

Project 1

Analysis of Timing Attacks on Modular Exponentiation

Task W0

For cryptography applications such as the Diffie-Hellman Key Exchange and RSA public/private key implementation, an efficient modular exponentiation algorithm is crucial. Modular exponentiation is exponentiation performed over a modulus ($a^b \% n$). Typical data primitives like int and long int would not suffice in a naive implementation of this formula because the size of a^b grows extremely fast; another issue with the growing size of numbers is that it causes significant slowdown in the calculations. To circumvent the first issue, in my mock simulation, the inputs are stored as a custom implementation of Java's BigInteger class – "LargeInt" (very creative). The inputs/outputs are all created using byte arrays, and BigInteger is used to output the values of the byte arrays to readable numbers for analysis. RSA recommends a size of 1024 bit exponent, which is about 300 digits; in reality, for utmost security, experts no longer recommend a 1024bit exponent but rather a 2048bit one. For this demonstration, I'm going to use 1024bit exponents to speed up the numerous tests I'm performing on my low-end PC. This will still work because due to the nature of ModExp, differences in the complexity of bits in the exponent will cause run-time differences. For the inputs, there will be three separate tests: firstly a baseline consisting of random 1024 bit exponents, secondly simple exponent tests (think 100 . . . 00), and finally complex exponent tests (think 111 . . . 11). To calculate the time to compute, Java's built in system clock, will be used to get the run-time down to nanoseconds, and these results will be written to an output file for further analysis. My implementation of ModExp will also be ran against the Java BigInteger modPow to confirm my results are accurate. This comparison happens separately outside of the time calculation and does not impact results. For our expected outputs, we would assume a fast implementation that makes use of a "math hack" would be faster than a naive implementation as seen on slides lecture 7. More analysis on the difference between the two algorithms will be provided in their respective sections (don't want to spoil too much!).

Task W1

Like stated briefly before, Modular Exponentiation algorithms are used widely in Diffie-Hellman Key Exchange and more importantly, RSA public/private key. For the purpose of this analysis, RSA will be used as the main example. A secure algorithm is crucial to ensure the resistance of RSA against timing attacks that attempt to steal the private key. If a computer is able to compute modular exponentiation on large bit exponents, the process to brute-force the exponent is of the magnitude $O(2^n)$. This assumption that the private key is "uncrackable" is the reason RSA is widely used, but since computers aren't "perfect" in a sense, a well-formulated timing attack could deduce the exponent based on the time it takes a computer to perform this modular exponentiation. Modular exponentiation is used in RSA encryption and decryption; the encryption uses the public key while the decryption uses the private key. The private key is what many attackers are sought after as this will fundamentally break the encryption standard. For instance, if an attacker were able to distinguish a simple exponent from a complex one, that knowledge could be used to learn information about the private key and eventually craft their own. Since RSA is widely used, this problem of modular exponentiation in relation to timing is a serious issue; according to Wikipedia, many trusted implementations of RSA use some sort of diffusion to protect its timing and ensure that no information gets leaked.

Task C2

LargeInt includes various helper methods to assist in basic arithmetic. The important pieces for the ModExp algorithm include **gradeschool multiplication**, which utilizes repeated additions, and **basic division**, which utilizes repeated subtractions. Various helper methods to assist both of these include: negate, trimLeadingZeroes, shiftLeft, and shiftRight.

A basic rundown of the helper methods:

- negate – negates the input and is used for subtraction
- trimLeadingZeroes – removes leading 0's before the most significant byte
- shiftLeft – extend the byte array by set amount and then shift values to the left
- shiftRight – trim the byte array by a set amount and then shift values to the right

