International Journal of Research in Engineering and Science (IJRES)
ISSN (Online): 2320-9364, ISSN (Print): 2320-9356
www.ijres.org Volume 10 Issue 2 ǁ 2022 ǁ PP. 53-71

Experimental Analysis of Algorithms to Enhance RSA

PRATHAM SAHAY
Vellore, Tamil nadu

PRABALJIT WALIA
Vellore, Tamil nadu

ABSTRACT
The RSA encryption algorithm, which was named after its 3 founders, Adi Shamir, Ron Rivest and Leonard Adleman, was one of the first public key asymmetric encryption systems. It was one of the first viable public key crypto systems extensively used to secure data channeling. This system is based on a public key and private key asymmetric encryption dialect, where the sender of a message encrypts the message with the public key and the received decrypts the encrypted message with a private key. The public key is visible but the private key is kept confidential. The Encryption is very hard to break because of two very large prime number which are almost impossible to retrace. Key generation excessively depends on detecting cofactors and using the modulo operation. In this project, we will be using different algorithms to accelerate certain key generation steps and explain its reduced time complexities.
KEYWORDS: *Modulo Inverse, Encryption and Decryption, Prime Numbers, Euler toitient, Exponent, Base and Divisor.*
SOFTWARE: *All calculation, graphs and tables are implemented using python's latest version on Google Collaboratory and libraries like matplotlib, time, pretty table were used.*

Date of Submission: 02-02-2022 Date of acceptance: 16-02-2022

I. INTRODUCTION

There was a time when RSA was carried out using symmetric public key encryption but, now a days the algorithm is enhanced by using a public key for encryption and private key for decryption process. The private key and the public key are modulo inverse of each other and the general method to calculate the private key is very long so, we have used an existing Extended Euclidean Algorithm and analyzed the execution time with the general method. The Encryption and Decryption process involves Exponential Modulo which again takes a long time if very large prime numbers are used which may take several hours so we have to use Modular Exponentiation Algorithm and compared its time with the general method, to verify the above algorithms work properly we have implemented the image encryption and decryption and results are verified. The Karatsuba fast multiplication is a new way to multiply very large numbers that too very fast and effectively as compared to general method multiplication, so we have verified the execution time by plotting it against the number of digits and calculated its efficiency.

II. METHODOLOGY

2.1 MODULAR EXPONENTIATION ALGORITHM

1.1)

A) OVERVIEW

The method of Modular Exponentiation is a very important algorithm not only in the field of the Encryption and Decryption but many other fields like competitive programming, in handling large data set where the naïve methods prove to be ineffective and its effectiveness is based on the fact that it breaks the exponent calculation into smaller parts and then the modulus is calculated which works faster. The time complexity of the algorithm is less than the time complexity of the general method and ahead in the paper we will observe that the efficiency of execution is a bit less in case of very small numbers as the number of digits increases the efficiencies increases drastically to a very large extent and hence in the case of encryption and decryption where the public key and the private key are 1024-bit it's a very useful algorithm for example in an image encryption decryption process there are 1500*800 points in a list and each point has another list of three values r, g, b which multiply to 1500*800*3 values so you can see so many data's are to be encrypted and decrypted, in such cases the naïve methods will take a long time hence modular exponentiation method comes to the rescue. Further in the paper we will observe the graphical analysis of the algorithm and compare it with the naïve methods.

B) MATHEMATICAL BACKGROUND

Consider three very large numbers x, y, n and we want to calculate (x^y% n) , So, we can represent y in terms of its binary digits,

- $y = (b_n*2^n) + (b_{n-1}*2^{n-1})+\dots\dots\dots(b_1*2^1)+(b_0*2^0)$

where b_n = 0 or 1

- $y = \sum_{i=0}^{n} b_i*2^i = (b_n*2^n) + (b_{n-1}*2^{n-1})+\dots+(b_0*2^0)$ (1)

- $x^y = x^{(bn*2^\wedge n)+(bn-1*2^\wedge n-1)+\dots\dots\dots(b1*2^\wedge 1)+(b0*2^\wedge 0)}$

- $x^y = (x^{(bn*2^\wedge n)})*(x^{(bn-1*2^\wedge n-1)})\dots..*(x^{(b0*2^\wedge 0)})$

- $x^y = \prod_{i=0}^{n}(x^{bi*2^\wedge i})$ - (2)

- Now when b_i = 0, then

$x^{(0*2^\wedge i)} = 1$

- And when b_i = 1, then

$x^{(1*2^\wedge i)} = x^{2^\wedge i}$

- Thus, we can show that

$x^y = \prod_{i:bi\neq 0}(x^{2^\wedge i})$ - (3)

- Now we assume $a_i = x^{2^\wedge i}$ where $i \geq 0$

Now, $x^y = \prod_{i:bi\neq 0}(a_i)$ - (4)

- We also see that $a_0 = x^{2^\wedge 0} = x$

and $a_i = x^{2^\wedge i}$ also $a_{i+1} = x^{2^\wedge(i+1)} = x^{(2^\wedge i)*(2^\wedge 1)}$

which is $a_{i+1} = (x^{2^\wedge i})^2 = (a_i)^2$

- Hence, we can conclude that the terms x_0, a_1, a_2 … are related that is to obtain the next term we just have to calculate the square of the preceding term.
- i.e $a_{i+1} = (a_i)^2$
- Since it is a modulo operation in each term, we take the modulo.
- Since (a*b)%n = (a%n)*(b%n).

C) ALGORITHM

STEP1: We check iteratively one by one the binary digit of the exponent and if it zero we go to step 3 or else we go to step 2

$y = (b_n b_{n-1} b_{n-2}\dots\dots\dots b_1 b_0)$

STEP2: If the binary digit of the power is one then we implement modulo multiplication between answer and the base and again carry out modulo operation and go to step3

