This approach is more optimal, because the overhead involved with the execution of Algorithm 5.2 causes it to run slower on small problems, but runs much faster than Algorithm 1.8 on very large integers.

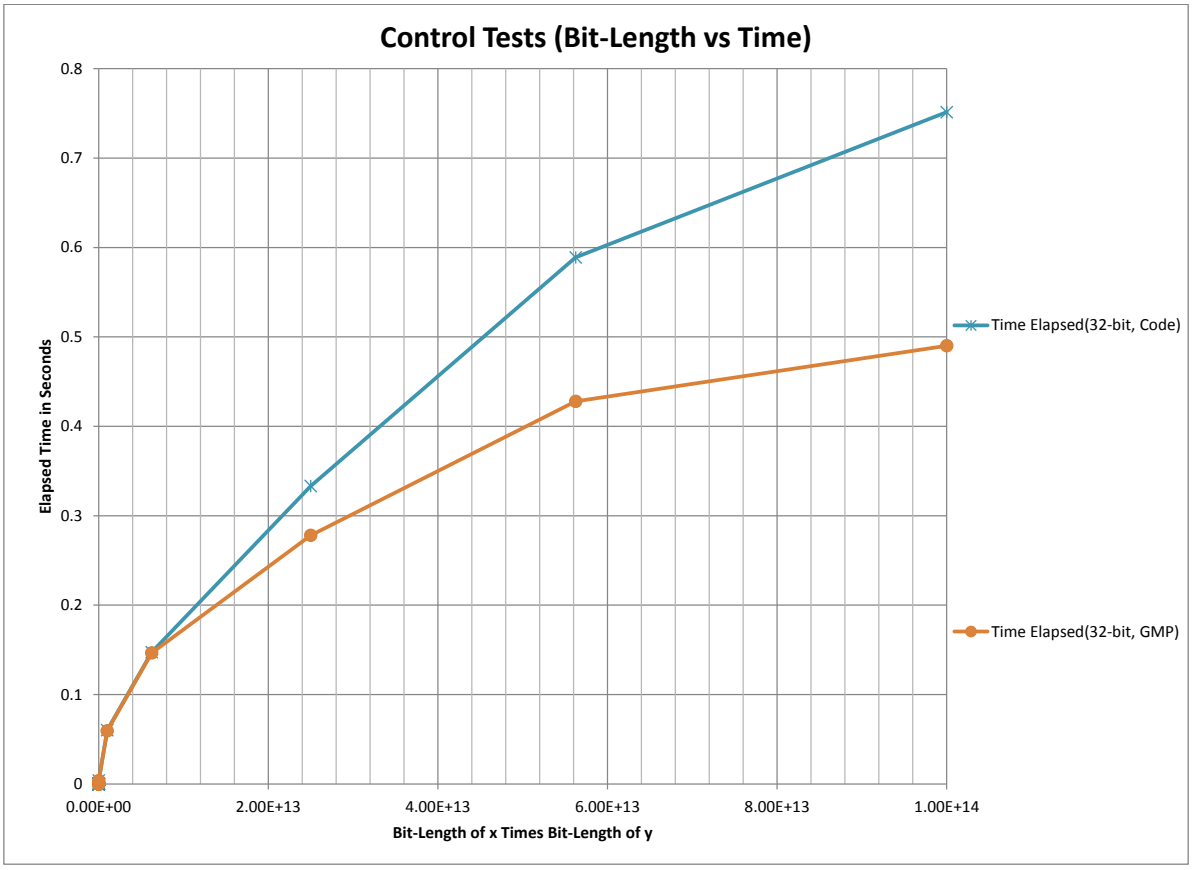
My code for multiplication of integers smaller than the breakpoint is based on a nested loop which iterates over the digits of both input numbers (this technique is described on Algorithm 1.8 from *The Analysis of Algorithms*[1]). I generalized this algorithm by allowing for multiplication of integers of unequal length. If the first number is of length n and the second is of length m, then the inner loop is done mn times. All work done within these two loops is of constant time complexity, so the time complexity of the entire algorithm is upper-bounded by $O(mn)$.

Multiplication of integers larger than the breakpoint is handled Karatsuba and Offman's algorithm for fast multiplication which is described in Algorithm 5.2 on page 213 of *The Analysis of Algorithms*[1]. My implementation of this algorithm allows for integers of unequal length. As described on page 213, this algorithm has a time complexity bounded by $O(n^{log_2 3}) \approx O(n^{1.59})$ when the integers are of equal length. Detailed analysis of my general solution can be found in section 4..

2 Control Test

This preliminary test was run without any changes to the template code. The template code consisted of a driver, *main32*, which executed a function *Product32* within *scaffold32.c*, which simply called GMP[2]'s multiplication algorithm, *mpz_mul*. modified code to see what impact any changes I did to the code would have on the running time of the GMP[2] algorithm and to see how efficient the provided code was as a baseline metric. A graph representing the data acquired in this test is provided below(this data can be viewed in Appendix 8.1):

Figure 1: Running Time Analysis of Unmodified Code



2.1 Control Test Hypothesis

From this view of the data, I hypothesize that both the algorithm provided to us with the template *scaffold32.c* code and the execution of *mpz_mul* within the *main32* driver have elapsed times which clearly do better than a linear multiple of the sum of $b_x b_y$, where b_x is the size in bits of x, and b_y is the size of y in bits. Because the increase in elapsed time decreases as the size of $b_x b_y$ increases, the function could be said to have a big O approximation of $O(b_x b_y)$, but has a big Θ which is less than $\Theta(b_x b_y)$, because the increase in elapsed time is less than one would see in a linear function of $b_x b_y$.

3 Algorithm 1.8 - Small Integers

My implementation of Algorithm 1.8 is detailed below:

