**What lines are most of the time spent on?**
In the flat profile section, most of the time is spent in the vector methods [] and size(), which makes sense. However, this is only 10% and 6% respectively, so not actually too high.
The top 4 are vector operations [], size(), and allocate (which comes from push_back I think). As well as BigNum -. It is very similar for encryption and decryption.

**What methods are most of the time spent on?**
I found the cumulative section to be much more interesting. Of course, modexp takes up 99% of the time, and the things it calls the most are /, %, *, with all other calls being inconsequential compared to the time these three take. / and % both funnel to divmod, which is why divmod takes up 60% of the time, and * takes up the remaining 40%. This is quite a watertight look actually, only two functions make up any significant portion of the runtime.

Looking closer at divmod, it spends a lot of its time doing the - operation, because in the first part of this assignment I found that to be faster than computing multiples and subtracting every time. It also spends some time in comparators.

Looking closer at *, it spends about half its time in intmult(), which I think could be an easy target for speedup. The other half of its time is distributed amongst + and vector operations, which I think are already fairly optimized.

**Optimization Techniques**
1. Implement multithreaded version: A non-threaded encryption approach encrypts one line at a time using one thread for the whole file. Change Bignum to have multithreading and to use k threads that each encrypt lines in parallel (you can pick k for your program or determine it based on the total number of lines). We need the output order to match the input order, so generate the encrypted data into a std::vector of std::string lines, and when every line is encrypted print the whole file.

I implemented a multithreaded version where I have k worker threads that each decrypt lines in parallel. (I didn't multithread encrypt because encrypt was already orders of magnitude faster than decrypt). I used a similar multithreading structure as the word counter program, with an atomic number that tells each worker which line it should work on right now, and a global result vector that all the workers write into. It has sped up the program dramatically, now decrypting the entire one liners file takes almost the same time as decrypting ½ of a line (tasks are split up be half-lines).

2. In modular exponentiation we have to decide at each step whether the corresponding bit of the exponent is 0 or 1. Ken did this using the % operator, but this is quite an expensive way to figure out if exp is even or odd. Of course you may have solved this some other way, but if you did what Ken did, is it worthwhile to special-case the modulus 2 computation. Would it pay off to modify the exponentiation logic to eliminate this inefficiency? Think about the gprof output before assuming that the best way to answer such a question is to just make the change to the code. Sometimes gprof can "tell you the answer!"

My modExp() function had this snippet of code

```
BigNum modded = b % BigNum("2");
b = b / BigNum("2");
```

Which is what (2) is talking about. However, I noticed that this will call divmod() twice, when in reality, it only needs to call it once. The gprof shows this really well.
This is before changing:

```
                0.00    1.09    1022/3576       BigNum::operator/(BigNum const&) const [12]
                0.00    2.73    2554/3576       BigNum::operator%(BigNum const&) const [5]
[4]     57.8    0.00    3.83    3576        divmod(BigNum const&, BigNum const&) [4]
```

And this is after changing the code

```
                0.01    1.37    1022/2554       BigNum::modExp(BigNum const&, BigNum const&)
        const [3]
                0.01    2.06    1532/2554       BigNum::operator%(BigNum const&) const [7]
[4]     52.6    0.02    3.43    2554        divmod(BigNum const&, BigNum const&) [4]
```

Clearly, exactly 1022 calls of divmod from % disappeared, while not increasing any other calls. And this was all done by changing the code to

```
auto [quotient, remainder] = divmod(b, BigNum("2"));
b = quotient;
```

I know that I can check even-odd very quickly by just looking at the last digit, however, I still need to divide by 2 anyways.

3. Ken's way of computing a % b is to compute a – (a/b)*b. This is trivial to code, but in fact inefficient: When we compute a/b we obtain the remainder at that same moment. Did you implement modulus the same way? Is this a significant part of the measured cost?

My code already had a divmod() static helper function that both divide and mod used.

4. Ken's version of Bignum doesn't pass arguments by reference or use the const declaration. Yet the Bignum methods mostly don't modify class variables (so the methods are const) and very few of them change their arguments (so those could be passed as const references). Ken also uses emplace to put digits in front of a vector in a few places, which can cause copying. This is an easy change to just make and test. Are there constants that weren't declared as const? Constant arithmetic is a big deal in C++… Try it.

My code passes arguments by const reference, and in divmod(), I made sure to use reversed vectors in order to only use push_back and not use emplace in the front.

5. Ideas 1-3 were all easy. Now we'll get wild and crazy. Our Bignum vector is base 10… but it didn't really need to be. We could have used base 100 or base 16, or really any number we like. With a vector of type unsigned long (64 bit integers), the math would be safe against overflow

up to a fairly large base value. You need to look hard at the Bignum class to figure out this upper safe limit: it is a function of the kinds of operations we are doing when we compute +, -, *, / and %. Ken's code can easily be changed to support powers of 10 bases up to 10,000 (but not larger). Modify Bignum to run with base 10,000.

With base 10,000 we could hold 4 digits per element instead of 1 digit per element, leading to a significant speedup for larger numbers. However, the code for this requires many changes across the board, and it overall makes Bignum less elegant to work with. Furthermore, I was simply not that interested in this optimization, so I decided to spend my time exploring other ideas.

6. When multiplying or dividing, we often need to know the single-digit multiples of a number. For example, to divide 97 into 1877327911, we will need to know the multiples of 97. With bignums, this can be a costly step – it won't be 97, but more likely a number with 2000 digits! Can you come up with a way to avoid doing more multiplications than necessary?

I tried this by simply caching the integer multiples of the number, and the results were quite dramatic. This is gprof before:

```
                0.18    1.14  235440/235440      BigNum::operator*(BigNum const&) const [5]
[10]    20.1    0.18    1.14  235440         BigNum::intMult(int) const [10]
```

And this is after

```
                0.01    0.14   40840/40840       getIntMultLookup(BigNum const&) [44]
[48]     2.7    0.01    0.14   40840         BigNum::intMult(int) const [48]
```

It was called 6x less, which is quite a massive speedup, since intmult used to be 20.1% of all time used, and now is only 2.7%. Furthermore, the optimization does not call any other methods, so this is a pure speedup with no downside.

I decided to use this optimization because it is simple and elegant, and looking at the before gprof output, it is clear that the * operator took a lot of time, 46% to be exact, and so a speedup to intmult, which accounts for around half the time that * spends, would be a worthwhile endeavor.