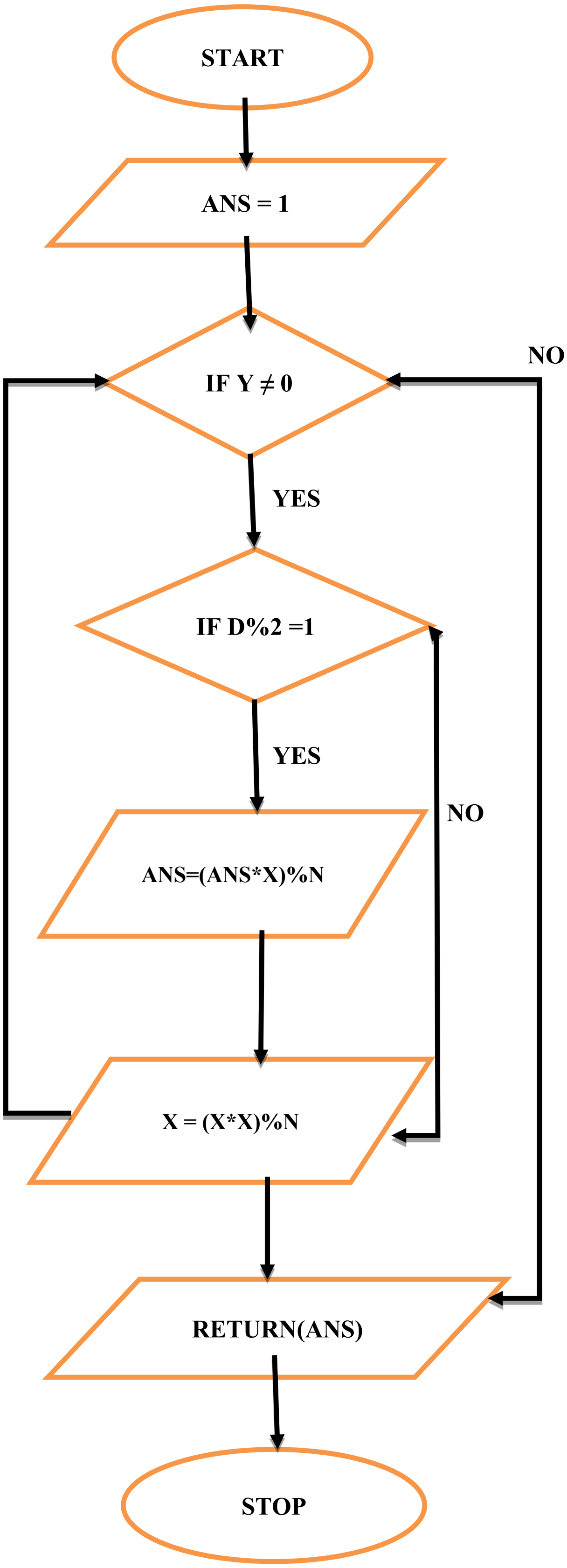
answer = (answer*x)%n

STEP 3: In this step we do the modulo square of the ith base to get the (i+1)th and again carry out the modulo operation x = (x*x)%n

STEP 4: Right shift the bit of the exponent and then carry out the steps until the exponent becomes zero.

y>>1 and perform all above steps till $y \neq 0$

D) FLOW DIAGRAM



2.2 EXTENDED EUCLIDEAN ALGORITHM

A) OVERVIEW

Extended Euclidean algorithm is an extended form of the Euclidean Algorithm to find greatest common divisor of two numbers when applied with the Bezouts theorem. In general, if the greatest common divisor of two number m and n is k then using the extended Euclidean algorithm, we can represent the numbers m and n in terms of k its gcd i.e. $m*x + n*y = k$ and we can obtain different pairs of x and y for the equation. When the number m and n are co-prime i.e. gcd (m, n) = 1hence x is the modular multiplicative inverse of (m) modulo (n) and y is the modular multiplicative inverse of (n) modulo(m). The above algorithm is very useful when we have to calculate the public and the private keys for Encryption and Decryption process because the naïve method to calculate the keys becomes infeasible when the number becomes too large. Further in the paper we will visualize graphs and efficiencies for different sets of prime numbers and draw conclusions.

B) MATHEMATICAL BACKGROUND

To calculate the gcd(m, n) we can find it using the Euclid algorithm by calculating the sequence q_i, r_i, a_i, b_i for $i \geq 2$ and $r_0 = m$, $r_1 = n$ and after every operation we increase the value of i. We calculate the values of q_i, r_i, a_i, b_i such that $r_{i-2} = r_{i-1}*q_i + r_i$-(1), such that $0 \leq r_i \leq r_{i-1}$ using the Euclid Algorithm. We can write r_i as linear combination of m and n i.e $r_i = a_i*m + b_i*n$ -(2),

- from (1) and (2) we obtain
- $= r_i = (a_{i-2}*m + b_{i-2}*n) – (a_{i-1}*m + b_{i-1}*n)*q_i$
- $= r_i = (a_{i-2} - a_{i-1}*q_i)*m + (b_{i-2} – b_{i-1}*q_i)*n$ -(3).
- Comparing (2) and (3) we obtain
- That $a_i = a_{i-2} + a_{i-1}*q_i$ and $b_i = b_{i-2} + b_{i-1}*q_i$.
- Since $m = r_0 = a_0*m + b_0*n$, $a_0 = 1$ and $b_0 =0$.
- Also, $n = r_1 = a_1*m + b_1*n$, $a_1 = 0$ and $b_1 =1$.
- Now when the r_{i-1} becomes 0 we stop and suppose it is the i^{th} operation then $r_{i-1} = 0$ which means that the $gcd(m, n) = gcd(r_0, r_1) = gcd(r_1, r_2) = \ldots\ldots\ldots\ldots gcd(r_{n-2}, r_{n-1}) = gcd(r_{n-2},0) = r_{n-2}$.
- Hence the linear combination we obtain,

$gcd(m, n) = r_{n-2} = a_{n-2}*m + b_{n-2}*n$ -(4).

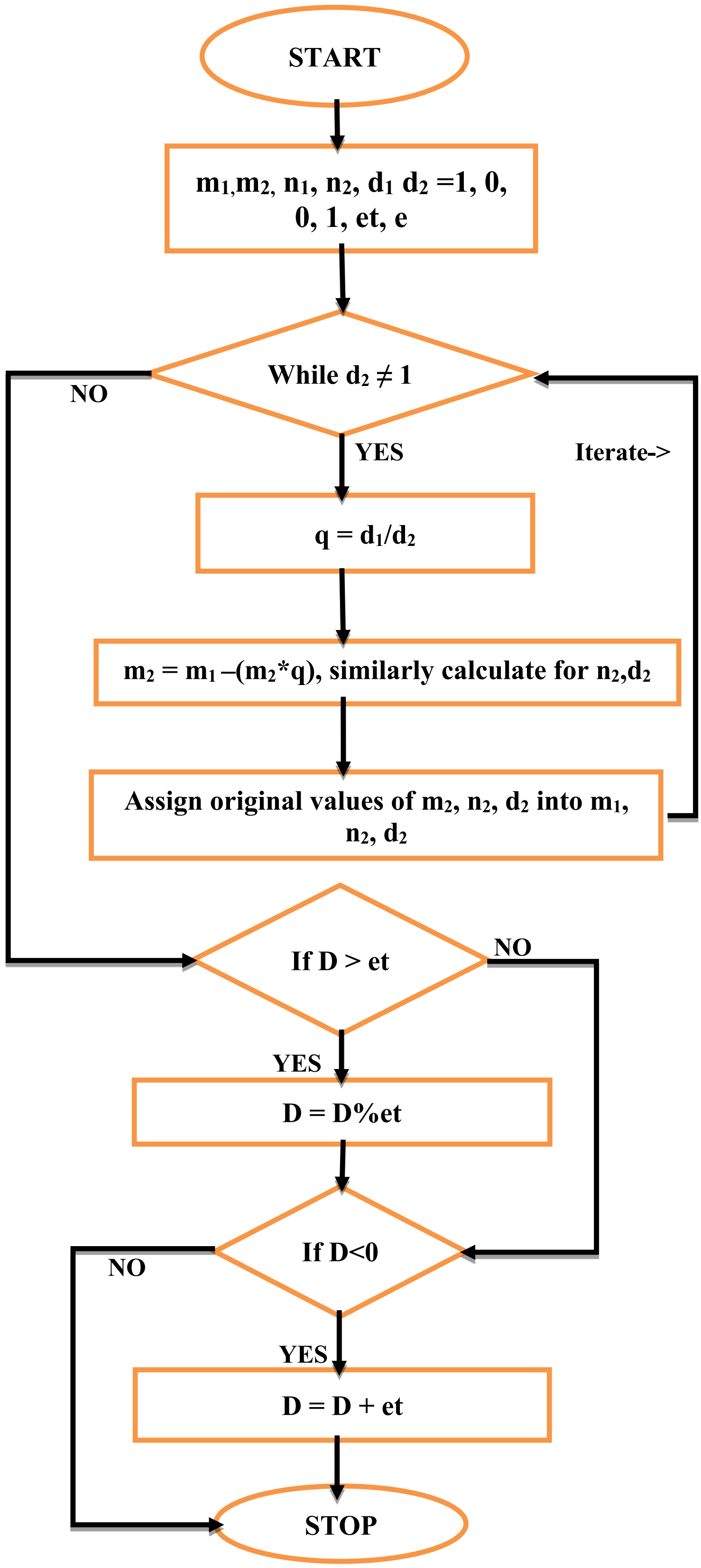
- The values of r_i decreases as we move ahead i.e $r_2> r_3>r_4>\ldots\ldots\ldots\ldots\ldots>r_{n-2}>r_{n-1} =0$
- Hence we obtain zero at some point.

