Stefen Pegels sgp62

Setup

Q1) Top Four methods in encrypt, based on total computing time (run on one-liners):

Bignum::operator-(Bignum)
emplace()
Bignum::operator*(int)
Bignum::operator+(Bignum)

Top Four methods in decrypt, based on shorter decrypt of one-liners (about 30% length)
Bignum::operator-(Bignum)
Bignum::operator+(Bignum)
emplace()
Bignum::operator*(int)

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 42.70      9.53     9.53  5297483     0.00     0.00  Bignum::operator*(int const&) const
 22.31     14.52     4.98 10780360     0.00     0.00  Bignum::operator-(Bignum const&) const
 20.65     19.13     4.61  2651288     0.00     0.00  Bignum::operator+(Bignum const&) const
 11.11     21.61     2.48 815136333     0.00     0.00  __gnu_cxx::__normal_iterator<int*, std::vector
<int, std::allocator<int> > > std::vector<int, std::allocator<int> >::_M_emplace_aux<int&>(__gnu_cxx:
:__normal_iterator<int const*, std::vector<int, std::allocator<int> > >, int&)
  1.30     21.90     0.29 69056685     0.00     0.00  void std::vector<int, std::allocator<int> >::_M
_realloc_insert<int>(__gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int> > >, int
&&)
```

The list of the top four is the same for both encryption and decryption, based on total time. It seems like the most improvements need to happen within the three operators, and maybe structuring code differently to avoid emplace(), or use a different data structure that can emplace in constant time rather than linear based on the size of the vector(since it's shifting the others). Emplace seems to have many recursive function calls within the operators, which contribute a lot to the total execution time, even if each call is short compared to an operator call.

Q2) For algorithm performance, the do-while loop of the division operator features much of the "grade school" subtraction algorithm in computing the division result, and if this were improved, a lot of the subtraction calls would go away. For a large bignum, like those created in the expmod functions, combined with a small divisor would result in A LOT of subtraction calls.

```
                0.01    0.00   17864/10780360     Bignum::operator%(Bignum const&) const [4]
                4.97    2.57 10762496/10780360     Bignum::operator/(Bignum const&) const [6]
[7]     33.8    4.98    2.57 10780360         Bignum::operator-(Bignum const&) const [7]
                2.48    0.00 815136333/815136333      __gnu_cxx::__normal_iterator<int*, std::vector<i
nt, std::allocator<int> > > std::vector<int, std::allocator<int> >::_M_emplace_aux<int&>(__gnu_cxx::_
_normal_iterator<int const*, std::vector<int, std::allocator<int> > >, int&) [9]
                0.04    0.00 10780360/18729131     std::_Deque_base<int, std::allocator<int> >::_M_in
itialize_map(unsigned long) [13]
                0.03    0.00 10279091/31994163     std::vector<int, std::allocator<int> >::operator=(
std::vector<int, std::allocator<int> > const&) [12]
                0.02    0.00 5447813/69056685     void std::vector<int, std::allocator<int> >::_M_rea
lloc_insert<int>(__gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int> > >, int&&)
[10]
                0.00    0.00  501269/526329      Bignum::Bignum(Bignum const&) [25]
```

The above gprof output shows how expensive the % operator is, with many division and subtraction calls beneath it. Why can't we return the tmp variable storing the remainder for us in division instead?

This issue is made worse in the expmod function, which curiously chooses to modify its exponent by DIVIDING by two at each iteration, which involves a long execution of the division loop subtraction, given the grade school implementation. The implementation of expmod could be changed to subtract 1 to the exponent at each step (while changing the multiplication) to reduce some of this algorithmic complexity. This expmod issue applies both to encryption and decryption.

A malloc_insert() method is called recursively millions of times as well; this might point to poor vector modification with methods like emplace that might be changed to reduce these calls to a reasonable level. If the vector index were not as long, there would need to be fewer calls allocating new vector elements.

These problems afflict encryption and decryption equally, as they both revolve around the expmod function and the operators it calls. However, rsa_d is much larger than rsa_e, so decrypt has even worse performance.

Possible Optimizations

Q3)

1. As shown in the gprof output from part 2, % contributes to cost in a significant way because of its 3 operations in one statement inside its function body, and it's possible to determine if exp is even or odd just by looking at the last bit of its binary representation. If the bit is a 0, it is even. If it is a 1, it's odd. This is an easy replacement which also exists inside a while loop inside expmod, so that saves us time if we can replace it with std::vector.back and a bitwise & to check if the number is even. This statement will be evaluated log2(exp) times, since exp is divided by two at each iteration of the while loop. This is worth pursuing, although modulus is called within the same while loop body at least once per iteration, so changing mod's implementation would maybe be more worthwhile afterwards.

2. The a % b implementation includes 3 operations, even though the remainder is calculated inside the division function by the tmp variable already. Since the “replacement” for a-(a/b)*b is a derivation of a/b, the validity of this improvement hinges on the significance of the * call in the % operator.

