CS4414 Fall 2024 Homework 4 Part 2
Due December 8

Goals: Homework 4 Part2 is built from the Homework 4 Part1 bignum application. As we learned on hw6, a main use of Bignum packages is to support RSA encryption. Briefly, RSA encryption is a mathematical transformation that centers on two keys: a public key that you can hand out without limitations, and a private one that you guard carefully. Any RSA system has a key length, usually 2046 or even 4092. To make our solution a tiny bit simpler we will work with 512 bit keys in homework 4 part2.

As explained on many very nice wiki pages (google them and read any that you like!), you use ssh_key to create these RSA keys (however, nowadays the recommended encoding for ssh_key is ed25519), and will typically make use of three of the ssh_key outputs: something called the public modulus **n**, the public exponent **e**, and the private exponent **d**. You can think of **(n,e)** as the public key, and **(n,d)** as the private key.

To encrypt a message m, you compute **m^e % n**. To decrypt it, you compute **m^d % n**. We implemented this operation in part 1 and will use it heavily here in part 2.

In RSA, both encryption and decryption use the identical expression: **m^e % n:** m can be either the message to encrypt or it can be the encrypted data we want to decrypt. Ken used ssh_key to generate a 512-bit key, then turned the key into a decimal number (because Bignum is coded for decimal arithmetic, but ssh_key output is in hexadecimal). Then, he simply encoded rsa_n, rsa_e and rsa_d as built-in constants, to keep his main program very simple. Normally, of course, you need to look up a public key in a certificate service, and you would find the private key in a hidden, carefully protected ssh key file, which the program would read. To publish a key, you would upload it to the certificate service with your identifying information. You carefully keep the private key secure. But adding file I/O to bignum is an extra complication we decided to avoid. Here are the values you will need:

```
rsa_n =
"96165402670130584772537629772934250633792434584735938169004540197211175700032488  0811399265283685752965867557035683506718471520123051990736165379532846 2699";
rsa_e = "65537";
rsa_d =
"48020339163872217484261813509148210724346418270901449753861827402748568533182765  18446521844642275539818092186650425384826827514552122318308590929813048801";
```

Notice that these are strings. Your bignum constructor will need to convert from string to bignum.

For part 2, add commands to your main program, e for encrypt and d for decrypt. The e command reads input one text line at a time, from std::cin (so this could be redirected from a file, or you could just type it). Line by line, it starts by scanning the line one character at a time and converting each character into a 3-digit ASCII decimal code. For example, the ASCII code for the letter A is 41 in hexadecimal, hence 65. The ASCII code for letter b (lower case, this time!) is hex 62, hence 98.

We end up with a string suitable for input to Bignum, in which each group of 3 digits represents one character from the input line.

Now we run into a small issue: a 512-bit RSA key can encode at most a 512-bit number. This works out to be a limit of 51 characters[1]. Our input will sometimes be shorter than 51, but a few lines of text are as much as 96 characters long. So the RSA encryption code will need to read its input, line by line, and encode each single line into two strings, each covering half of the input line. We will pad, if needed, so that every encrypted line is identical in length. If a line happens to be longer than 96 characters, truncate it and only consider the first 96. Lines can thus be any non-zero length, but we will always end up encrypting two 51-character chunks.

How should we do this padding? A natural thought would be that if the input text is shorter than 51 characters, pad it to be 102 characters long by appending spaces. *But this is insecure.* The problem is that if we append blanks, when encrypting a short line with just 37 characters, for example, we will end up with one 51-character chunk that is entirely blank – and because it will always be identical, it will always encrypt identically. This reveals a lot about the input.

A further problem is that identical inputs would produce identical outputs.

To solve all of these issues we will add a *line counting* feature. As we read each line of input, prepend a line-count, which for us will be a 3-digit integer converted to 3 ascii characters. Then, when padding, generate padding that has this same line count in the last 3 digits. In effect, out of our budget of 102 characters, we are going to deliberately waste 6: the line count at the front, and the one at the back, starting the counter at 1.

Examples: Input "The cake is a lie!" has 18 characters. If this is input line 93 we prepend " 93" (notice the blank) and now have a 21 character string: " 93The cake is a lie!". 102-21 is 81, so we now append 78 blanks followed by " 93" again, yielding:

" 93The cake is a lie! 93".

This 102-character string then is broken into two 51-character lines for encryption. If we encounter the identical text input again as line 821 of the input, we encrypt it as

"821The cake is a lie! 821".