C) ALGORITHM

STEP 1: First we assign the variables the value m_1, m_2, n_1, n_2, d_1 and d_2 as 1,0,0,1, euler toitient and public key.
STEP 2: We iterate till the variable $d_2 \neq 1$
STEP 3: We divide d_1/d_2 and assign it to a new variable q.
STEP 4: We calculate $m_2 =m_1-(n_2*q)$ and similarly for n_2 and $d_2$and also store the original values of m_2, n_2, d_2 in m_1, n_1, d_1.

STEP 5: We assign $D = n_2$ and iterate again until d_2=1.
STEP 6: If D larger then euler toitient
Then: we take modulus of the D with the euler toitient and store it to D, otherwise we add euler toitient and assign it to D
STEP 7: The D is the private key generated and returned.

D) FLOW DIAGRAM



2.3 KARATSUBA FASTMULTIPLICATION

A) OVERVIEW

The Karatsuba algorithm is a fast multiplication algorithm. The Algorithm was discovered by Anatoly Karatsuba in the year 1960 and was published as a journal in 1962.Multiplication is a very important task in any field whether be encryption or decryption process or image processing etc. So, enhancing and making it efficient is a very important task. The time complexity of Karatsuba multiplication of two n-digit numbers is $O(n^{\log 2^{\wedge}3}) = O(n^{1.585})$. The time complexity of general method for Multiplication is $O(n^2)$ which suggest though the time complexity of Karatsuba is less than the time complexity of the general multiplication method but we also have to check for very large numbers and calculate its efficiency which we will be carrying out in the paper.

B) MATHEMATICAL BACKGROUND

Let's use this method to multiply the base-10 numbers 1234 and 8765

$x = (x1*B^m + x2)$ (1) $y = (y1*B^m + y2)$ (2)
  = 12 * 10^2 + 34 = 67 * 10^2+89

$x*y = (x1*B^m + x2) * (y1*B^m + y2)$(3)
  = (12 * 10^2 + 34) * (67 * 10^2 + 89)

$x*y = (x1*y1)*B^{2m} +(x1*y2)*B^m +(x2*y1)*B^{2m}+(x2*y2)$ (4)
=12*67*10^4 + (12*89 + 34*67)* 10^2 + (89*34) = 83,77,626

C) ALGORITHM

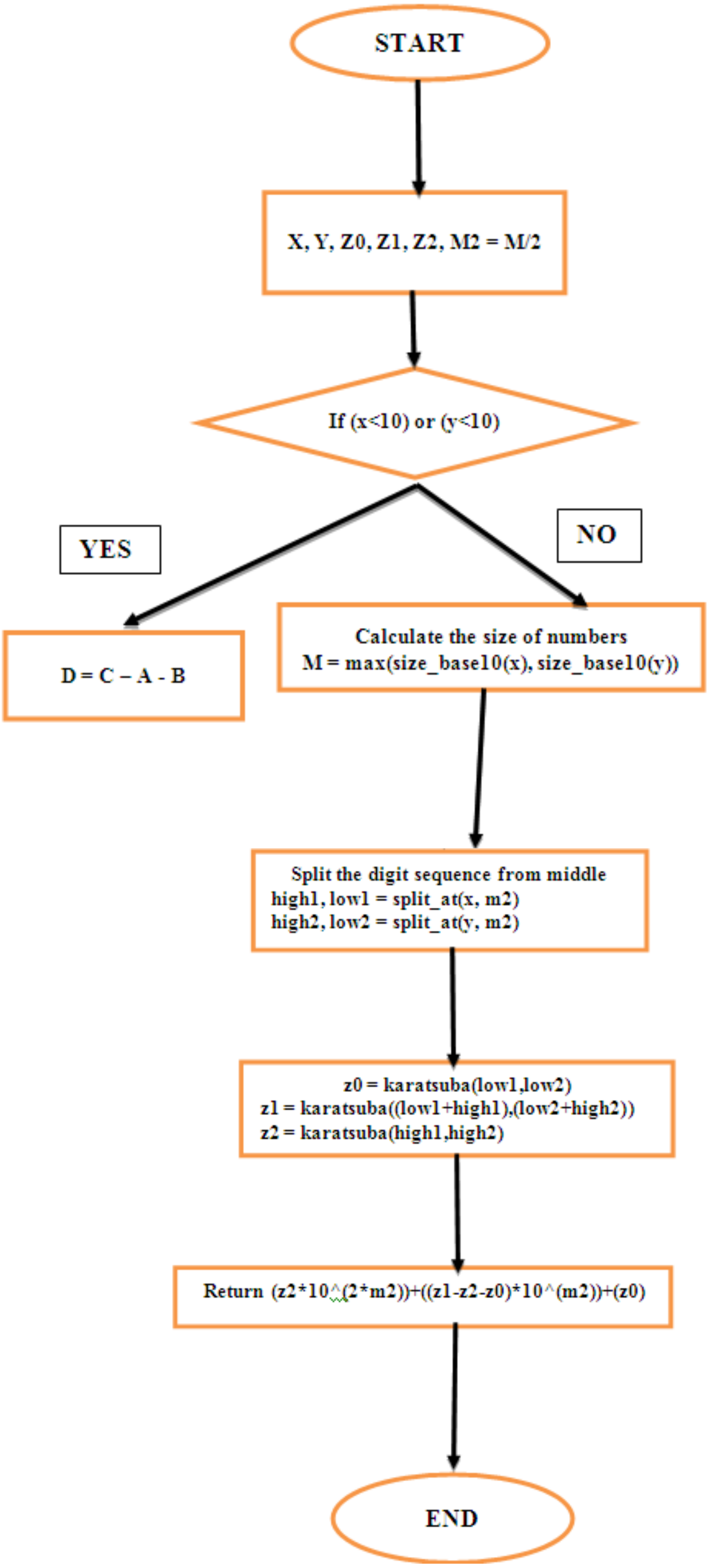
STEP 1: First both the numbers x and y can be represented as x1, x2 and y1, y2 with
- $x = (x1*B^m +x2)$
- $y = (y1*B^m + y2)$

STEP 2: Now their product will be
- $x*y = (x1*B^m + x2) * (y1*B^m + y2)$
- $x*y = (x1*y1)*B^{2m} +(x1*y2)*B^m +(x2*y1)*B^{2m}+(x2*y2)$

STEP 3: Observe the equation obtained in **Step 2**which gives us 4 sub divisions of the main problem i.e x1*y1, x1*y2, x2*y1, x2*y2.

STEP 4: Let a = x1*y1, b = x1*y2 + x2*y1, c = x2*y2,
This makes our product
- $x*y = a*B^{2m} + b*B^m + c$ Now,
- $b = ((x1+x2)*(y1+y2)) – a – c$
- The above algorithm can be applied recursively to a number until the numbers being multiplied are only a single-digit long (base-case)

D) FLOW DIAGRAM



III.RESULTS AND INFERENCES

A) MODULAR EXPONENTIATION

As we know that Modular Exponentiation is used to calculate ($X^Y\%N$) where X is base, Y is exponent and N is the Divisor. Modular Exponentiation Time Comparison was carried out with the naïve method for four different cases in order to understand the dependency and the term which affects the modular exponent operation. The cases of variation like varying power and rest terms constant and similarly for base variation and the divisor variation was carried out for both Modular Exponentiation Approach and the Naïve Approach and the time variation graphs and efficiency variation graph along with the table to compare the execution time for both the methods are also included.