Gradeschool Multiplication (gradeschool())

```
//Actual gradeschool algorithm using partial products and remainder
for (int i = 0; i < firstLarge.getLength(); i++) {
    for (int j = 0; j < secondLarge.getLength(); j++) {
        //Creating partial product storage
        temp = new byte[firstLarge.getLength() * 2];
        //Retrieving the current bytes currently calculating the partial product
        firstCurr = firstLarge.getBytes(i);
        secondCurr = secondLarge.getBytes(j);

        //Only work with positive numbers for mult
        //Adding 256 will get the correct 2s complement representation of negative bytes (-128 -> 127)
        if (firstCurr < 0) {
            firstCurr += 256;
        }
        if (secondCurr < 0) {
            secondCurr += 256;
        }
        //computing the partial product
        int mult = (firstCurr * secondCurr);
        //deciding where to put the partial product and the remainder (if there is one)
        int location = i + j + 1;
        temp[location] = (byte) (mult & 0xFF);
        location -= 1;
        //shifting right to include only the remainder portion of the product (overflow byte)
        temp[location] = (byte) ((mult >> 8) & 0xFF);
        //add the partial product to the product
        product = product.add(new LargeInt(temp));
    }
}
```

The gradeschool algorithm first converts the 2's complement representation of the number in the byte array to an int, then computes the partial product and overflow. Since the maximum size of the product will be $\times 2$ the largest length of the one of the inputs, some of the product will not fit in a single byte in the array. Therefore it will have to be shifted to the next block. An issue with this implementation that took significant tests and analysis to figure out, was the growing number of leading 0s. Because it assumes worst case scenario and thinks the product will be $\times 2$ the length, it creates an array with a lot of leading 0s if not the case. If input A were 1000 and input B 2, it would create the length of the byte array to be eight spaces long even though the output would only take up four spaces. When doing repeated multiplications, i.e. exponentiation, this number will grow exponentially. To circumvent the leading 0s, a trim method is enforced after every multiplication; the trim will still leave one leading 0 if the number were positive to avoid confusion between positive and negative numbers. (This issue was seriously mind-boggling because the way I output the numbers was converting the byte arrays to BigIntegers for easy printing but that would not output the leading 0s)

Basic Division (division())

```
//Algorithm
//for n bits in dividend (same amount we shifted the divisor)
for (int n = 0; n < dividend.length; n++) {
    //shift the divisor right one
    divisor = shiftRight(divisor, 1);
    LargeInt partialDivisor = new LargeInt(divisor);

    //while we can fit the current divisor into the remaining number
    while (!remainderLarge.subtract(partialDivisor).isNegative()) {
        //subtract by current amount of divisor
        remainderLarge = remainderLarge.subtract(partialDivisor);

        //calculating the correct amount of shifts for where the quotient should go
        //ex: 700/25 on step 700/250 the "2" that 250 goes into 700 should go at position 0 in the quotient array
        //next step 200/25 the "8" that 25 goes into 700 should go at position 1 in the quotient array
        //result 0-2 1-8

        //calculate the correct shifts by doing the length of dividend - n amount of shifts so far
        byte[] partialQuotient = new byte[1];
        partialQuotient[0] = 1;
        partialQuotient = shiftLeft(partialQuotient, (dividend.length - n - 1));

        //add the partial quotient to the total quotient
        quotientLarge = quotientLarge.add(new LargeInt(partialQuotient));
    }
}
```

The basic division algorithm returns the quotient and remainder in the form of an array. First, it normalizes the size between the divisor and dividend, and then manipulates the shifted divisor to figure out the partial quotient. The variable remainderLarge is initialized to the dividend. To calculate the partial quotient, you subtract it from the remaining dividend and keep track of the amount of subtractions made. Whatever value is left in the remainderLarge is the remainder. If I were to want to test my implementations on 2048 bit numbers, I would first implement a faster division algorithm that doesn't rely solely on repeated subtractions because this method is not as fast as it could be. The main slowdown in modular exponentiation algorithm is the modulus calculation. (Computers are not best friends with division).