Listing 1: Algorithm 1.8 Source Code

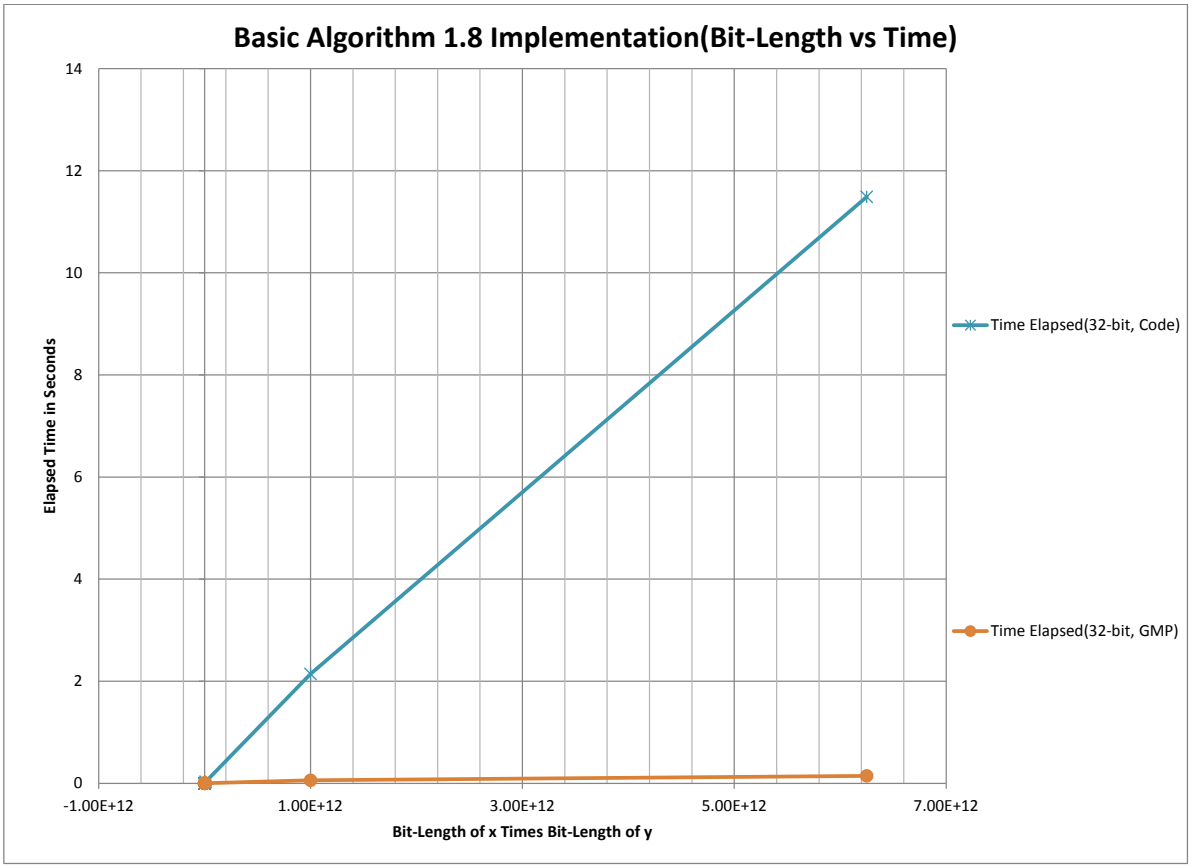
```
1  void onePointEight(void *a, void *b, void *c,
2      unsigned int wa, unsigned int ba, unsigned int wb,
3      unsigned int bb, unsigned int *wc,
4      unsigned int *bc) {
5    unsigned int *int_a = (unsigned int *) a;
6    unsigned int *int_b = (unsigned int *) b;
7    unsigned int *int_c = (unsigned int *) c;
8
9    unsigned int carry = 0L;
10   int twiceLength = wa+wb;
11   unsigned long long int product = 0LL;
12   unsigned long long int BASE = 4294967296LL;
13
14   int zIter = 0;
15   for(zIter = 0; zIter < twiceLength; zIter++) {
16     int_c[zIter] = 0;
17   }
18
19   int j = 0;
20   int i = 0;
21   for(i = 0; i < wa; i++) {
22     carry = 0LL;
23     for(j = 0; j < wb; j++) {
24         product = (unsigned long long int)int_a[i] *
25                   (unsigned long long int)int_b[j] +
26                   (unsigned long long int)int_c[i+j] +
27                   (unsigned long long int)carry;
28         int_c[i+j] = (unsigned int)(product % BASE);
29         if(int_c[i+j]!=0) {
30           (*wc) = (i+j)+1;
31         }
32         carry = (unsigned int)(product / BASE);
33     }
34     int_c[i+wb] = carry;
35     if(carry!=0) {
36       (*wc) = (i+wb)+1;
```

```
37        }
38    }
39    *bc = (*wc * 32);
40 }
```

3.1 Algorithm 1.8 Time Complexity Hypothesis

This algorithm produces 100 out of 100 correct multiplications for all tests within *test_32.sh*, a shell script which runs tests of *main32* using two bit-lengths varying from size 0 to 100000). Zeroing out *int_c* takes $\Theta(w_a + w_b)$ time, and the main nested loop has an upper bound on its time complexity of $\Theta(w_a w_b)$, as all work done within these loops is constant-time work (because the size of digits is capped at 32 bits, in the worst case, the number of steps in the inner multiplication/division/modulus operations will depend on the size of the digit and not on the size of the input). For very large values of w_a and w_b, their product dominates their sum, so the upper bound can be described by $\Theta(w_a w_b)$ A test of the running time for this algorithm against the GMP Library[2] function *mpz_mul* can be found below:

Figure 2: Running Time Analysis of Algorithm 1.8 vs. GMP



3.2 Algorithm 1.8 Test Analysis

As the above graph clearly demonstrates, the elapsed time is linearly proportional to the product of the bit-lengths of x and y(the time t is bounded by some $m(yx)+b$). This result confirms the hypothesis for the calculation of the big-Θ upper bound for the running time of this algorithm being $\Theta(w_a w_b)$, where x is of length w_a bits and y is of length w_b bits. The above graph was generated using data from Appendix 8.2.

4 Algorithm 5.2 - Large Integers

For numbers larger than my breakpoint, I utilize my own implementation of Karatsuba and Offman's multiplication algorithm, the full code for this implementation can be found in Appendix 8.5. To give a very brief summary of the full text of the code, the operations which are involved in this computation and have time complexity worse than constant time are:
Given x and y:
1: Split the digits of x and y into two parts
2: Compute the sum of the upper and lower half of x
3: Compute the sum of the upper and lower half of y
4: Multiply the sums together (call this p_1)
5: Multiply the upper halves (call this p_2)
6: Multiply the lower halves (call this p_3)
7: Subtract p_2 and p_3 from p_1
8: Split the digits of p_3 in two
9: Add the top half of p_3 to p_1
10: Split the digits of p_1 in two
11: Add the top half of p_1 to p_2
12: Compile the digits of p_1, p_2, and p_3 into the final number. The first digits of the output number are set to to the digits of p_2, followed the digits of the bottom half of p_1, followed finally by the bottom digits of p_3.

The equation for the running time of this algorithm can be expressed by the following equation, where the subscripts refer to the input length:

$$l_x + l_y + s_{x/2} + s_{y/2} + p_{s_x/2+s_y/2} + 2p_{x/2+y/2} + m_{p_1} +$$
$$l_{(x/2+y/2)/2} + s_{max((p_3/2),(s_x/2+s_y/2))} + l_{p_{s_x/2+s_y/2}} +$$
$$s_{max(m_{p_1},p_{s_x/2+s_y/2}/2)} + c_{x+y} + \Theta(1)$$

Where l is the time required to perform the splitting operation, s is the time required to perform the sum operation, p_n is the time required to perform the product operation, m is the time required to perform the subtraction operation, c is the time required to do the compilation operation in step 13, and $\Theta(1)$ is our error term to handle all the other operations of the algorithm such as assignments and comparisons.

The splitting operation l takes linear time, as all of the digits of the number being splitted over are iterated over and copied into their respective arrays. The sum operation s takes linear time, as explained in detail in Sections 6.1

and 6.2. The subtraction operator m takes linear time, as explained in detail in Sections 7.1 and 7.3. The compilation operation c takes linear time, as all empty digits of the output number must be iterated over and filled. With this knowledge, we can simplify the previous equation to:

$$3p_{n/2} + n + \Theta(1) \tag{1}$$

$$3p_{n/2} + n \tag{2}$$

For simplicity, n equals the time to do all of the linear-time operations in the above equation, which are halved in every iteration of Algorithm 5.2. $n = x + y$, as both of the inputs are halved in Step **1**. Successive iterations of using Algorithm 5.2 obtains the following result:

$$3(3p_{n/4} + n/2) + n \tag{3}$$

$$3(3(3p_{n/8} + n/4) + n/2) + n \tag{4}$$

$$3(3(3(3p_{n/16} + n/8) + n/4) + n/2) + n \tag{5}$$

For k iterations, this becomes:

$$3^k p_{n/2^k} + n \sum_{0 \le i < k} \left(\frac{3}{2}\right)^i \tag{6}$$

The summation can be turned to a closed form. The equation becomes:

$$3^k p_{n/2^k} + 2n \left[\left(\frac{3}{2}\right)^k - 1\right] \tag{7}$$

All that we have left to do is find a k so that $n/2^k$ is the same as our breakpoint. My breakpoint is determined in the next section, it is found to be optimal to place it at 64. $n/2^k = 64$ when $k = lg(n) - 6$ or $k = lg(\frac{n}{6})$. Replacing this in for our k in the previous equation ($2n$ is also distributed) yields the following:

$$3^{lg(n)-6} p_{n/2^{lg(n)-6}} + 2n \left(\frac{3}{2}\right)^{lg(n)-6} - 2n \tag{8}$$

$$3^{lg(n/6)} p_{n/(n/6)} + 2n \left(\frac{3}{2}\right)^{lg(n)-6} - 2n \tag{9}$$

$$\left(\frac{n}{6}\right)^{lg3} p_6 + 2n \left(\frac{n}{6}\right)^{lg(\frac{3}{2})} - 2n \tag{10}$$

$$\frac{p_6}{6^{lg3}}n^{lg3} + \frac{2}{6^{lg(\frac{3}{2})}}n^{lg3} - 2n \tag{11}$$