CASE 1: EXPONENT VARIED WITH BASE AND DIVISOR CONSTANT

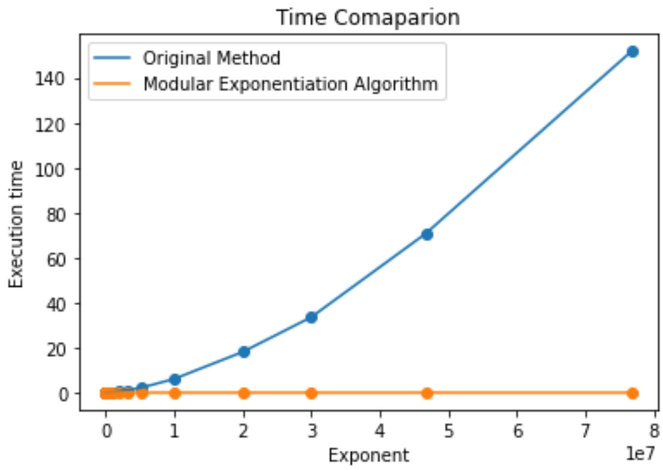


Figure 1 : *Variation of Execution Time with Exponent for Both Modular and General Method.*

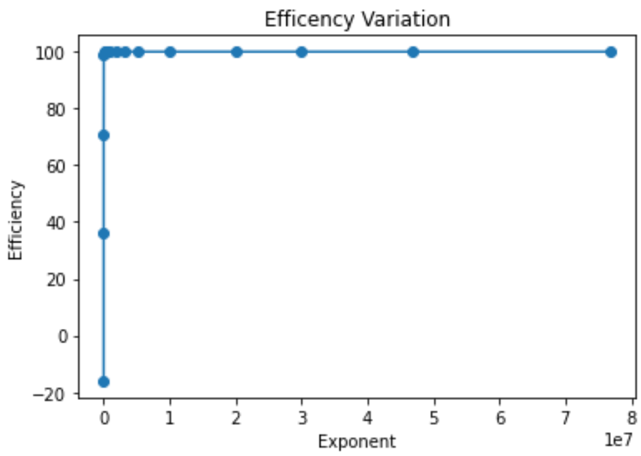


Figure 2 *Variation of Efficiency of Modular Exponentiation with Exponent*

Varying Power	Modular Exponentiation	Naive Method
18	6.9141387939453125e-06	5.9604644775390625e-06
180	3.337860107421875e-06	5.245208740234375e-06
1800	4.291534423828125e-06	1.4781951904296875e-05
19000	5.0067901611328125e-06	0.0005006790161132812
29082	4.5299530029296875e-06	0.0009069442749023438
100000	5.0067901611328125e-06	0.004402875900268555
467889	6.198883056640625e-06	0.048171281814575195
797238	5.4836273193359375e-06	0.11313772201538086
1800098	6.67572021484375e-06	0.4213571548461914
3238457	5.7220458984375e-06	1.0555145740509033
5087234	5.7220458984375e-06	2.078784465789795
9871234	6.4373016357421875e-06	5.904173135757446
20005675	7.152557373046875e-06	18.049366235733032
30006745	6.9141387939453125e-06	33.509174823760986
46729878	6.9141387939453125e-06	70.71131300926208
76789123	7.152557373046875e-06	151.84040236473083
	Average Time =5.841255187988281e-06 s	Average Time =17.733576953411102 s

Table 1: Execution Time Comparison for Modular and General Approach when Power is Varied

Number Of Digits	Modular Exponentiation	Naive Method
10	1.71661376953125e-05	Unresponsive
20	2.9325485229492188e-05	Unresponsive
38	5.984306335449219e-05	Unresponsive
77	0.00012826919555664062	Unresponsive
154	0.00029754638671875	Unresponsive
309	0.0008182525634765625	Unresponsive
617	0.0020084381103515625	Unresponsive
1233	0.006145477294921875	Unresponsive
3011	0.03649616241455078	Unresponsive
9029	0.26302552223205566	Unresponsive
15052	0.7225244045257568	Unresponsive
21073	1.4077339172363281	Unresponsive
27092	2.317553758621216	Unresponsive
33113	3.4513251781463623	Unresponsive

Table 2 : Execution Time Comparison for very Large Number of Digits of Power

INFERENCES

- We observe from Figure 1 that Execution time almost remains the same when exponent is very small but as the number of digits of the exponent increases keeping the base and the divisor constant there is a steep rise in the execution time for the Naïve Method but for Modular Exponentiation the time increases insignificantly.

- From the Figure 2 the efficiency for very small Exponent i.e number of digits is 2 or 3 is negative but as the number of digits of the exponent increases the efficiencies increases sharply and for 8+ digits almost reach 90%+.
- Table 1 contains all the execution time for the Modular Exponentiation and the Naïve Method along with the Average time for number of digits less than 9 and we obtain an Average time of **$5.8412*10^{-6}$s**for Modular Approach and a Average time of **17.3345 s** for the Naïve Approach which suggest a large difference in execution time.

- Table 2 contains the time variation for both the methods when the number of digits for exponents are very large as it can be observed that we have calculated the Execution time for 33000+ digits or a 110000 bit number and the results are quite interesting i.e the naïve method did not get executed for any of the cases but for the same cases we achieved a very less execution time with an Average time of 0.5862967 seconds, that is Enormously less than the general methods we apply.

➢ From above inferences we can strongly conclude that the Modular Exponentiation Execution time is very less even for 110000 bit or 33113-digit exponent and can be utilized for faster calculations.

CASE 2: BASE VARIED WITH EXPONENT AND DIVISOR CONSTANT

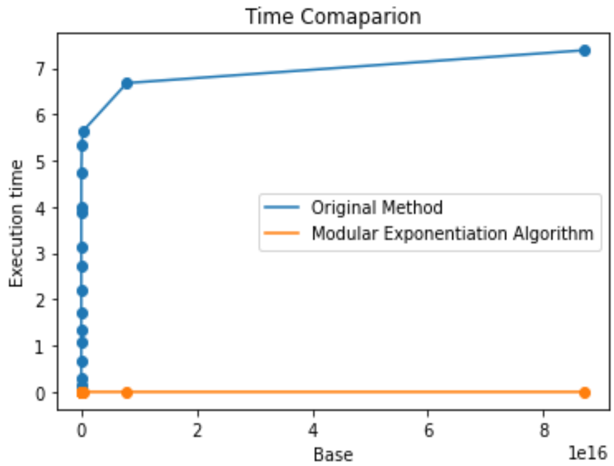


Figure 3 Execution Time Comparison when Base is Varying

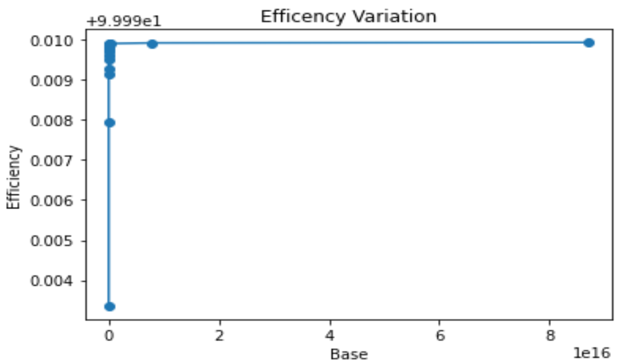


Figure 4 Efficiency Variation of Modular Exponentiation with Base

Varying Base	Modular Exponentiation	Naive Method
21	9.298324584960938e-06	0.1400282382965088
190	6.198883056640625e-06	0.3023514747619629
7800	5.7220458984375e-06	0.6635525226593018
90883	5.245208740234375e-06	1.0767936706542969
891989	5.4836273193359375e-06	1.354353666305542
7398732	5.245208740234375e-06	1.6970398426055908
89023782	5.0067901611328125e-06	2.2144553661346436
972108789	5.7220458984375e-06	2.720888376235962
5670936289	2.2649765014648438e-05	3.1466798782348633
89076528763	5.245208740234375e-06	3.8954391479492188
678920873682	5.4836273193359375e-06	3.996790647506714
9997320967327	5.4836273193359375e-06	4.727440118789673
97625609568921	5.0067901611328125e-06	5.348644256591797
273919737208331	5.4836273193359375e-06	5.636598825454712
7892763907562873	5.4836273193359375e-06	6.677135467529297
87094512874503653	5.245208740234375e-06	7.385764122009277
	Average Time =6.750226020812988e-06 s	Average Time =3.18649722635746 s