```
                0.00   14.65   17864/17864       Bignum::expmod(Bignum, Bignum const&) const [2]
[4]     65.6    0.00   14.65   17864         Bignum::operator%(Bignum const&) const [4]
                0.01    9.17   17864/28574       Bignum::operator*(Bignum const&) const [3]
                0.04    5.42   17864/25018       Bignum::operator/(Bignum const&) const [6]
                0.01    0.00   17864/10780360     Bignum::operator-(Bignum const&) const [7]
```

In the above gprof segment, the * function calls actually take up more time than the / and - calls combined within the expmod % calls, so optimizing this to remove the multiplication will hopefully cut the time for modulus execution by more than half, assuming returning the remainder from the division operation does not add any unnecessary cost. I believe it shouldn’t since its calculation is already being factored into the speed of division anyways.

3. All of the bignum operations besides expmod can pass their arguments by reference, and all of the methods don’t modify the digits variable, so they can be declared const. This would shift the evaluation to compile time for a lot of operations, saving execution time resolving the formerly constant expressions. Also, passing by reference will prevent copies of Bignums to accumulate in memory where possible, so we might avoid some slowdown in a long execution where otherwise memory would be come densely filled with useless Bignum copies.

```
 11.11     21.61     2.48 815136333     0.00     0.00  __gnu_cxx::__normal_iterator<int*, std::vector
<int, std::allocator<int> > > std::vector<int, std::allocator<int> >::_M_emplace_aux<int&>(__gnu_cxx:
:__normal_iterator<int const*, std::vector<int, std::allocator<int> > >, int&)
  1.30     21.90     0.29 69056685     0.00     0.00  void std::vector<int, std::allocator<int> >::_M
_realloc_insert<int>(__gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int> > >, int
&&)
```

In the base execution, emplace is called the most of any method, with over 800 million calls in this data sample. For reference, multiplication, the most expensive operator from this gprof output, is only called 5 million times. This means that for most operators, there are at least 100 emplace operations happening in EVERY operation, which would add up cost since emplace for a std::vector has linear time complexity based on the number of elements. This is because when emplacing, the vector creates space at the front, and pushes every single element backwards. I think removing emplace will have significant benefits for the code, especially if it can be replaced with a constant time operation.

4. The advantage of base 10000 seems to be that we can do up to 4 digit by 4 digit multiplication at once, instead of being limited by only one digit per index in the vector.