ModularExp (modularExp())

```
//implementation of pseudo code presented in lecture 7
public LargeInt modularExp(LargeInt exponent, LargeInt modulus) {
    LargeInt base = new LargeInt(this.getValue());
    LargeInt result = new LargeInt(ONE);
    BigInteger tracker = new BigInteger(exponent.getValue());

    //for k=0 to b-1 (little endian notation so actually reading the MSB first)
    for (int i = 1; i <= tracker.bitLength(); i++) {
        //if k bit == 1
        //System.out.println("CURRENT = " + (tracker.testBit(tracker.bitLength() - i)));
        if (tracker.testBit(tracker.bitLength() - i)) {
            //r = (r * base) mod n
            result = (base.gradeschool(result)).division(modulus)[1];
        }
        if (tracker.bitLength() - i > 0) {
            //r = r^2 mod n
            result = (result.gradeschool(result)).division(modulus)[1];
        }
    }

    return result;
}
```

The basic modular exp algorithm is built off of the pseudo-code provided in lecture 7. It follows it basically line by line utilizes BigInteger testBit and bitLength. The arithmetic operations are still calculated using my implementations.

Task C3

For testing three different simulations were ran: Random, Simple, and Complex. For all tests, the base was set to a random 5 digit base (this remained the same throughout the tests), and a random 512 bit modulus. Google recommends a public key of at least 512 bits. The reason for the base being 5 is arbitrary as changing the base should not change the results since the exponent complexity has a more significant impact. Each of the tests are ran with a sample size of 20. Due to the increasing computational complexity for some of the larger exponents, 20 should suffice. All the input files are included in the "input" folder while the output is recorded using java filewriter in "output" folder. To calculate the difference in time required to execute the algorithm, the current system time is recorded before and after, and the difference in those time is written to the output file.

Random

```
// Method to generate bit strings
// Parameter length is how many bits long
static String generateBitString(int length) {
    String toReturn = "";
    for (int i = 0; i < length; i++) {
        //assuring the most significant bit is a 1 and therefore the resulting string will have length 1024 bit
        if (i == 0) {
            toReturn += one;
        } else if (randomGen.nextBoolean()) {
            toReturn += zero;
        } else {
            toReturn += one;
        }
    }
    return toReturn;
}
```

The method pictured above, generateBitString, will randomly generate a bit string of desired length. With parameter length = 1024, Random will return a bit string of length 1024; the most significant bit is guaranteed to be set to 1 to ensure the length will always be 1024. The initialization code runs this for a total of 20 times and outputs the random bit strings to an input file. The random samples were generated using Java random boolean, which ideally should choose 0 or 1 50% of the time. The reason for included a random sample is to represent a baseline average for computing the mod exp algorithm.

To generate the random input file:

timingAnalysis 20 generate FALSE

*Note this will override the current random input file if one is present

Simple & Complex

For simple and complex input, there are two respective input files with a majority of the individual bit sets to 0 for simple, and the opposite for complex. Again, RSA is OK with a private key of 1024. In total, there are three input files that can be ran via command line arguments.

For simple, each of the bit strings start with a 1 followed by 1000 0s then random permutations of the remaining bits. For complex, each of the bit strings start with 1000 1s followed by random permutations of the remaining bits. The stark difference in the exponent bit strings is an extreme example to highlight the difference in computational complexity when doing modexp. Obviously in applicable sense, there won't be such striking polarization between exponents you are analyzing, but this example is meant to highlight that there is such difference.

To run the simulation, the input is as followed:

timiningAnalysis #ofTests(20 maximum) typeOfTest(random, simple, complex) FALSE ←

FALSE signifies the simple approach

Results

For each test, the time in nanoseconds elapsed is recorded. At the end of the simulation, the average is computed and written to the last line of the file. The averages are as followed:

Simple Modular EXP

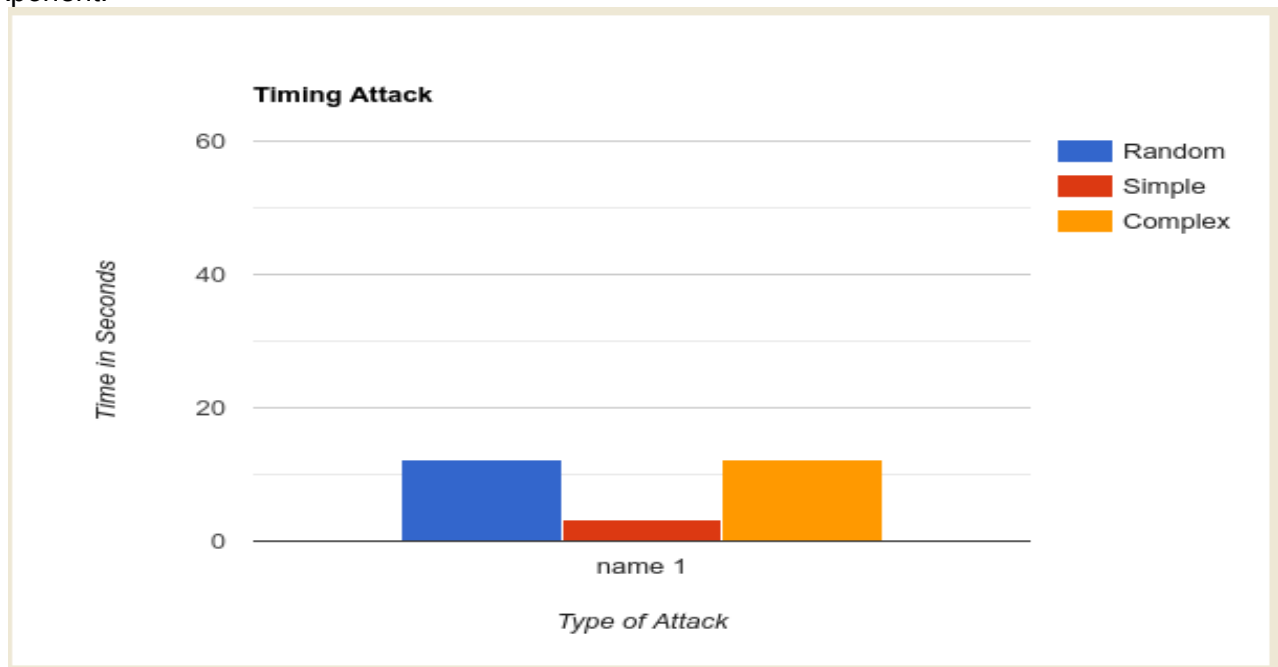
Type	Average Time in Seconds
Random	12.2732774122
Simple	3.286510076
Complex	12.205368668

Task W4

In the modular exp algorithm, the current bit being set plays a big impact in the necessary calculations. If the current bit is set, the result is multiplied by the base and mod n. This is seen in the pseudo code:

```
if currBit isSet
    result = (result * base) mod n
if not last bit
    result = (result*result) mod n
```

So depending on the amount of bits set in the exponent, the amount of calculations will differ. An exponent, for example "1111," would be more computationally taxing than "1000." This is because the number of calculations required for the first String would be equal to seven while the second would require five. Note this is only for a four bit exponent and the difference is already two. For 1024bit exponent, this difference grows tremendously. As seen in the graph below, the time complexity of "simple" vs "complex" exponents varies *a lot*. Not to sound redundant, but like stated previously the words "simple" and "complex" don't do enough justice to them actually being polar opposite calculations in terms of difficulty. But with there being that much time difference, this proves there is potential to mount a timing attack and looking at the variances in time depending on the bits set in the exponent.