Table 2: *Execution Time for both the Methods when the Base is Varied.*

Number Of Digits for Base	Modular Exponentiation	Naive Method
10	7.867813110351562e-06	0.0011858940124511719
19	4.291534423828125e-06	0.0034880638122558594
39	3.814697265625e-06	0.014373779296875
77	4.0531158447265625e-06	0.03257632255554199
154	4.0531158447265625e-06	0.09065985679626465
308	4.76837158203125e-06	0.26733946800231934
617	6.4373016357421875e-06	0.8111312389373779
1233	7.867813110351562e-06	2.4216408729553223
3011	1.3828277587890625e-05	10.168423414230347
9031	5.054473876953125e-05	55.962655782699585
15052	5.316734313964844e-05	122.90123677253723
21073	7.486343383789062e-05	217.56899118423462
27093	9.608268737792969e-05	309.78817224502563
33114	0.00013637542724609375	434.0342173576355
	Average Time =3.3429690769740515e-05 s	Average Time =82.43329230376652 s

Table 4: *Time Comparison for Very Lage Dataset up to 110000-bit Base keeping Exponent and divisor constant*

INFERENCES

- From figure 3 it can be observed that the execution time is varying very steeply for general method but for Modular Exponentiation method the time almost remained the same i.e we obtain a line with almost zero slope.
- The figure 4 suggests that the efficiency is always positive and significantly largei.ethe Modular Exponentiation algorithm works very efficiently no matter how large or small the base is when the exponent and divisor remain constant.
- We also infer from the table 3 the execution time variation for small numbers and table 4 the time variation for very large number that the execution time is always in the order of 10^{-6} and 10^{-5} as well as the average execution time for the Modular Exponentiation method but it is amazing and quite surprising to observe from both the tables that the average execution time changes from 3.186 s to 82.433 when the set of number of digits are significantly increased keeping the exponent and divisor constant.

➢ From all of the above Inferences we conclude that the Modular Exponentiation algorithm is far better than General or the Naïve approach and no matter how big the base be it works with a very less execution time.

CASE 3: DIVISOR VARIED WITH BASE AND EXPONENT CONSTANT

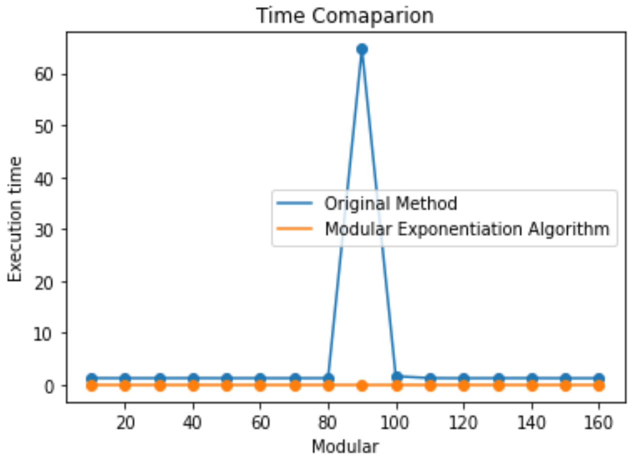


Figure 5: Execution time Variation when the Divisor is Varied keeping the base and exponent constant

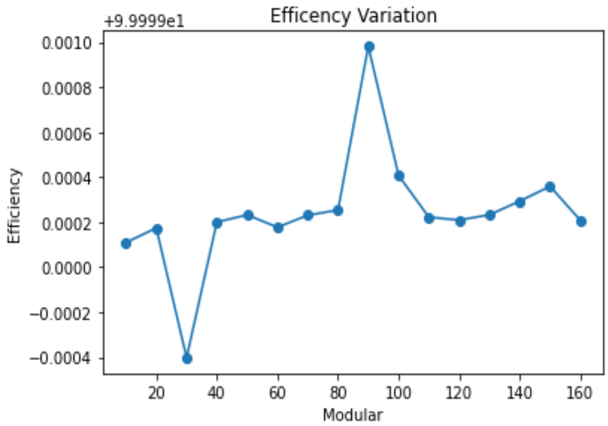


Figure 6:Efficiency Variation when Divisor is varied keeping the base and the exponent constant.

Varying Modulo	Modular Exponentiation	Naive Method
10	1.1444091796875e-05	1.284681797027588
20	1.049041748046875e-05	1.2695870399475098
30	1.7881393432617188e-05	1.2749390602111816
40	1.0251998901367188e-05	1.2812938690185547
50	9.775161743164062e-06	1.2727687358856201
60	1.049041748046875e-05	1.2747762203216553
70	9.775161743164062e-06	1.2699296474456787
80	9.5367431640625e-06	1.277545690536499
90	1.239776611328125e-05	64.67881679534912
100	9.775161743164062e-06	1.648488998413086
110	1.0013580322265625e-05	1.286931037902832
120	1.0013580322265625e-05	1.2659721374511719
130	9.775161743164062e-06	1.273364543914795
140	9.059906005859375e-06	1.2815558910369873
150	8.106231689453125e-06	1.2640345096588135
160	1.0013580322265625e-05	1.2645833492279053
	Average Time =1.055002212524414e-05 s	Average Time =5.260579332709312 s

Table 5: *Execution Time and the Average Time for both the Approach when Divisor is Varied*

INFERENCES

➢ Figure 5 suggest that the execution time variation is similar for both the approach when the divisor is varied although the original method variation is a bit higher than the Modular Exponentiation approach that suggests that the modular exponentiation method is better than the general approach.

➢ The efficiency variation from figure 6 is quite uneven which suggest that both the approach work with similar execution time when the divisor is varied and the base, exponent is kept constant.

➢ Also, from the table we can observe that on an average the modular exponentiation is better than the general method, as well as it can be inferred from the data that the average time for general method when divisor is kept constant is better than the average time for the general method when the exponent or the base is varied.

➢ Hence, we conclude that the modular exponentiation algorithm is quite better than the general method as the data and the variations suggest. We also observe that though the divisor increases but time almost remains constant in both the methods i.e higher the divisor easier it to find the remainder.

CASE 4: ALL THE FACTORS ARE VARIED

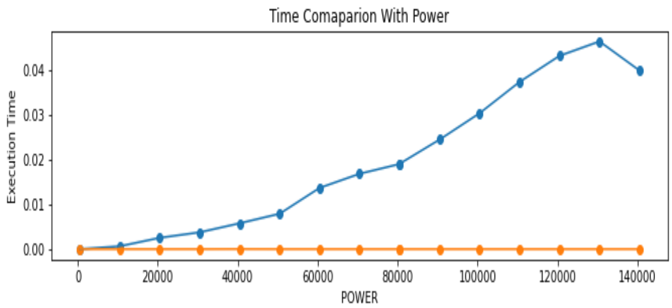


Figure 7: Execution Time Variation for both the methods when all factors are Varied.