The term p_6 means that there are 6 iterations for this algorithm to run through with Algorithm 1.8, which can be considered constant time, since the size of the input is constrained to be less than 64. Replacing some polynomials with asymptotic notations creates the following final running time as a function of the problem size n:

$$\Theta(n^{lg3}) - \Theta(n) = \Theta(n^{lg3}) \tag{12}$$

The technique for finding the running time of secondary recurrences is explained in section 5.2.1 of *The Analysis of Algorithms*[1]. This analysis is based heavily on that technique.

5 Finding an Optimal Breakpoint

Before every multiplication, *Product32* is executed. The purpose of this algorithm is to decide whether Algorithm 1.8 or Algorithm 5.2 should be used for the multiplication of the two input integers. The reason why both 1.8 and 5.2 are useful to use together is that Algorithm 5.2 runs much faster as the size of the input gets very large, but carries overhead. Algorithm 1.8 works faster on small input, as it does not carry as much overhead as Algorithm 5.2. My approach was to find an optimal breakpoint where the switch from Algorithm 5.2 to Algorithm 1.8 produces fast computation time for input sizes of very different magnitudes.

The overhead involved in the execution of the decision routine is negligible, as it only contains three type conversions, eight comparisons, and a maximum of two assignment operations before execution of either Algorithm 1.8 or Algorithm 5.2, and is therefore $\Theta(1)$ work (this is not counting the invocation of my multiplication routines, the timings of which are analyzed in their respective sections). The full source code for my decide routine is listed below:

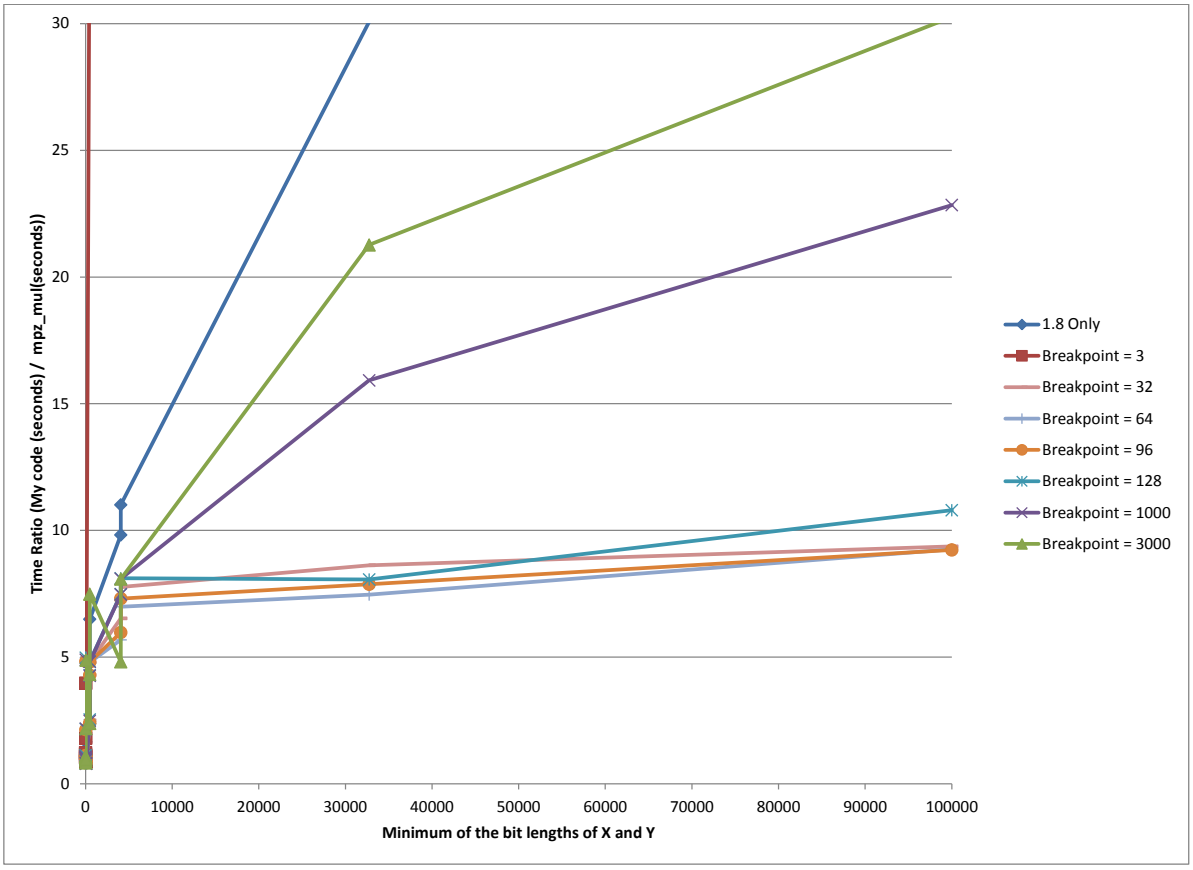
Listing 2: The Final Decide Routine

```
1 const unsigned int BREAKPOINT = 64;
2
3 void Product32(void *a, void *b, void *c,
4     unsigned int wa, unsigned int ba, unsigned int wb,
5     unsigned int bb, unsigned int *wc,
```

```
6       unsigned int *bc){
7     unsigned int *int_a = (unsigned int *)a;
8     unsigned int *int_b = (unsigned int *)b;
9     unsigned int *int_c = (unsigned int *)c;
10
11    if(ba == 0 || bb == 0 || wa == 0 || wb == 0 ||
12      (ba == 1 && int_a[0] == 0) ||
13      (bb == 1 && int_b[0] == 0)) {
14      *wc = 0;
15      *bc = 0;
16      return;
17    }
18
19    unsigned int min = wa;
20    if(wa > wb) {
21      min = wb;
22    }
23    if (min < BREAKPOINT) {
24      onePointEight(a,b,c,wa,ba,wb,bb,wc,bc);
25    }
26    else {
27      fivePointTwo(int_a,int_b,int_c,wa,ba,wb,bb,wc,bc);
28    }
29    return;
30  }
```