In RSA, this attack would be used to try and reduce the private key d in the equation $M^{ed} \pmod n$. As a refresher, d – private key, e & n – public key, and m – message. So in my test simulation, the complexity of the message (base) would not impact the results. Since e and n are available to anyone who wants to send a message, the attacker would already have access to them. Ideally, this would be used in a scenario where there isn't much internet latency that would skew the data, and in true ideal scenarios, the attacker could present this attack in an offline fashion that wouldn't need the internet. The type of attack is chosen plaintext where the attacker has access to multiple plaintext-ciphertext pairs and knows which correspond to the other. In actuality, the fact that this is a chosen plaintext attack does not have much impact because the attacker can make as many plaintext-ciphertext pairings as they want due to the nature of RSA encryption. In this encryption, the formula is $M^e \pmod n$ and e/n are the public key. An assumption is also that the server would not bounce the attackers request because in order to execute this attack, many attempts need to be made. With a small test sample, it becomes increasingly harder to use statistics to predict the next bit. The attacker would need to perform a similar series of simulation runs like I did in my test example;

they would need to collect the timing in the decryption/signing step of the process. After collecting enough samples, statistics can be used to approximate the likelihood that the exponent was split into 2 and a multiplication was performed. After predicting a bit with enough confidence, use that bit and repeat for the following bits, growing the bit string you are crafting. With enough patience and time, the bit string will eventually grow to that of the RSA private key. The main statistic necessary is variance. By computing the variance between locally executing the mod exp algorithm and the person you are communicating with, you can see if the variance increases or decreases within rounds since they are independent. In simpler words, try and guess the first x bits, then use those bits to guess the next x bit(s). Rinse and repeat until finished.

Task C5

To implement a more constant time algorithm, I implemented a fast recursive method of modular exponentiation that manipulates the exponent until it is a power of 2. As stated in paper (1), the number of steps the algorithm will take is $O(\log \text{exponent})$. The math behind it is as follows: $b^e = b^{(e/2)} * b^{(e/2)}$. So by calculating $b^{(e/2)}$ once, the algorithm should benefit. The recursive implementation ensures this.

Fast Modular Exponentiation (modularExpFast())

```
//The idea is to split the modular exponentiation once the exponent is a power of 2 (subtract 1 if not negative)
public LargeInt modularExpFast(LargeInt exponent, LargeInt modulus) {
    LargeInt LI_ONE = new LargeInt(ONE);
    LargeInt LI_TWO = new LargeInt(TWO);
    //Base Cases
    //Exponent == 0
    if (exponent.isZero()) {
        return LI_ONE;
    }
    //Exponent == 1
    else if ((exponent.subtract(LI_ONE)).isZero()) {
        return (new LargeInt(this.getValue()));
    }
    //Exponent % 2 == 0 (even)
    else if (exponent.division(LI_TWO)[1].isZero()) {
        //Return modExp(base*base%n, exp/2, n)
        return (this.gradeschool(this).division(modulus)[1]).modularExpFast(exponent.division(LI_TWO)[0], modulus);
    }
    //Otherwise exponent is not base 2
    else {
        //Return base * modExp(base, exp-1, n)%n
        return this.gradeschool(this.modularExpFast(exponent.subtract(LI_ONE), modulus)).division(modulus)[1];
    }
}
```

If the current exponent is even, the resulting calculations will be much more taxing than if the exponent were not. For a square-and-multiply algorithm like this one, the number of squaring operations is fixed. In other words, the amount of times you reduce the exponent by 2 is dependent on the complexity of the exponent. A more simpler exponent will require less reductions. This is seen in the return statements. In pseudocode:

```
If exponent == even
    return modExp(base*base%n, exp/2, n)
else
    return base*modExp(base, exp-1, n)%n
```

The first return statement splits the equation into the $b^{(e/2)} * b^{(e/2)}$ while the second return statement splits the equation into the form $b * b^{(e-1)}$. The second return statement being much more computationally simpler. The average run-time results reflect this sentiment as seen above.

Bugs/Limitations

I tried very hard to get an implementation of Montgomery multiplication to work, but I could never get the correct values to return – not even for basic numbers. The amount of time it took to implement the LargeInt class from scratch was a lot longer than I anticipated. The amount of bugs I encountered when implementing the basics like multiplication and division had me extremely flustered. If I were to start the project over, I would rely on Java BigInteger only as I later learned that it does not have a built in Montgomery multiplication for large bit modpow; I would use the standard library to at least get a working version. Even though this advanced algorithm is the main portion of this project, I feel like even though I didn't get a successful implementation of Montgomery to work, I still learned a lot about it through the papers.