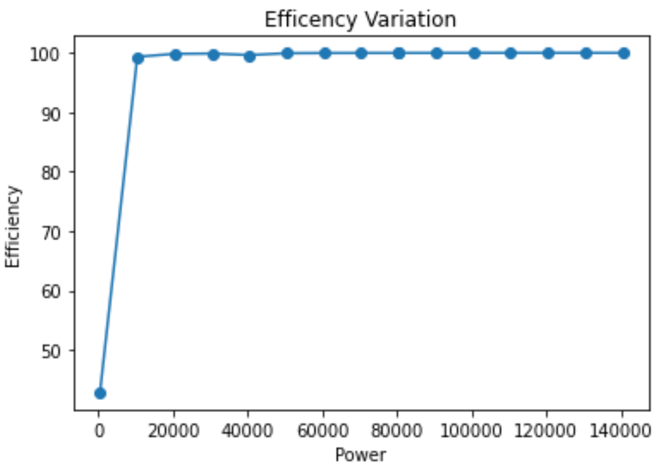


Figure 8: Efficiency Variation when all the factors are varied.

Varying Values	Modular Exponentiation	Naive Method
(23, 500, 12)	6.67572021484375e-06	1.1682510375976562e-05
(46, 10500, 24)	4.291534423828125e-06	0.000640869140625
(69, 20500, 36)	4.291534423828125e-06	0.0025217533111572266
(92, 30500, 48)	4.5299530029296875e-06	0.003744363784790039
(115, 40500, 60)	2.193450927734375e-05	0.005756378173828125
(138, 50500, 72)	5.245208740234375e-06	0.00788116455078125
(161, 60500, 84)	5.4836273193359375e-06	0.013636589050292969
(184, 70500, 96)	4.76837158203125e-06	0.01677846908569336
(207, 80500, 108)	5.4836273193359375e-06	0.018945693969726562
(230, 80500, 120)	5.245208740234375e-06	0.01900339126586914
(253, 90500, 132)	4.5299530029296875e-06	0.024384498596191406
(276, 100500, 144)	5.245208740234375e-06	0.030251026153564453
(299, 110500, 156)	5.0067901611328125e-06	0.03722262382507324
(322, 120500, 168)	5.245208740234375e-06	0.04303479194641113
(345, 130500, 180)	5.0067901611328125e-06	0.04623842239379883
(368, 140500, 192)	5.245208740234375e-06	0.03976893424987793
	Average Time =6.139278411865234e-06 s	Average Time =0.01936379075050354 s

Figure 6: Table for Execution Time and Average Execution Time for both Methods

INFERENCES

- This case is to observe the variation of the execution time when all the factors are varied together to observe how better the algorithm works as compared to the original method and draw conclusions on the basis of the data obtained from the graphs and table.
- From figure 7 it is observed that the execution time steeply increases for general method but the plot for the Modular Exponentiation suggest that the slope is almost zero from which we can infer that the Modular Exponentiation method is better.
- The efficiency variation from figure 8 suggest that the efficiency of Modular Exponentiation is far better over the general method. The value almost reaches 99% for larger set of number.
- From table 6 the average execution time for the modular exponentiation is $6.139*10^{-6}$s and the average execution time for the general method is 0.0193 s which is quite higher than the Modular Exponentiation approach and also for any set of numbers the execution time for Modular Exponentiation always remains in the range of 10^{-6} and 10^{-5}.
- From all the above inferences we conclude that the Modular Exponentiation method is far more better than the general method even when all the factors are varied and also efficiency obtained in this case is the best then all the cases which suggest Modular Exponentiation algorithm is the best.

FINAL CONCLUSION

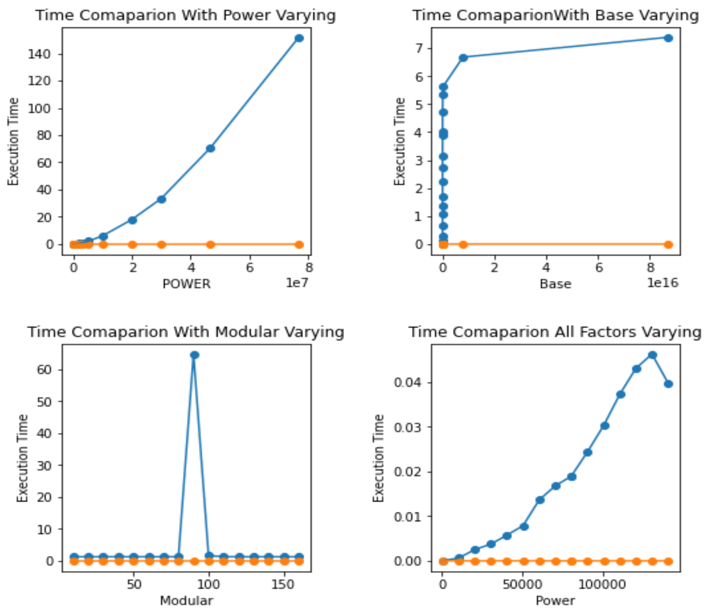


Figure 9: Variation of all the cases

Efficiency_Power	Efficiency_Base	Efficiency_Divisor	Effciency_All
-16.0	99.99335967894899	99.99910918860816	42.857142857142854
36.36363636363637	99.99794977581587	99.999173714196	99.33035714285714
70.96774193548387	99.999137664962	99.99859747073484	99.82981941949514
99.0	99.99951288636967	99.99919987138398	99.87901942056669
99.50052576235542	99.99959511112527	99.99923197659814	99.61895294897283
99.8862836410895	99.99969092011817	99.99917707772444	99.93344627299129
99.98713157959861	99.99977390421873	99.99923025958462	99.95978739772012
99.9951531397118	99.99978969935157	99.99925351059969	99.97158041322079
99.99841566230974	99.99928020116786	99.99998083179821	99.97105607570724
99.99945789039401	99.99986535000185	99.99940702293114	99.97239856472537
99.99972474078037	99.99986279923561	99.99922190233764	99.98142281669209
99.99989097031053	99.9998840042987	99.99920902048109	99.98266105515361
99.99996037225198	99.99990639141583	99.99923233595675	99.98654906708173
99.99997936643075	99.99990271389734	99.99929305416414	99.98781170187424
99.99999022201894	99.99991787455346	99.99935870171048	99.98917179717229
99.99999528942412	99.99992898217904	99.99920815181314	99.98681078877479
Avg Efficiency =86.85611793348724%	Avg Efficiency =99.9920987235375%	Avg Efficiency =99.9992427556639%	Avg Efficiency =96.32737423375927%

Table 7: Efficiencies and Average Efficiencies for all the cases.

➢ From figure 9 we conclude that the General method for exponential modular calculation depends heavily on the exponent i.e as the number of digits of the exponent increases the execution time also increases drastically whereas the same is not in the case of the base and the divisor.

➢ Table 7 contains the efficiencies for all the cases and as it can be observed from the table that avg efficiency for base and division variation is 99.99% in both the cases which suggest that the no matter how large number you may use for base or the divisor, the execution time will remain small. But not the same case with the general method.

➢ Even when the power is varied the maximum time taken for the calculation for the modular exponential is 3 seconds when the exponent is 110000-bit number or 33000+ digit numbers i.e the modular exponentiation algorithm works with a very small execution time no matter which factor or how much it is varied.