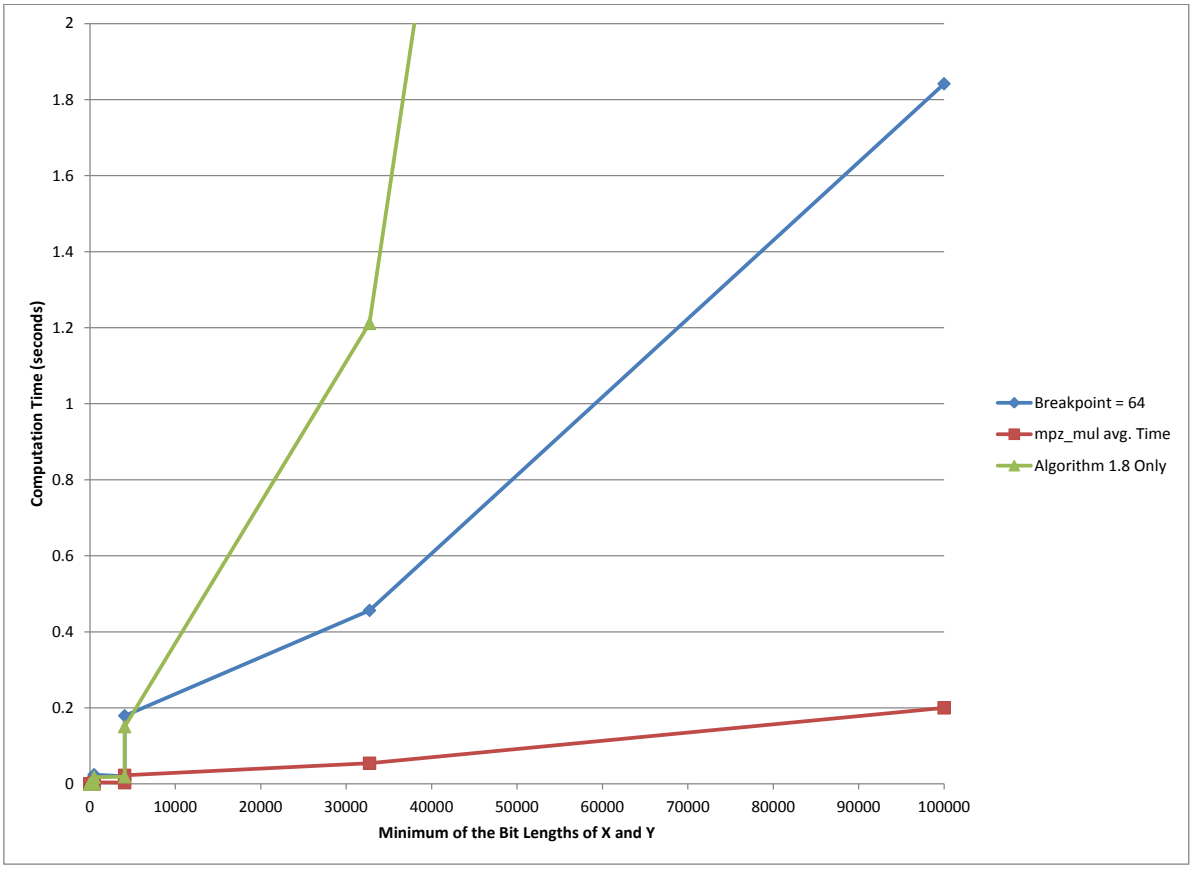
In order to find a value for *BREAKPOINT* which would provide quick execution for all magnitudes of x and y, I ran *test_32.sh* with many different values for *BREAKPOINT*, and compared relative execution time versus the GMP [2] mpz_mul() operation. Relative timings were a more useful indicator of algorithm speed than absolute timings, because there is a chance of the processor slowing or speeding up between tests due to factors external to my tests. A visualization of these tests can be observed in the graphic below:

Figure 3: Relative Timings of Multiplication Algorithm vs. *mpz_mul*



When *BREAKPOINT* was set to 64, I found the relative time ratio to be at a minimum for all tests between 0 and 10000. In order to view the growth of absolute timings with respect to the size of the input, I produced the following graphic from the same data:

Figure 4: Absolute Timings of Multiplication Algorithm vs. *mpz_mul*



From this view of the timings, one can see that the multiplication algorithm increases at a rate faster than linear time, but much less quickly than using only Algorithm 1.8. This concurs with the calculation of the running time of my implementation of Algorithm 5.2 to be $\Theta(n^{lg3})$ which was calculated in Section 4.

6 Addition

In several stages of the execution of my implementation of Algorithm 5.2, the addition of two large-base integers is required (See line numbers 56, 66,

141-142, and 169-170 of Appendix 8.5). This algorithm is based on Algorithm 1.4 on page 12 of *The Analysis of Algorithms*[1]. The source code is detailed below:

Listing 3: Algorithm 1.4 Source Code

```
1  void onePointFour(unsigned int *X, unsigned int *Y,
2      unsigned int *out, unsigned int Xlength,
3      unsigned int Ylength, unsigned int ansLen,
4      unsigned int *filledAns) {
5    *filledAns = 0;
6    unsigned long int carry = 0LL;
7    unsigned long long int sum = 0LL;
8    unsigned long long int BASE = 4294967296LL;
9
10   unsigned int i = 0;
11   for(i = 0; i < ansLen; i++) {
12     unsigned int Xcomp = X[i];
13     unsigned int Ycomp = Y[i];
14     if(i >= Xlength) {
15       Xcomp = 0L;
16     }
17     if(i >= Ylength) {
18       Ycomp = 0L;
19     }
20     sum = (unsigned long long int)Xcomp +
21           (unsigned long long int)Ycomp +
22           (unsigned long long int)carry;
23     out[i] = (unsigned int)(sum % BASE);
24     if(out[i] > 0) {
25       (*filledAns) = (i + 1);
26     }
27     carry = (unsigned int)(sum / BASE);
28   }
29 }
```

6.1 Addition Algorithm Time Complexity Hypothesis

This algorithm consists of five constant-time assignment operations followed by a for-loop which iterates through the length of *ansLen*. The size of *ansLen* is equal to the maximum of the size of the two inputs plus one

(the value which is interpreted as *ansLen* is initialized on lines 49-52, 59-62, 134-137, and 162-165 in Algorithm 5.2 found in Appendix 8.5). All processes done within the for loop are constant-time operations (all addition, modulo, and quotient operations are bounded by the size of the digit, not the size of the input). In the worst case, the algorithm will execute the contents of the for-loop $max_{xy} + 1$ number of times, where max_{xy} is equal to the maximum number of digits of x and y; the algorithm is bounded by $\Theta(max_{xy})$.

In order to test my assumption that this algorithm is bounded by $\Theta(max_{xy})$, I modified *main32* so that *mpz_add*[2] was executed instead of *mpz_mul*[2], and changed the body of *Product32* to the following set of instructions:

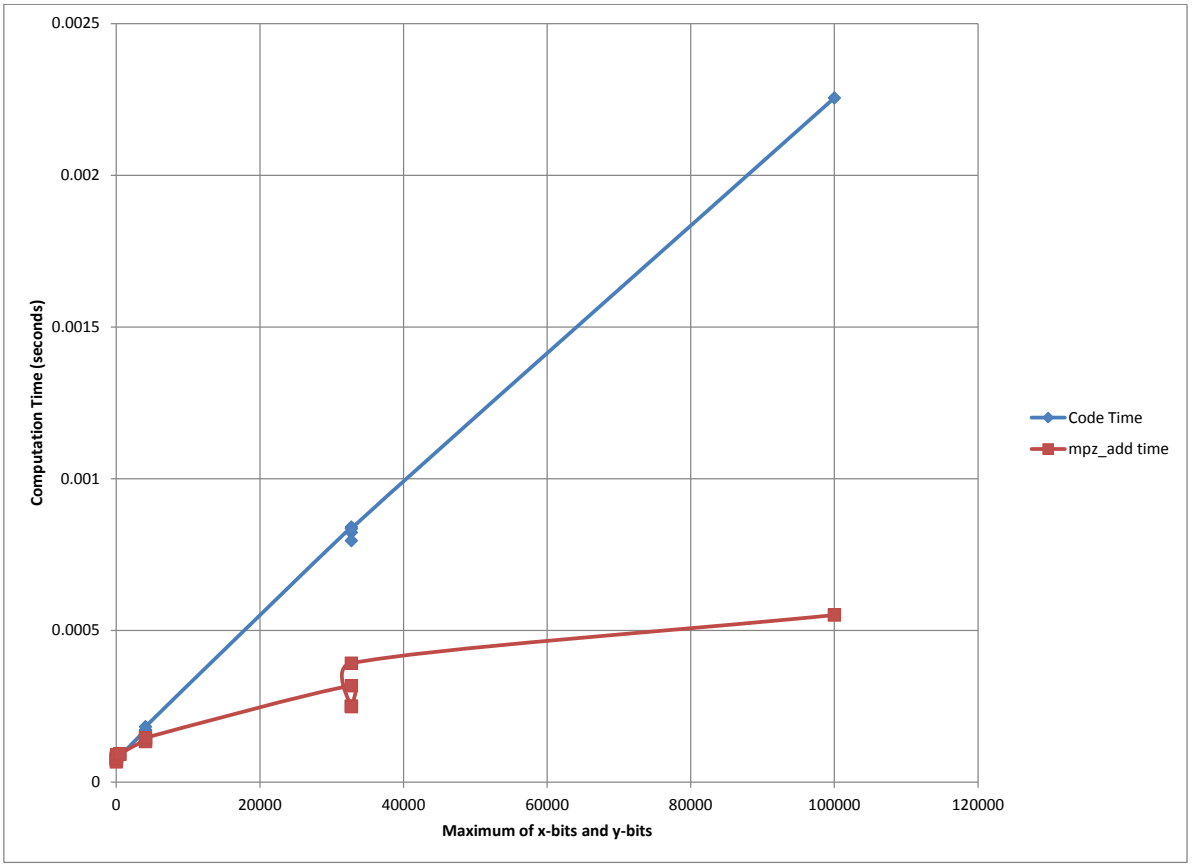
Listing 4: Product32 modified for addition

```
1  void Product32(void *a, void *b, void *c,
2      unsigned int wa, unsigned int ba, unsigned int wb,
3      unsigned int bb, unsigned int *wc,
4      unsigned int *bc){
5    unsigned int *int_a = (unsigned int *)a;
6    unsigned int *int_b = (unsigned int *)b;
7    unsigned int *int_c = (unsigned int *)c;
8
9    if(wa==0) {
10     int i;
11     for(i = 0; i < wb; i++) {
12       int_c[i]=int_b[i];
13     }
14     *wc=wb;
15     *bc=bb;
16     return;
17   }
18   if(wb==0) {
19     int i;
20     for(i = 0; i < wa; i++) {
21       int_c[i]=int_a[i];
22     }
23     *wc=wa;
24     *bc=ba;
25     return;
26   }
27
28   unsigned int large = wa+1;
```

```
29      if(wb > wa) {
30          large = wb+1;
31      }
32      onePointFour(int_a, int_b, int_c, wa, wb, large, wc);
33      return;
34  }
```

On execution of *test_32.sh*, the additions were completed on two numbers from size 0 to 100000 using both methods of addition (the modified *Product32* above and GMP's *mpz_add*[2]). All tests had equivalent output from my tests and GMP's *mpz_add*. The data was then compiled and visualized by the graphic below. The table used to construct this graph can be found in Appendix 8.3.

Figure 5: Running Time Analysis of Addition Algorithm vs. *mpz_add*



6.2 Addition Algorithm Test Analysis

The clustering of multiple timings at similar computation time but identical max_{xy} is due to multiple timings with different minimums but the same maximum, which demonstrates how the time complexity only depends on the size of the largest integer. The linearity of the growth of the computation time versus max_{xy} indicates that the upper-bound to the time complexity of this function is linear, O(max_{xy}).

7 Subtraction

There are two instances in my implementation of Algorithm 5.2 which require subtraction of large-base integers (see line numbers 109-110 and 112-113 in Appendix 8.5). The source code for this algorithm is detailed below:

Listing 5: Subtraction Algorithm

```
1  void subtraction(unsigned int *x, unsigned int *y,
2      unsigned int *out, unsigned int Xlength,
3      unsigned int Ylength, int *filledAns) {
4    unsigned long long int BASE = 4294967296LL;
5    unsigned long long int difference = 0LL;
6
7    unsigned int i = 0;
8    unsigned int j = 0;
9
10   int shouldSubtract = 0;
11   for(i = 0; i < Xlength; i++) {
12     if (i >= Ylength) {
13       out[i] = x[i] - shouldSubtract;
14       shouldSubtract = 0;
15     }
16     else if(x[i] >= y[i] && (shouldSubtract == 0)) {
17       out[i] = x[i] - y[i];
18       shouldSubtract = 0;
19     }
20     else {
21       if(shouldSubtract) {
22         if (x[i] == 0) {
23           out[i] = BASE - y[i] - 1;
24           shouldSubtract = 1;
25
26         }
27         else if (x[i] > y[i]){
28           out[i] = x[i] - y[i] - 1;
29           shouldSubtract = 0;
30         }
31         else if (x[i] <= y[i]) {
32           out[i] = BASE - 1 - y[i] + x[i];
33           shouldSubtract = 1;
34         }
```

```
35          }
36          else {
37            if(y[i] == 0) {
38              out[i] = 0;
39              shouldSubtract = 0;
40            }
41            else {
42              out[i] = BASE - y[i] + x[i];
43              shouldSubtract = 1;
44            }
45          }
46        }
47        if(out[i] != 0) {
48          (*filledAns) = i+1;
49        }
50      }
51  }
```

7.1 Subtraction Algorithm Time Complexity Hypothesis

This algorithm consists of four constant-time operations(initializations of the base, the difference, and the iterators), then is followed by a for-loop which iterates over *Xlength*, which is guaranteed to be the larger of the two input integers. Within the for-loop, there are a series of one to three decisions which lead to an assignment within the output-array and an assignment of *shouldSubtract*. All work done within this for-loop are constant-time operations. After the for loop there is a comparison with a possible constant time operation. These running times can be expressed by the following equation:

$$\Theta(4a) + \Theta(3c + 2a)\Theta(n) + \Theta(c + a) \tag{13}$$

In the above equation, a is the amount of time required to do an assignment operation of an integer, n is the size of the larger of the two inputs, and c is the amount of time required to do a comparison of two integers. The running times for a and c do not depend on the size of the input, so the equation can be simplified to

$$\Theta(1) + \Theta(1)\Theta(n) + \Theta(1) \tag{14}$$

$\Theta(1)\Theta(n)$ can be combined to become $\Theta(n)$. As the size of n becomes large, the contribution of the constant-time operations to the sum becomes insignif-

icant, so the final running time of this subtraction algorithm has a running time bounded by $\Theta(n)$.