```
                9.53    0.22 5297483/5297483     Bignum::operator*(Bignum const&) const [3]
[5]     43.7    9.53    0.22 5297483         Bignum::operator*(int const&) const [5]
                0.19    0.00 44129992/69056685     void std::vector<int, std::allocator<int> >::_M_re
alloc_insert<int>(__gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int> > >, int&&)
 [10]
                0.02    0.00 5297483/18729131     std::_Deque_base<int, std::allocator<int> >::_M_ini
tialize_map(unsigned long) [13]
                0.01    0.00 5297483/31994163     std::vector<int, std::allocator<int> >::operator=(s
td::vector<int, std::allocator<int> > const&) [12]
```

If we look at the gprof output for the division operator, it is evident that a lot of function calls are used on vector operations (69 million in this case). If we can use base 10000,

many of those operations would be diminished with up to 4x reduced vector size and fewer carry operations since more can be condensed into one element * another element by the CPU, rather than our digit by digit multiplication. I think this can create a good amount of improvement, but the setup to support base 10000 might add other costs that could slow down, especially switching between base 10 and base 10000, since the base can no longer be a constant variable as it would be with only one base. For the division operator, the grade school subtraction method will create really long function calls when dealing with base 10000 numbers, so a binary search that optimizes the number of subtractions that have to be called would be useful. The performance increase of this depends on how inexpensive the searching in the binary search is, compared to the simple subtraction in the original division algorithm.

Results and Analysis

Q5)

Table of Speedups(or Lack Thereof)

Flags	**Execution Time**	**Speed Increase**
None*	38.070s	N/A
-o1	37.744s	**1.01x**
-o2	29.493s	**1.29x**
-o3	28.381s	**1.34x**
-o4	16.221s	**2.35x**
-o123	17.322s	**2.20x**
-o1234	9.827s	**3.87x**

Data taken from first 5 lines of one-liners in one file

***None includes const methods and pass by reference**

How I Went About Things

To handle the optional flags in the first place, I created them as static boolean class variables, and set them in main. I also set an integer to bump the assignments of the other arguments down by 1 in argv[], in the event that one or more of the optional flags were present.

-o1: For this flag, only one line of code was changed, converting from the modulus operator inside the expmod function to a bitwise & comparison. I duplicated the conditional and used my static boolean OPT1 for the modularity.

```
                0.00   24.41      16/16          Bignum::decrypt(std::__cxx11::basic_string<char, std
::char_traits<char>, std::allocator<char> > const&, std::__cxx11::basic_string<char, std::char_traits
<char>, std::allocator<char> > const&) const [1]
[2]    100.0    0.00   24.41      16          Bignum::expmod(Bignum, Bignum const&) const [2]
                0.00   16.02   20416/20416       Bignum::operator%(Bignum const&) const [3]
```

(no flags above)

```
                0.00   24.81      16/16          Bignum::decrypt(std::__cxx11::basic_string<char, std
::char_traits<char>, std::allocator<char> > const&, std::__cxx11::basic_string<char, std::char_traits
<char>, std::allocator<char> > const&) const [1]
[2]     99.8    0.00   24.81      16         Bignum::expmod(Bignum, Bignum const&) const [2]
                0.00   13.28   12240/12240       Bignum::operator%(Bignum const&) const [4]
```

From the decrypt gprof we can see the calls of % was reduced from 20416 to 12240, a 40% decrease, which accounts for the minor speedup, but the percentage improvement for this method was only 3%. I wasn't really expecting much from this optimization given its one line in a conditional block with 3 modulus operations, but one second was a good sign of actual change.

-o2: For this optimization, I duplicated the entire division function and replaced the return type with a std::pair that included two bignums; the quotient and remainder. Since tmp was storing the remainder throughout execution, returning it didn't require any new calculations. Inside the modulus function, I added a branch for my opt flag and returned pair.second to get the second element (the remainder). The goal here was to remove the * operations that occupied a lot of the modulus time.

```
[3]     65.6    0.00   16.02   20416         Bignum::operator%(Bignum const&) const [3]
                0.01    9.97   20416/32656       Bignum::operator*(Bignum const&) const [4]
                0.03    5.99   20416/28592       Bignum::operator/(Bignum const&) const [6]
                0.01    0.01   20416/12017653     Bignum::operator-(Bignum const&) const [7]
-----------------------------------------------
```

(no flags above)