➢ From all the above results and efficiencies, we finally conclude that no matter how big the base or exponent be the modular exponentiation algorithm works far better with a very less execution time. So, instead of using the general method one must use the Modular Exponentiation Algorithm.

B) EXTENDED EUCLIDEAN ALGORITHM

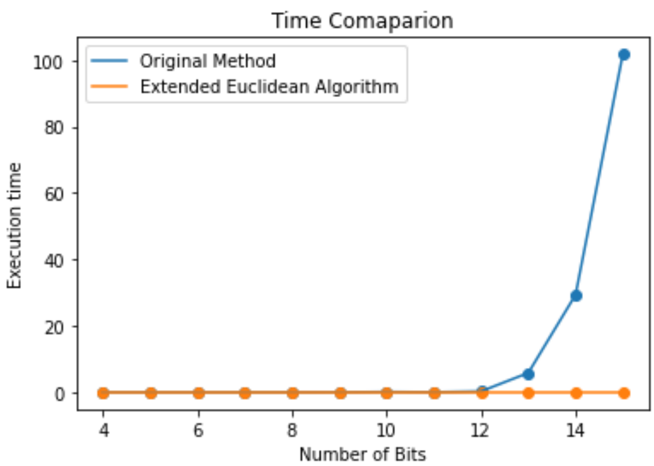


Figure 10: Execution Time Variation with number of Bit for Extended Euclidean algorithm and the General Method

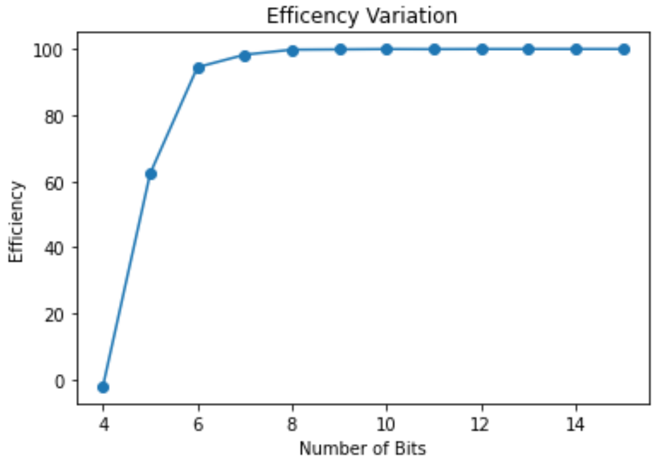


Figure 10: Efficiency Variation with Number of Bits

Number of Bits	Extended Euclidean	Naive Method	Efficiency
4	1.1920928955078125e-05	1.1682510375976562e-05	-2.0408163265306123
5	9.059906005859375e-06	2.4080276489257812e-05	62.37623762376238
6	7.867813110351562e-06	0.0001430511474609375	94.5
7	1.3828277587890625e-05	0.0008058547973632812	98.28402366863905
8	1.0728836059570312e-05	0.005665302276611328	99.81062200151503
9	1.9550323486328125e-05	0.017882823944091797	99.89067541263366
10	2.2172927856445312e-05	0.12861895561218262	99.9827607620114
11	1.8835067749023438e-05	0.04561209678649902	99.9587059813602
12	1.71661376953125e-05	0.27605605125427246	99.99378164774244
13	2.2411346435546875e-05	5.828280687332153	99.99961547242424
14	2.09808349609375e-05	29.34084415435791	99.99992849273576
15	2.384185791015625e-05	101.77326560020447	99.99997657355519
	Average Time =1.6530354817708332e-05 s	Average Time =11.451434195041656 s	Average Efficiency =87.7296259424874%

Table 8: Execution Time and Average Time along with Efficiencies and Average Efficiencies

INFERENCES

➢ The method to calculate the private key is too long as one has to check for each and every value until we obtain the value which satisfies the condition satisfies as explained in the overview. But as the number of bits increases the execution time increases sharply. So, we calculated the variation and efficiencies for the Extended Euclidean algorithm a different algorithm to calculate the private key.

➢ It is observed from the execution time variation graph that as the number of bits increases for the randomly generated prime number the time for the general method increases sharply after certain bit whereas the execution time remains almost constant and very less for the Extended Euclidean Algorithm.

➢ The efficiency variation increases sharply as the number of bits for randomly generated prime number increases i.e for very small bit number the efficiency is negative whereas for very large bit number the efficiency increases up to 99.999%

➢ The average execution time and the average efficiency for the extended Euclidean algorithm is 1.6530×10^{-5} s and 87.23% which is very much better than the execution time for the general method proposed.

FINAL CONCLUSION

➢ From all of the above inferences we conclude that the extended Euclidean algorithm is a very efficient approach as compared to the general method and must be used to calculate the private key for RSA encryption because the private key generated is 4096 bit or 2048 bit for a stronger and secure encryption.

➢ In order to check the correctness of the algorithm we implemented the results of both Extended Euclidean and Modular Exponentiation Algorithm and encrypted and decrypted an Image and it was successful.

RESULTS FOR IMAGE ENCRYPTION AND DECRYPTION

C) KARATSUBA ALGORITHM

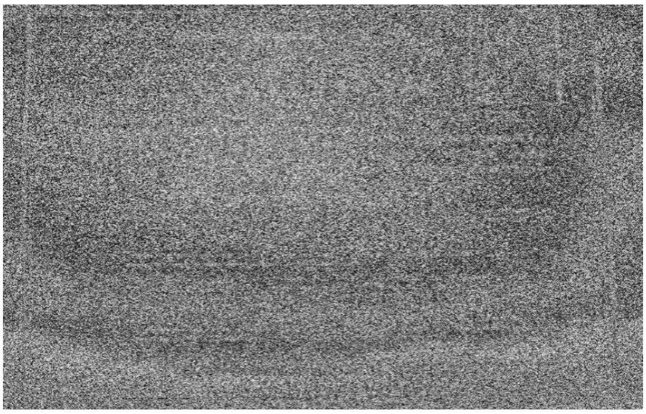
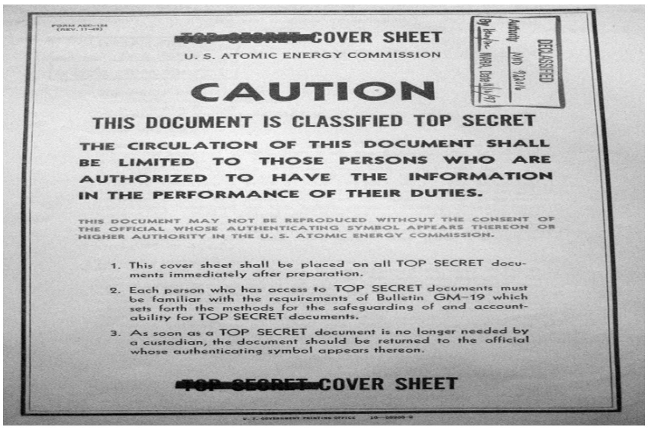


Figure 11: Result for Encrypted Image



[redacted] COVER SHEET

U. S. ATOMIC ENERGY COMMISSION

CAUTION

THIS DOCUMENT IS CLASSIFIED TOP SECRET

THE CIRCULATION OF THIS DOCUMENT SHALL BE LIMITED TO THOSE PERSONS WHO ARE AUTHORIZED TO HAVE THE INFORMATION IN THE PERFORMANCE OF THEIR DUTIES.

THIS DOCUMENT MAY NOT BE REPRODUCED WITHOUT THE CONSENT OF THE OFFICIAL WHOSE AUTHENTICATING SYMBOL APPEARS THEREON OR HIGHER AUTHORITY IN THE U. S. ATOMIC ENERGY COMMISSION.