7.2 Modifying the Testing Framework

In order to test the accuracy and running time of my subtraction algorithm, significant changes to *Product32* and *main32.c* were required in order to avoid a smaller number subtracting a larger number, as this algorithm is not intended to handle negative integers. Listed below are some relevant portions of *main.c* before modification:

Listing 6: Relevant Portions of *main32.c* Before Modifications

```
mpz_urandomm(x, randstate, xsize_z);
mpz_urandomm(y, randstate, ysize_z);
...
start = mytime();
Product32(x_address, y_address, z_address, *x_size,
          x_bits, *y_size, y_bits, z_size, z_bits);
finish = mytime();
elapsed += finish-start;
...
start = mytime();
mpz_mul(result,x,y);
finish = mytime();
elapsed_gmp += finish-start;
```

Listed below is my modified version of the code listed above to enable subtraction without negative integers:

Listing 7: Relevant Portions of *main32.c* After Modifications

```
mpz_urandomm(x, randstate, xsize_z);
mpz_urandomm(y, randstate, ysize_z);
int shouldFlip = mpz_cmp(y, x);
...
if(shouldFlip > 0) {
  start = mytime();
  Product32(y_address, x_address, z_address, *y_size,
    y_bits, *x_size, x_bits, z_size, z_bits);
  finish = mytime();
  elapsed += finish-start;
```

```
}
else {
  start = mytime();
  Product32(x_address, y_address, z_address, *x_size,
    x_bits, *y_size, y_bits, z_size, z_bits);
  finish = mytime();
  elapsed += finish-start;
}
...
if(shouldFlip > 0) {
  start = mytime();
  mpz_sub(result,y,x);
  finish = mytime();
}
else {
  start = mytime();
  mpz_sub(result,x,y);
  finish = mytime();
}
```

Product32 was also modified to do subtraction operations. The modified code is listed below:

Listing 8: *Product32* Modified for Subtraction Tests

```
1  void Product32(void *a, void *b, void *c,
2      unsigned int wa, unsigned int ba, unsigned int wb,
3      unsigned int bb, unsigned int *wc,
4      unsigned int *bc){
5    unsigned int *int_a = (unsigned int *)a;
6    unsigned int *int_b = (unsigned int *)b;
7    unsigned int *int_c = (unsigned int *)c;
8
9    if(wa==0) {
10     int i;
11     for(i = 0; i < wb; i++) {
12       int_c[i]=int_b[i];
13     }
14     *wc=wb;
15     *bc=bb;
16     return;
17   }
```

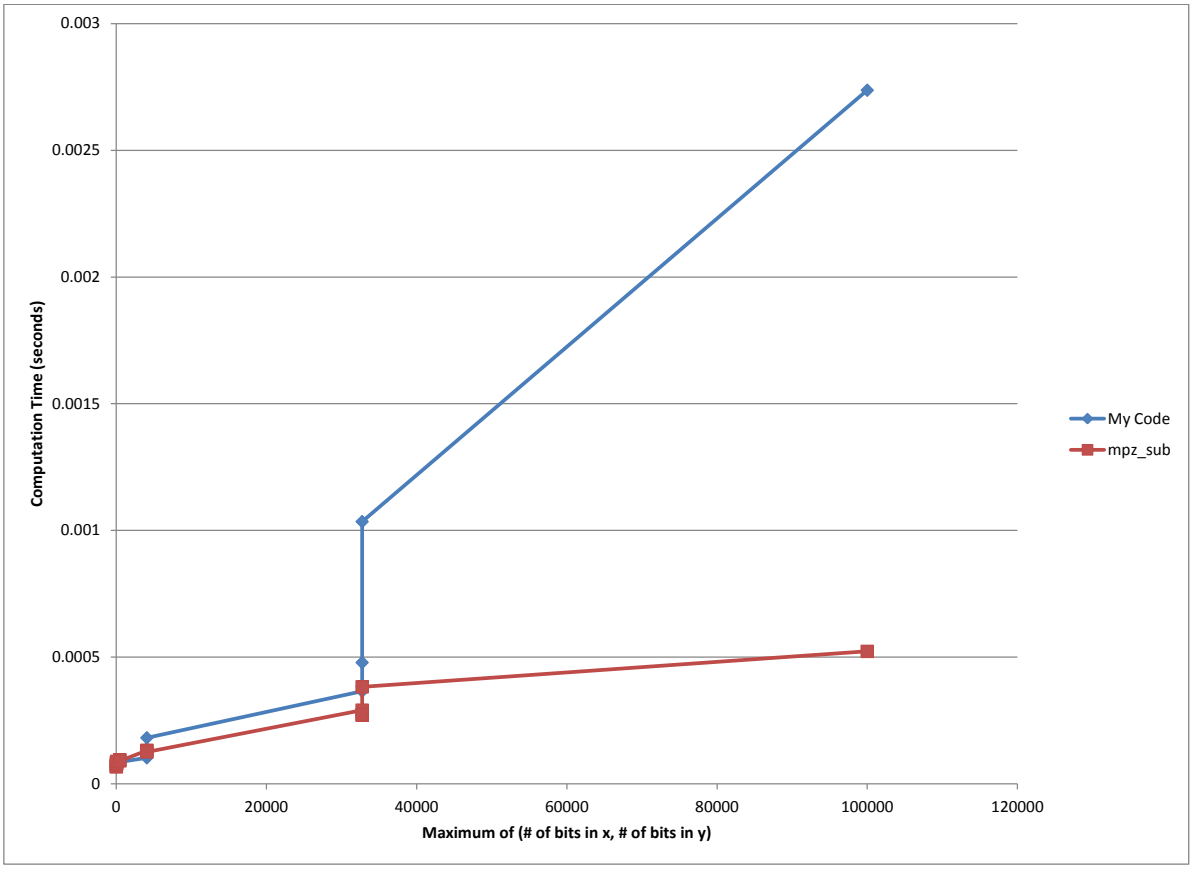
```
18    if(wb==0) {
19      int i;
20      for(i = 0; i < wa; i++) {
21        int_c[i]=int_a[i];
22      }
23      *wc=wa;
24      *bc=ba;
25      return;
26    }
27    subtraction(int_a, int_b, int_c, wa, wb, wc);
28    return;
29  }
```

It is important to note that due to these two changes, the tests of *main32* and *test_32.sh* do not perform the subtraction of $x - y$, but rather performs the subtraction operation $max_{xy} - min_{xy}$, where max_{xy} is the larger of x and y, and min_{xy} is the smaller of x and y.

7.3 Subtraction Algorithm Test Analysis

After running *test_32.sh*, 1300 tests were run using 13 sets of values for the bit lengths of x and y; this process tested the running time and accuracy of my subtraction algorithm compared to *mpz_sub*. These results were compiled and visualized by the graphic below:

Figure 6: Running Time Analysis of Subtraction Algorithm vs. *mpz_sub*



This result confirms my earlier hypothesis that the computation time of my subtraction algorithm increases linearly based on some constant times max_{xy}. The running time can be roughly approximated to be $(2.737 \times 10^{-10}) \cdot max_{xy}$ for each subtraction operation.

An unexpected and interesting result of this test is the evidence that the size of the smaller integer has a significant impact on the running time of my subtraction routine. This is most visible from the following subset of timings:

Table 1: Effect of min_{xy} on the total running time

min_{xy}	max_{xy}	My Time(s)	*mpz_sub* Time(s)
32	32736	0.000366	0.000291
480	32736	0.000384	0.000271
4064	32736	0.000479	0.000274
32736	32736	0.001035	0.000383

There is much less of a pronounced effect of the minimum number on *mpz_sub*'s running time. This difference in computation time is the result of my algorithm taking less time to subtract an empty digit from the larger integer than it would to subtract a digit filled with numbers. This is because the algorithm can exit as early as possible when the iterator over the larger number is larger than min_{xy} (see my subtraction algorithm in Listing 5); if the iterator is less than min_{xy}, there are potentially two further comparisons that need to be made.