```
                0.01    6.70   20416/20416       Bignum::operator%(Bignum const&) const [5]
[6]     39.8    0.01    6.70   20416         Bignum::pair_divide(Bignum const&) const [6]
                4.29    2.34 9162181/11997237     Bignum::operator-(Bignum const&) const [3]
                0.05    0.00 9162181/28708639     std::vector<int, std::allocator<int> >::operator=(s
td::vector<int, std::allocator<int> > const&) [12]
                0.02    0.00  241136/1721100     void std::vector<int, std::allocator<int> >::_M_real
loc_insert<int const&>(__gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int> > >, i
nt const&) [14]
                0.00    0.00 2049833/2682777     Bignum::operator==(Bignum const&) const [26]
                0.00    0.00   40832/615200      Bignum::Bignum(Bignum const&) [27]
```

Visible above, even though both operators had the same number of calls (20416), the original modulus occupied 65.6% of the code runtime, whereas the new one only took up 39.8%. Also notice there are no calls to multiplication in the -o2 version. This sizeable improvement in the optimization resulted in a ~30% increase in speed, which is expected given this gprof data and limited other concessions made to implement the pair.

-o3: For this optimization, I first made all methods const that I could const if they did not modify the digits Bignum vector, and passed arguments by reference when they were not modified. However, this was a bit inconsequential for comparison's sake, since it applies to the speedup of the base version and ALL versions. This reduced the amount of allocations for digits vectors when they are passed by reference, instead of copies.

To handle emplace, I decided to use the std::deque data structure, which functions similar to a vector except with constant time emplace at the beginning. In each operator that uses emplace, I created a std::deque and used it for all of the accumulation and indexing operations of that operator, so that it mirrored what res.digits would hold. Then, I was able to call emplace_front on the deque for a constant time operation. Initially, I was faced with the issue of a linear time for loop that copied all of the elements back into res.digits to return the

proper Bignum, but I was able to solve this by assigning res.digits to a new vector that used the iterators from the deque so that the beginning and end pointed to the same block of memory, requiring no copying of the elements.

```
 time   seconds   seconds    calls   s/call   s/call  name
 41.84     10.13    10.13  5933603     0.00     0.00  Bignum::operator*(int const&) const
 21.85     15.43     5.29 12017653     0.00     0.00  Bignum::operator-(Bignum const&) const
 20.98     20.51     5.08  2969744     0.00     0.00  Bignum::operator+(Bignum const&) const
 11.73     23.35     2.84 889320631     0.00     0.00  __gnu_cxx::__normal_iterator<int*, std::vector
<int, std::allocator<int> > > std::vector<int, std::allocator<int> >::_M_emplace_aux<int&>(__gnu_cxx:
:__normal_iterator<int const*, std::vector<int, std::allocator<int> > >, int&)
  1.98     23.83     0.48 75581295     0.00     0.00  void std::vector<int, std::allocator<int> >::_M
_realloc_insert<int>(__gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int> > >, int
&&)
```

(no flags above, -o3 below)