Task C6

To give a fair test for my fast modular exponentiation implementation, I followed the same procedures as the other test. Instead of generating a new random exponent file, I opted to use the same one to see the differences. The same 20 sample size test for random, simple, and complex were utilized. The input sizes were not changed: random 5 digit base, random 512 bit modulus, and the exponent derived from the same input files. If you wish to replicate the results the following command line syntax is used:

```
java timingAnalysis #ofTests typeOfTest TRUE ← TRUE signifies the fast algorithm
```

The results:

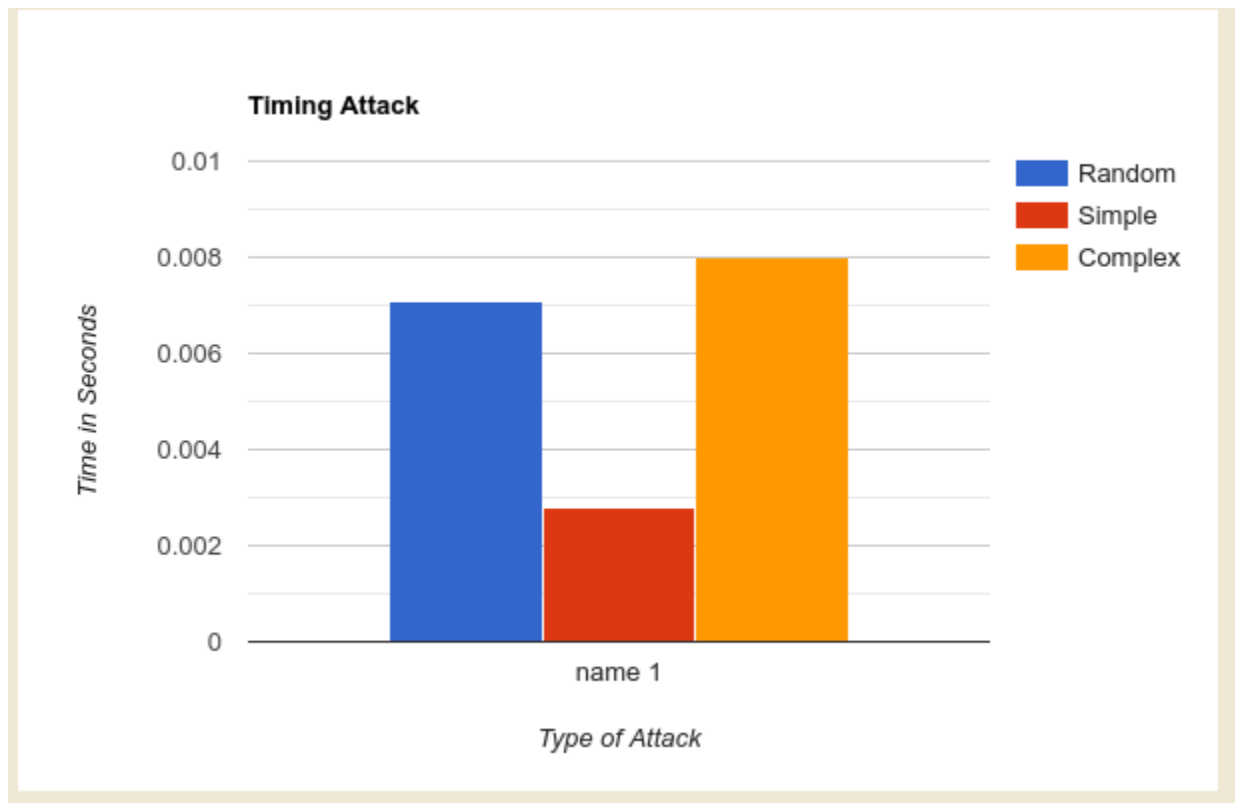
Fast Modular EXP

Type	Average Time in Seconds
Random	32.278880538
Simple	3.2662311623
Complex	58.776296375 