8 Appendix

8.1 Control Tests

8.1.1 Test 1

x bits	y bits	My Time(s)	*mpz_mul* time(s)
0	0	0.000002	0.000001
0	1	0.000002	0.000000
1	0	0.000002	0.000001
1	1	0.000003	0.000001
32	32	0.000002	0.000001
32	64	0.000003	0.000001
64	64	0.000004	0.000001
32	480	0.000003	0.000001
32	4064	0.000004	0.000001
480	480	0.000005	0.000002
32	32736	0.000015	0.000006
480	4064	0.000012	0.000007
480	32736	0.000060	0.000052
4064	4064	0.000047	0.000034
4064	32736	0.000279	0.000253
32736	32736	0.000811	0.000778
100000	100000	0.003895	0.003802
1000000	1000000	0.060232	0.059447
2500000	2500000	0.147444	0.146614
5000000	5000000	0.333221	0.278127
7500000	7500000	0.588891	0.427918
10000000	10000000	0.751350	0.490148

Note: each entry in this table refers to 100 multiplication operations

8.2 Algorithm 1.8 Tests

8.2.1 Test 1

x bits	y bits	My Time(s)	Total time(*mpz_mul*)(s)
0	0	0.000001	0.000001
0	1	0.000001	0.000001
1	0	0.000001	0.000001
1	1	0.000001	0.000001
32	32	0.000001	0.000001
32	64	0.000001	0.000001
64	64	0.000001	0.000002
32	480	0.000001	0.000001
32	4064	0.000002	0.000002
480	480	0.000002	0.000003
32	32736	0.000007	0.000006
480	4064	0.000008	0.000008
480	32736	0.000054	0.000053
4064	4064	0.000057	0.000038
4064	32736	0.000442	0.000269
32736	32736	0.003520	0.000788
100000	100000	0.032750	0.003802
1000000	1000000	2.142525	0.031048
2500000	2500000	11.487816	0.076691

Note: each entry in this table refers to 100 multiplication operations

8.3 Addition Tests

8.3.1 Addition Test 1

X bits	Y bits	My Time(s)	(*mpz_mul*) Time(s)
0	0	0.000069	0.000067
1	1	0.000072	0.000076
0	1	0.000079	0.000083
1	0	0.000069	0.000075
32	32	0.000073	0.000079
32	64	0.000084	0.000089
64	64	0.000075	0.000091
32	480	0.000084	0.000092
480	480	0.000085	0.000093
32	4064	0.000172	0.000139
480	4064	0.000172	0.000134
4064	4064	0.000183	0.000146
32	32736	0.000841	0.000318
480	32736	0.000823	0.000251
4064	32736	0.000796	0.000249
32736	32736	0.000836	0.000392
100000	100000	0.002255	0.000551

8.3.2 Addition Test 2

X bits	Y bits	My Time(s)	(*mpz_mul*) Time(s)
0	0	0.000067	0.000066
1	1	0.000065	0.000073
0	1	0.000067	0.000073
1	0	0.000065	0.000070
32	32	0.000072	0.000079
32	64	0.000069	0.000076
64	64	0.000071	0.000083
32	480	0.000080	0.000088
480	480	0.000078	0.000086
32	4064	0.000148	0.000123
480	4064	0.000149	0.000127
4064	4064	0.000154	0.000127
32	32736	0.000763	0.000285
480	32736	0.000760	0.000232
4064	32736	0.000768	0.000233
32736	32736	0.000793	0.000365
100000	100000	0.002140	0.000522

8.3.3 Addition Test 3

X bits	Y bits	My Time(s)	(*mpz_mul*) Time(s)
1	1	0.000067	0.000075
0	1	0.000065	0.000071
1	0	0.000067	0.000072
32	32	0.000068	0.000079
32	64	0.000070	0.000077
64	64	0.000069	0.000085
32	480	0.000077	0.000093
480	480	0.000079	0.000092
32	4064	0.000151	0.000124
480	4064	0.000148	0.000123
4064	4064	0.000160	0.000125
32	32736	0.000764	0.000294
480	32736	0.000768	0.000233
4064	32736	0.000763	0.000236
32736	32736	0.000794	0.000366
100000	100000	0.002137	0.000513

Note: each entry in these tables refers to 100 addition operations

8.4 Subtraction Tests

8.4.1 Subtraction Test 1

X Bits	Y Bits	My Time(s)	*mpz_sub* time(s)
0	0	0.000070	0.000067
1	1	0.000070	0.000083
0	1	0.000069	0.000071
1	0	0.000069	0.000073
32	32	0.000072	0.000081
32	64	0.000073	0.000080
64	64	0.000077	0.000089
32	480	0.000078	0.000095
480	480	0.000087	0.000091
32	4064	0.000103	0.000132
480	4064	0.000112	0.000132
4064	4064	0.000182	0.000126
32	32736	0.000366	0.000291
480	32736	0.000384	0.000271
4064	32736	0.000479	0.000274
32736	32736	0.001035	0.000383
100000	100000	0.002737	0.000523

8.4.2 Subtraction Test 2

X Bits	Y Bits	My Time(s)	*mpz_sub* time(s)
0	0	0.000072	0.000066
1	1	0.000070	0.000080
0	1	0.000069	0.000071
1	0	0.000069	0.000072
32	32	0.000074	0.000080
32	64	0.000072	0.000081
64	64	0.000075	0.000085
32	480	0.000076	0.000092
480	480	0.000091	0.000099
32	4064	0.000102	0.000132
480	4064	0.000111	0.000134
4064	4064	0.000183	0.000127
32	32736	0.000366	0.000290
480	32736	0.000380	0.000276
4064	32736	0.000465	0.000276
32736	32736	0.001036	0.000380
100000	100000	0.002762	0.000521

8.4.3 Subtraction Test 3

X Bits	Y Bits	My Time(s)	*mpz_sub* time(s)
0	0	0.000072	0.000066
1	1	0.000070	0.000078
0	1	0.000069	0.000074
1	0	0.000069	0.000075
32	32	0.000074	0.000082
32	64	0.000075	0.000084
64	64	0.000080	0.000088
32	480	0.000079	0.000092
480	480	0.000086	0.000092
32	4064	0.000102	0.000130
480	4064	0.000111	0.000132
4064	4064	0.000181	0.000129
32	32736	0.000371	0.000295
480	32736	0.000376	0.000272
4064	32736	0.000461	0.000276
32736	32736	0.001032	0.000378
100000	100000	0.002748	0.000516