```
 time   seconds   seconds    calls   s/call   s/call  name
 42.97     10.97    10.97  5933603     0.00     0.00  Bignum::operator*(int const&) const
 30.98     18.89     7.91 12017653     0.00     0.00  Bignum::operator-(Bignum const&) const
 23.35     24.85     5.96  2969744     0.00     0.00  Bignum::operator+(Bignum const&) const
  0.94     25.09     0.24 48276775     0.00     0.00  void std::vector<int, std::allocator<int> >::_M
_realloc_insert<int>(__gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int> > >, int
&&)
  0.82     25.30     0.21 35710971     0.00     0.00  std::vector<int, std::allocator<int> >::operato
r=(std::vector<int, std::allocator<int> > const&)
  0.31     25.38     0.08    28592     0.00     0.00  Bignum::operator/(Bignum const&) const
  0.27     25.45     0.07 26854603     0.00     0.00  std::_Deque_base<int, std::allocator<int> >::_M
_initialize_map(unsigned long)
  0.12     25.48     0.03 16265259     0.00     0.00  void std::deque<int, std::allocator<int> >::_M_
push_front_aux<int&>(int&)
  0.12     25.51     0.03  6306688     0.00     0.00  void std::deque<int, std::allocator<int> >::_M_
push_front_aux<int>(int&&)
```

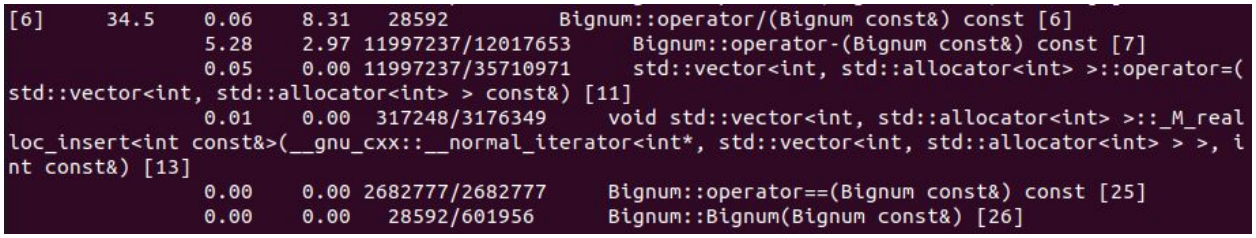
In the original implementation, emplace was called an incredibly large number of times, with linear complexity. In the new version, the deque operations for push_front take up much less time, as shown at the bottom of the second gprof. However, the performance tradeoff for this implementation was the cost of initializing a map, shown above the push_front for the deque. Since I created a deque in every operator, this became expensive, and limited the scale of my optimization. Had I changed the bignum data structure to utilize a deque instead of a vector, this might have been negated, but I didn't want to change every vector occurrence in the code.

-o4: Optimization 4 was the hardest academic task I've done at Cornell in all 3 years I've been here, and I nearly lost my sanity numerous times over the past five days implementing it. That being said, I found it incredibly rewarding in terms of my own growth with C++. To start, I implemented Bignum::BASE as a static class int that changes based on the argument flag, and replaced every 10 in any of the arithmetic operations with BASE. From there, I modified the string constructor of bignum to incorporate base 10000, by grouping the string argument into sections of four and adding that to the digits variable with a while loop. I also made a method that took a Base 10000 implementation and turned it back into a base 10 representation of a vector, because I was having all sorts of issues with my character output for the decrypt function, since I wasn't able to modify the to_char() function successfully. This function does the reverse of the constructor: it takes the indices, extends them to 4 digits, and slices them individually into a new vector with one digit per index.

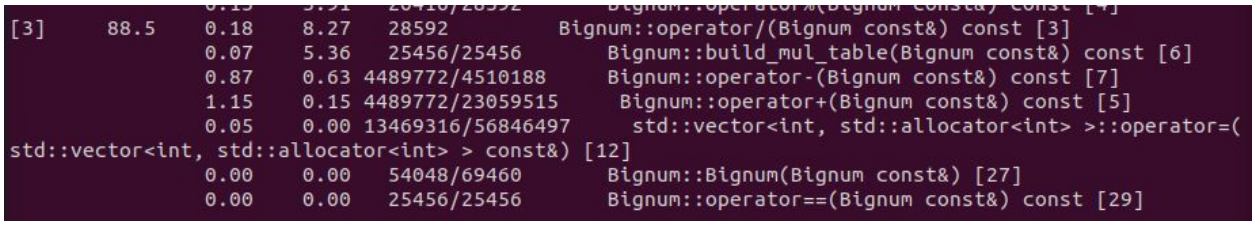
For tackling binary search, I added a static class mul_table variable, which was a vector that was going to store b*2^n values, where b is the divisor in A/b and n is the minimum number of entries where b*2^n > A. Inside this function, I first clear the current mul_table and vector I

made that stores powers of 2 to add quickly to the quotient. I clear by swapping them with an empty vector to save time. The mul_table is built by adding the previous value to itself in such a way that B, B+B, 2B+2B, 4B+4B are the entries, to avoid multiplication. The entries are all Bignums, a decision I made to allow integers greater than the maximum int value in the table.

In the binary search algorithm, the multiplication table is built, and 3 index values perform binary search until the greatest b value <= A is found, and then subtracted out of A. After this, the search algorithm is run again, with the new value of A (since A = A-b), until A is <= b and we have finished our division method. I also switched some logic in main with the base of the output to give my base 10000 bignums the correct character output during decrypt.



(no flags above)



While a lot of speed was improved with the Base 10000 and binary search implementation (2.35x), the limiting factor on my code versus the expected instructor improvement was the amount of resources used creating and maintaining the mul_table for searching. Division actually took up 88.5% of execution time in the binary search method; the only reason total time was shorter was time saved in other operations due to faster base 10000 arithmetic compared to base 10. Build_mul_table actually took more time than the subtraction operations themselves, the main point of optimization with binary search. Also, allocating space for the table vector and clearing it took a huge performance hit, although it took me many hours to figure out I needed to clear it in the first place. Alternative methods to maintain the table (or replace it entirely) proved unsuccessful for me, but if I had more time I would investigate it further. A binary search without a multiplication table, combined with base 10000, seems like it would provide a greater speedup than my current implementation.