How can we convert a text string like this to a Bignum, and later convert back? Just turn each character of text into its ascii numerical code. In some sense this is automatic: a char in C++ "is" an unsigned 8-bit number. You don't have to convert it. But you do need to turn it into a number in the proper base (radix): if your Bignum is using base 10, you need to know what the character looks like in base 10 (it will need to always be 3 digits to accommodate the biggest possible char). If your Bignum uses base 16, you extract two base-16 digits, etc. In base 512, the whole character becomes a single digit! So: extract these digits, a fixed number of digits per char, and initialize the digits of your Bignum from the digits of your character string. The length of the Bignum will be k*s where s is the size of the string (always 51) and k is the number of digits per character determined by the Bignum base. Later, when decrypting, you

[1] Why 51? Well, it takes 3 decimal digits to encode an ASCII character as a base-10 integer. Thus a 51 character text string will be converted to a 153 digit base-10 number. The limit comes from rsa_n, the modulus. In 512-bit RSA, rsa_n is 154 digits long in base-10: we need the length of our bignum to be smaller than the length of the modulus.

will implement a method to do the exact reverse operation: given a sequence of digits it will turn it into a sequence of chars, and then turn that sequence of characters into a std::string.

Command ‘e’ computes the RSA encryption of the two Bignums. We end up with two outputs, also Bignums (the second one will be an encryption of the second line, or if the line was short, an encryption of a small line-number like 3, 4, … this is so that the output doesn’t reveal information about the length of the input).

With d, you are expected to input two numbers that were created with command e. The program converts them to Bignums, computes the RSA decryption of each, turns them back into ASCII strings 3 characters at a time, and outputs those two strings. Decryption will also strip off the leading 3-digit line number, the trailing 3-digit line number, and any trailing blanks.

For example, if the only input to our program is “The cake is a lie!” the strings to encrypt would be

“ 1The cake is a lie! “ // 51 characters
“ 1” // again, 51 characters

To understand the conversion rule, consider “ 1The”. This has two spaces. As a decimal number, a space is 32, which we can think of as 032. 1 is 049 and T is 084. The h and e become 104 and 101. So this 6-character sequence encodes as 032032049084104101. We strip the leading 0 (recall the rule from hw6, or revisit hw6 to remind yourself) and end up with bignum 32032049084104101. Our 51-character strings, of course, yield longer bignums, with about 51*3 digits each (minus 1 if the first character is a blank!) We won’t show you the bignums for these two lines, but you are welcome to compare your findings on Ed Discussions.

Now, you pass those bignums to “encrypt” and they yield the following two bignums:

3220487666737937614884785254841020111661727275946715212893264848451743012978853378
665534875079311067297097752983602964461327272887729543786636781964673567

2175470433948046271847840877193998313227407722846015136888068311851877229153457560
848005416823127493571017510841417042029597213105566588302576600310172698

Later, if you were to run bignum d and input these two lines, the decryptor will regenerate the exact same 51 character strings. Append the second one to the first one, yielding a 102-character string. Now strip 3 characters from each end (the line numbers) and remove trailing blanks, and you should get:

“The cake is a lie!”

… which is to say, we have encrypted and then decrypted our input!

Ken’s code is kind of slow! Your hand-coded version probably is too.

Although Ken is a performance-oriented developer, he deliberately wrote his original version of bignum in what we might call a “very basic” way. The code is proper C++ but lacks a bunch of things we learned to do in class. In fact, he even made some questionable design choices and omitted a few obvious things that could speed his solution up. Your job in part 2 will be to (1) time your program or his version of the program, (2) use gprof or other tools to understand where that time is spent, (3) use the timer and gprof output to identify which of a few suggested improvements would be helpful, (4) improve it as much as you can.

In fact Ken also wrote a better version (our internal sample solution). It fixes all of the issues and runs quite a bit faster. So Ken isn't a total incompetent! But the idea here was to have you improve a working but somewhat "problematic" piece of code. If you do this with your own code as a starting point, same idea, but the optimizations might differ.

We will provide you with a part of the poetry from Robert Frost to work with as a sample test data, but beware: Amdahl's law applies. In fact, RSA is an example of a task that can't easily be parallelized or optimized. With the original, non-improved program, you will find that *encrypting* 100 or so lines of poetry takes about 30-50 seconds. But decryption is nearly 5x slower; combine the two and you may need to wait for 5 or more minutes if you pipe "encrypt e < one-liners" into "encrypt d". This is unreasonably long to wait. So while searching for speedups, focus on a one-line test case, like:

```
time echo "The cake is a lie" | bignum e | bignum d
```

Once you have this running really fast, *then* perhaps you should test on the whole file – and while the test runs, get a bite to eat!

What to do.

1) Profile bignum e < one-liners > encoded, which will create a file "encoded" of encrypted output and also leave you with a gmon file for this task. You can use just a few lines from one-liners if you find this too slow; profiling a run on the whole file isn't necessarily "better" than profiling a run for a single line, or two or three lines.
2) Run gprof bignum (it automatically knows how to find gmon.out), sending the output into "more" for you to study line by line.
3) Repeat, but now profile bignum d < encoded. You'll see the poetry line by line on the console, and then at the end, you can run gprof bignum again, and this time you get a profiler output for the second run. Again, you study it line by line.