Task W7

The biggest shocker is how this algorithm polarizes the results even more. I think the largest drawback for this algorithm is the complexity needed for the if statements. The amount of conditions is too much when dealing with the large numbers which is why the run-time suffered significantly. For example, the statement `|if(exponent.division(LI_TWO)[1].isZero())|` has to perform a messy division to see if the exponent is even. With a sufficient division algorithm, this implementation should follow the $O(\log \text{exponent})$ run-time and not be susceptible to the information leakage of complexity in bits because the algorithm wants to normalize it into a base of 2. This is similar to how a Montgomery implementation would've wanted to transform the number. Computers work best with base of 2. The recursive implementation would need a lot more memory usage than the simple implementation due to the growing stack of recursive calls. After encountering this issue, I tested my algorithm using java BigInteger. If you wish to repeat the results, I added another command line argument "Big" that means you want to test the fast algorithm using BigInteger instead of LargeInt:

```
java timingAnalysis # type false true ← last argument should be true for BIG
```



From the above bar graph from BIG test, there is still a clear difference between the simple and complex values. But remember, an exponent that is truly "simple" is extremely unlikely to actually happen. The interesting result is the closing gap between random and complex numbers. The algorithm seems to be standardizing on timings between completely random numbers and complex. The modified algorithm to accommodate BigInteger can be found starting line 511 in largeInt.java. There is no difference between that and largeInt implementation other than difference in helper method names.

Task W8

I still believe a timing attack could be carried out on this modified algorithm due to observed difference between random and simple tests. With a better way of generating *good* tests, I think the algorithm deserves to be further analyzed and see if this result was a fluke.

Complex runtime analysis:

1024-bit exponent with all bits set:

107150860718626732094842504906000181056140481170553360744375038837035105112493612
249319837881569585812759467291755314682518714528569231404359845775746985748039345
677748242309854210746050623711418779541821530464749835819412673987675591655439460
77062914571196477686542167660429831652624386837205668069375

Log(exp) = **699.98693920514**

1024-bit exponent with only MSB bit set:

535754303593133660474212524530000905280702405852766803721875194185175525562468061
246599189407847929063797336458776573412593572642846157021799228878734928740196728
388741211549271053730253118557093897709107652323749179097063369938377958277197303
8531457285598238843271083830214915826312193418602834034688

Log(exp) = **692.45403337939**

The difference in the log of these two variable inputs is very tiny. I think more thorough tests need to be tried that I didn't have to the time to try. Because like stated previously, the paper stated that the run-time should be dependent on the length of exponent rather than complexity of exponent because the steps look at whether it is even rather than individual bits.

A side-note about the recursive implementation, is that it is probably more susceptible to cache-attacks because of the increased amount of memory usage required.

I think a good attack against this implementation would be a timing & cache double attack that would try to use the best of both worlds in this scenario. Any and all algorithms are susceptible to power attacks given the attack has physical access to the machine. In reality, I would say this algorithm is fairly safe given the proper security is followed at the physical location.

Conclusion

Even though things did not work out my way for me most of the time, I actually had a really good time doing this project. I enjoyed the freedom for once, and I hope my code is sufficient enough!

Sources

1. <http://www.cs.ucf.edu/~dmarino/progcontests/modules/matexpo/RecursionFastExp.pdf>
2. https://en.wikipedia.org/wiki/Modular_exponentiation#cite_note-1
3. https://en.wikipedia.org/wiki/Montgomery_modular_multiplication
4. https://link.springer.com/content/pdf/10.1007%2F3-540-36400-5_22.pdf
5. <https://gmplib.org/~tege/modexp-silent.pdf>
6. <https://www.nayuki.io/page/montgomery-reduction-algorithm>