1. This cover sheet shall be placed on all TOP SECRET documents immediately after preparation.
2. Each person who has access to TOP SECRET documents must be familiar with the requirements of Bulletin GM-19 which sets forth the methods for the safeguarding of and accountability for TOP SECRET documents.
3. As soon as a TOP SECRET document is no longer needed by a custodian, the document should be returned to the official whose authenticating symbol appears thereon.

[redacted] COVER SHEET

Figure 12: Result for Decrypted Image

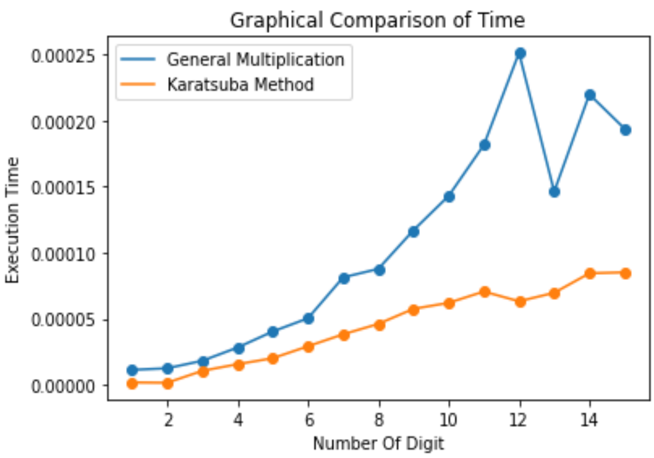


Figure 13: Execution Time Variation for Karatsuba and General Multiplication Method

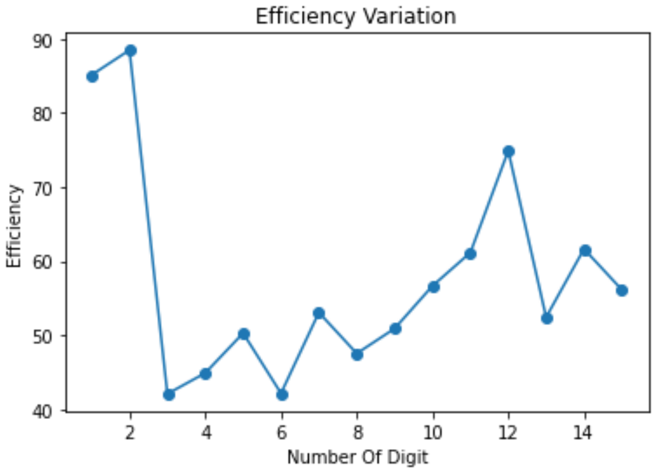


Figure 14: Efficiency Variation for Karatsuba Algorithm.

Number of Digits	Karatsuba Multiplication	Naive Multiplication	Efficiency Of Karatsuba
1	1.6689300537109375e-06	1.1205673217773438e-05	85.1063829787234
2	1.430511474609375e-06	1.239776611328125e-05	88.46153846153845
3	1.049041748046875e-05	1.811981201171875e-05	42.10526315789473
4	1.5497207641601562e-05	2.8133392333984375e-05	44.91525423728814
5	2.002716064453125e-05	4.029273986816406e-05	50.29585798816568
6	2.9087066650390625e-05	5.030632019042969e-05	42.18009478672986
7	3.814697265625e-05	8.130073547363281e-05	53.0791788856305
8	4.601478576660156e-05	8.7738037109375e-05	47.55434782608695
9	5.745887756347656e-05	0.00011706352233886719	50.91649694501018
10	6.198883056640625e-05	0.0001430511474609375	56.666666666666664
11	7.05718994140625e-05	0.00018167495727539062	61.15485564304461
12	6.318092346191406e-05	0.00025153160095214844	74.88151658767772
13	6.961822509765625e-05	0.00014638900756835938	52.44299674267101
14	8.440017700195312e-05	0.0002200603485107422	61.64680390032503
15	8.511543273925781e-05	0.00019407272338867188	56.14250614250614
	Average Time =4.364649454752604e-05 s	Average Time =0.00010555585225423177 s	Average Efficiency =57.83665072999727%

Table 9: Execution time, Average Execution time and Average Efficiency for Karatsuba Algorithm

INFERENCES

➢ The variation for Karatsuba algorithm increases slower as compared to the variation for the General Multiplication Algorithm as the number of digit increases.

➢ As the number of digits increases the execution time difference increases which suggest that as the number of digits for the numbers to be multiplied increases the Karatsuba will work faster than the General Multiplication Method.

➢ The efficiency variation is a bit uneven for smaller number but as the number of digits increases the efficiency gets a bit stable and reaches a range of 45% to 55%.

➢ The average time for Karatsuba algorithm is 4.36×10^{-5} and for the naïve method is 1 millisecond and an average efficiency of 57.8366% that ranges between 45% to 58% which suggest that the Karatsuba algorithm is better than the general method.

FINAL CONCLUSION

➢ From all of the above inferences we conclude that the Karatsuba Algorithm is approximately 50% efficient than the General Multiplication method. We also conclude that for smaller digit both approach works the similar way but as the number of digits increases the run time always remains constant or in a certain range, Hence it can be used for different purposes.

VERDICT

The analysis of the algorithms in this paper conclusively proves the magnitude of difference in execution time and efficiency over their naïve counterparts. The implications of using these algorithms in consumer-based products are not limited to any one domain like cryptography, instead these methods could save a lot of time in various domains involving large scale computations.

REFERENCES

[1]. http://www.mathaware.org/mam/06/Kaliski.pdf
[2]. https://people.csail.mit.edu/rivest/Rsapaper.pdf
[3]. http://airccse.org/journal/nsa/6414nsa02.pdf
[4]. https://scialert.net/fulltext/?doi=itj.2013.1818.1824
[5]. https://sites.math.washington.edu/~morrow/336_09/papers/Yevgeny.pdf
[6]. https://core.ac.uk/download/pdf/82467764.pdf
[7]. https://patentimages.storage.googleapis.com/97/ea/37/7 0c3796ee1cf08/US20070083585A1.pdf
[8]. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.8270&rep=rep1&type=pdf
[9]. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.300.7959&rep=rep1&type=pdf
[10]. D.E. Knuth, The Art of Computer Programming: Semi numerical Algorithms, Volume 2, Second edition, Addison-Wesley, Reading, MA, (1981)
[11]. An RGB image Encryption using RSA Algorithm, International Journal for current Trends in Engineering and Research Volume 3 issued 3 March 2017.
[12]. T. E1Gamal, A public key cryptosystem and a signature scheme based on discrete logarithms, IEEE Transactions on Information Theory 31 (4), 469-472 (July1985)
[13]. On the complexity of extended Euclidean algorithm, by George Havas, published by Elsevier ScienceDirect.
[14]. Implementation of Karatsuba algorithm using polynomial multiplication by Sudhanshu Mishra and Manoranjan Pradhan published under the journal Indian Journal of Computer Science and Engineering (IJCSE).
[15]. Comparative study of Efficient Modular Exponentiation Algorithm by Ibrahim Marouf, Mohammed Mosab Asad, Qasem Abu Al-Haija, Published under (COMPUSOFT), an international journal of advanced computer technology, 6 (8), august-2017 (volume-vi, issue-viii)
[16]. Image Encryption/Decryption Using RSA Algorithm by Sunita published under International Journal of Computer Science and Mobile Application, Vol.5 Issue. 5, May- 2017, pg. 1-14
[17]. Implementation Color-Images Cryptography Using RSA Algorithm by Ali Saleh AL Najjar published under International Journals of Advanced Research in Computer Science and Software Engineering ISSN: 2277-128X (Volume-7, Issue-11).