Some questions we want you to answer.

Q1) Whichever version you work with, where is the majority of time spent? We mean, "in what methods?" Both encryption and decryption call modular exponentiation, but this method calls other Bignum methods. List the top four methods in terms of total time spent computing in that method, or in things that it calls. Is this the same list for both encryption and decryption? If the answer involves a C++ std library method that has a very long name, try to understand what higher level operation it corresponds to. For example, a call to the emplace_front() operation (a std::vector method) might actually show up in gprof with a lot of detailed template type information that wasn't specified in the Bignum code per-se, yet it might be important in terms of cost, which would be a warning that Ken is doing something unexpectedly time-consuming that perhaps could be done in a cheaper way! We want to know which lines of code were costly, not all that type information "generated" by the C++ template expansion!

Q2) Variant on Q1. Same as Q1, but now try to use the gprof information to gain an insight into which *algorithms* might be root causes of the performance we see for the two cases (encryption and decryption). The point here in Q2 is that even if “subtraction” is called a million times and uses most of the compute time, there is also a question of *why* subtraction is being called quite so frequently. You might realize that actually, most of those million calls could have been avoided – and this would tell you that the payoff is to change the algorithm, not the implementation of subtraction!

Q3) A few ideas

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.

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!”

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?

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.

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.

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?

Ideas we will *not* be implementing:

7. The recommended approach to handling long lines by just encrypting every line as two chunks basically doubles the cost of encrypting a short line. It would be faster to encrypt the whole file in even sized chunks of, say, 50 characters at a time. **We won't try this because it changes the behavior of the whole application** – we are only considering line by line solutions in hw2. Similarly, it wouldn't be appropriate to output one line if the input was short, and two if the input was long: that sort of change reveals something about the length of the input.
8. Caching "final results". Ken wondered if RSA somehow might repeat the same Bignum addition or multiplication or division operations again and again. If so, caching the results of operations and checking to see if the answer is known could be a useful optimization. (Note: this is **not** the same as suggestion 6. Reread the two side by side if at first they seem to be talking about the same idea, until you see why they are actually different suggestions). Anyhow, Ken checked to see if the exact same Bignum operations occur during long-running computations, and discovered that no, this doesn't occur. You can definitely precompute and cache other things, but don't waste time trying to cache results for any of the Bignum operations.

In summary… your mission is to explore ideas (1-6). You do not need to implement an idea that has no impact, if you can give a convincing argument that the idea couldn't possibly matter. However, we think that you actually will find some things that need to be done on all four tasks. We may deduct some points if you say something couldn't be important but your reasoning is incorrect or not supported by your own measurements.

Don't test on the full file until you have a threaded version working. The complete list of poetry is a little slow to encrypt and decrypt this way, and you might do better testing with just two or three lines. We plan to check your code on a 20-line test file. For the performance competition we may use a longer input file, depending on how fast the fastest competitors turn out to be.

What to hand in. Your upload should consist of a zip file containing (1) a CMake file to build the program; (2) source code; (3) a PDF writeup with your findings from the various optimization steps including: a short description of what you did to answer that question, any relevant small summaries from the gprof analysis (do NOT include the full gprof output).

We are especially focused on your report of time improvement after the optimization. You should justify this by showing us excerpts from the new gprof output and you should explain why that gprof output allowed you to conclude that the optimization had an impact. We are looking for concrete evidence that you figured out where time is spent and can explain why this is the case, but also *that you have learned to use Linux tools to validate your hypothesis.* Even if your theory about inefficiency was correct, without the before-after use of timing data you would lose points.

We plan to test that your code in fact actually works. We want to see your reasoning and your mastery of understanding where performance improvement opportunities "really" could be found, and your way of enhancing the code. This homework is our chance to see how you think about performance.

Grading. We test for both correctness and performance. If your result is correct but exceeds the expected time of completion, there will be points taken off. You may not assume the one-liner inputs, as we could test on any valid inputs.

Using Ken's solution as a starting point. For people who want to start with Ken's solution, you need to implement the multithreading optimization on the program in order to get full credits. If you don't implement multithreading optimization but other optimizations, we will give partial credits, but not the full credits. You will need to attend the recitation on **Nov 15th (Friday)** this week when we will talk about how to use multithreading in the program, and we will send the starter code for people who choose to use Ken's solution and optimize based on that.

Leaderboard. Like hw3 part2, there will be a leaderboard for this assignment. The solution that runs the fastest will get 10 points extra credit, and the following top3 programs will get 5 points extra credit.