Note: each entry in these tables refers to 100 subtraction operations

8.5 Algorithm 5.2 Code

```
1  void fivePointTwo(unsigned int *int_a,
2      unsigned int *int_b, unsigned int *int_c,
3      unsigned int wa, unsigned int ba, unsigned int wb,
4      unsigned int bb, unsigned int *wc,
5      unsigned int *bc) {
6    unsigned long long int BASE = 4294967296LL;
7
8    unsigned int max = wa;
9    unsigned int min = wb;
10   if (wb > wa) {
11     max = wb;
12     min = wa;
13   }
14
15   unsigned int n = (max/2) + (max%2);
16
17   unsigned int wU2 = n;
18   unsigned int wU1 = 0;
19   if(wa>wU2) {
20     wU1 = wa-wU2;
21   }
22   unsigned int *U1 =
23         (unsigned int*)malloc(wU1*sizeof(unsigned int));
24   unsigned int *U2 =
25         (unsigned int*)malloc(wU2*sizeof(unsigned int));
26   unsigned int filledU1 = 0;
27   unsigned int filledU2 = 0;
28   splitDigits(int_a, wa, U1, wU1, &filledU1, U2, wU2,
29       &filledU2);
30   wU1 = filledU1;
31   wU2 = filledU2;
32
33   unsigned int wV2 = n;
34   unsigned int wV1 = 0;
35   if(wb>wV2) {
36     wV1 = wb-wV2;
```

```
37    }
38    unsigned int *V1 =
39          (unsigned int*)malloc(wV1*sizeof(unsigned int));
40    unsigned int *V2 =
41          (unsigned int*)malloc(wV2*sizeof(unsigned int));
42    unsigned int filledV1 = 0;
43    unsigned int filledV2 = 0;
44    splitDigits(int_b, wb, V1, wV1, &filledV1, V2, wV2,
45        &filledV2);
46    wV1 = filledV1;
47    wV2 = filledV2;
48
49    unsigned int wT1 = wU1+1;
50    if(wU2>wU1) {
51      wT1 = wU2+1;
52    }
53    unsigned int *T1 =
54          (unsigned int*)malloc(wT1*sizeof(unsigned int));
55    unsigned int filledT1 = 0;
56    onePointFour(U1, U2, T1, wU1, wU2, wT1, &filledT1);
57    wT1=filledT1;
58
59    unsigned int wT2 = wV1+1;
60    if(wV2>wV1) {
61      wT2 = wV2+1;
62    }
63    unsigned int *T2 =
64          (unsigned int*)malloc(wT2*sizeof(unsigned int));
65    unsigned int filledT2 = 0;
66    onePointFour(V1, V2, T2, wV1, wV2, wT2, &filledT2);
67    wT2=filledT2;
68
69    unsigned int wW3 = filledT1+filledT2;
70    unsigned int *W3 =
71          (unsigned int*)malloc(wW3*sizeof(unsigned int));
72    unsigned int filledW3 = 0;
73    unsigned int w3bc = 0;
74    Product32(T1, T2, W3, wT1, filledT1*32,
75                      wT2, filledT2*32, &filledW3, &w3bc);
76    wW3=filledW3;
77    free(T1);
```

```
78    free(T2);
79
80    unsigned int wW2 = wU1+wV1;
81    unsigned int *W2 =
82          (unsigned int*)malloc(wW2*sizeof(unsigned int));
83    unsigned int filledW2 = 0;
84    unsigned int w2bc;
85    Product32(U1, V1, W2, wU1, filledU1*32, wV1,
86                        filledV1*32, &filledW2, &w2bc);
87    wW2=filledW2;
88    free(U1);
89    free(V1);
90
91    unsigned int wW4 = filledU2+filledV2;
92    unsigned int *W4 =
93        (unsigned int*)malloc(wW4*sizeof(unsigned int));
94    unsigned int filledW4 = 0;
95    unsigned int w4bc;
96    unsigned int step5Product = 0;
97    Product32(U2, V2, W4, filledU2, filledU2*32, filledV2,
98        filledV2*32, &step5Product, &w4bc);
99    wW4=step5Product;
100   free(U2);
101   free(V2);
102
103
104   unsigned int wW3minusW2 = wW3;
105   unsigned int *W3minusW2 =
106       (unsigned int*)malloc
107       (wW3minusW2*sizeof(unsigned int));
108   unsigned int filledW3minusW2 = 0;
109   subtraction(W3,W2,W3minusW2,wW3, wW2,
110       &filledW3minusW2);
111   wW3minusW2 = filledW3minusW2;
112   subtraction(W3minusW2,W4,W3,wW3minusW2, wW4,
113       &filledW3);
114   wW3=filledW3;
115   free(W3minusW2);
116
117   unsigned int wNewW4 = n;
118   unsigned int wC = 0;
```

```
119   if(wW4 > n) {
120     wC = wW4-n;
121   }
122   unsigned int *C =
123         (unsigned int*)malloc(wC*sizeof(unsigned int));
124   unsigned int *newW4 = (unsigned int*)
125         malloc(wNewW4*sizeof(unsigned int));
126   unsigned int filledC=0;
127   unsigned int filledNewW4=0;
128   splitDigits(W4, wW4, C, wC, &filledC,
129                                newW4, wNewW4, &filledNewW4);
130   free(W4);
131   wC=filledC;
132   wNewW4=filledNewW4;
133
134   unsigned int wNewW3 = wW3+1;
135   if(wC>wW3) {
136     wNewW3 = wC+1;
137   }
138   unsigned int *newW3 =
139     (unsigned int*)malloc(wNewW3*sizeof(unsigned int));
140   unsigned int filledNewW3 = 0;
141   onePointFour(C, W3, newW3, wC, wW3,wNewW3,
142       &filledNewW3);
143   free(C);
144   wNewW3 = filledNewW3;
145
146   unsigned int wC2 = 0;
147   if(wNewW3 > n) {
148     wC2 = wNewW3 - n;
149   }
150   unsigned int filledC2=0;
151   unsigned int *C2 =
152     (unsigned int*)malloc(wC2*sizeof(unsigned int));
153   wW3 = n;
154   splitDigits(newW3, wNewW3, C2, wC2,
155                      &filledC2, W3, wW3, &filledW3);
156   wC2 = filledC2;
157   wW3 = filledW3;
158
159   unsigned int wW1 = wW2;
```

```
160   unsigned int *W1 =
161     (unsigned int*)malloc(wW1*sizeof(unsigned int));
162   unsigned int wNewW2 = wW2+1;
163   if(wC2>wNewW2) {
164     wNewW2 = wC2+1;
165   }
166   unsigned int *newW2 =
167     (unsigned int*)malloc(wNewW2*sizeof(unsigned int));
168   unsigned int filledNewW2 = 0;
169   onePointFour(C2, W2, newW2, wC2, wW2, wNewW2,
170     &filledNewW2);
171   wNewW2=filledNewW2;
172   free(C2);
173
174   unsigned int filledW1 = 0;
175   wW2 = n;
176   wW1 = 0;
177   if(wNewW2 > n) {
178     wW1 = wNewW2-n;
179   }
180   splitDigits(newW2, wNewW2, W1, wW1, &filledW1, W2,
181     wW2, &filledW2);
182   wW2 = filledW2;
183   wW1 = filledW1;
184
185   unsigned int constructIt = 0;
186   unsigned int totalIt = 0;
187   unsigned int finalConstruct[max*4];
188   unsigned int restZeroIt = 0;
189   unsigned int restZero = 0;
190   for(constructIt = 0;
191       constructIt < wNewW4;
192       constructIt++) {
193       int_c[totalIt] = newW4[constructIt];
194       totalIt++;
195   }
196   free(newW4);
197   for(constructIt = 0;
198       constructIt < wW3;
199       constructIt++) {
200       int_c[totalIt] = W3[constructIt];
```

```
201         totalIt++;
202     }
203     free(W3);
204     for(constructIt = 0;
205         constructIt < wW2;
206         constructIt++) {
207         int_c[totalIt] = W2[constructIt];
208         totalIt++;
209     }
210     free(W2);
211     for(constructIt = 0;
212         constructIt < wW1;
213         constructIt++) {
214         int_c[totalIt] = W1[constructIt];
215         totalIt++;
216     }
217     free(W1);
218
219     *wc = totalIt;
220     *bc = totalIt*32;
221
222     return;
223 }
```

9 References

1. *The Analysis of Algorithms* by Paul Walton Purdom, Jr and Cynthia A. Brown. ©1985, Hardcover.

2. GMP: The GNU Multiple Precision Arithmetic Library. https://gmplib.org/

3. I discussed this algorithm with Zhaozhou Situ, an associate instructor for B403. His explanations were very helpful, especially regarding how to implement the decision routine.

4. I found Professor Bramley's *elapsedtime.c* to be a useful resource in learning how to time my code. His method for timing using *clock_gettime()*(using CLOCK_ABSTIME instead of CLOCK_REALTIME) was used extensively in my code before a final version was completed.

   http://www.cs.indiana.edu/classes/p573/notes/clock_examples/c/elapsedtime.c