Int.J.HIT.TRANSC:ECCN.Vol.3: Issue 2A(1) (2017) Page 1-8 ISSN: 0973-6875

Available Online at www.hithaldia.in/locate/ECCN

ORIGINAL CONTRIBUTION

Fast Modular Exponentiation using Number Theoretic Library

Arijit Choudhury[1,*], Utpal Roy[2] and Debasis Giri[3]

[1,2]*Visva Bharati University, Bolpur, Santiniketan West Bengal, India*

[3]*Haldia Institute of Technology, Haldia, India*

(Received Date: 20th August, 2017; Acceptance Date: 30th September, 2017)

ABSTRACT

Different cryptographic algorithms use to calculate $a^p \bmod n$ for their encryption, decryption or any other operation. There exist many algorithms for finding $a^p \bmod n$. Objective of this paper is to find an efficient calculation for modular exponentiation. The proposed algorithm is inspired from the method of addition chain. In case of addition chain (using window method), the window size is fixed depending upon the clustering of ones, because the ones are needed for multiplications. So our objective is based on to reduce the number of arithmetic operations (multiplications or squiring). It can be possible by finding the maximal consecutive matched bits into the exponent. If we can able to find maximal consecutive matched bits, then the decimal value for the matched bits will be calculated and then we can get the total value for $a^p \bmod n$. Here the main attention is given to reduce the multiplication. Because when we able to calculate the all decimal values of the matched bits, then each decimal values are considered as the exponent and we calculate the value considering decimal value is in the power of a and upon it we do the modulus operation. Now the next decimal value as power of a is evaluated using help of previous one. Preceding this manner the total exponent (p) as power of a over modulo n is evaluated. However, our experiment shows the satisfactory result.

Key words: Cryptography, Repeated Squire, m-Array, Sliding Window

1. INTRODUCTION

Our current research focuses on fast evaluation of many cryptographic algorithms like RSA. Many cryptographic algorithm consist computing $a^p \bmod n$ where a, p, n all are very large numbers. Reduce the number of multiplications increase the time efficiency, which is the most required policy of all cryptographic algorithms which are used to calculate $a^p \bmod n$ repeatedly. In general procedure to calculate a^p, p times multiplications of a is required. Some authors proposed some methods in [3, 6, 4, 7, 5].

2. EXISTING METHODS

In this section, we briefly describe the existing methods to evaluate $a^p \bmod n$.

2.1 Repeated Square and Multiplication method

Repeated square methods concentrated on the increasing the number of squaring on behalf of multiplications. It can be achieved by a very simple method. To find a^p, if p = 0 then returns 1, else if p = 1 then returns a, else if p = odd number then calculate $a \times (a^{(p-1)/2})^2$, else if p = even number then returns $(a^{p/2})^2$.

Algorithm 1: Calculate a p mod n using Repeated Square Method

RepeatedSquareMethod(a, p, n)
1: initialize ZZ type variables c, d;.
2: initialize long type variables k,i;
3: k ← NumBits(p);
4: c ← 0;
5: d ← 1;
6: for i = k to 0 do
7: c ← c × 2;
8: d ← d 2 mod n;

*Corresponding Author:arijitc.007@gmail.com

```
 9:        if  bit(b,i) = true then
10:             c ← c + 1;
11:             d ← (d × a) mod n;
12:       end if
13: end for;
14: return d;
```

The value of c is the binary values of bit combinations of p. We can write $d = a^c \bmod n = a^p \bmod n$.

2.2 Addition chain

Here the idea is to create the a^p using the previous calculations. For example, if we have to calculate the value of a^{15} then in general procedure we have to multiply a with itself ^{14} times, but using addition chain based on following properties:

1. The first number is always one.

2. Every number can be achieved using sum of some earlier numbers.

3. The numbers are occurred in a sequence.

4. We use such a list of exponents to achieve exponentiation.

5. The length of addition chain is equal to the number of elements in the series; we can able to do so using 5 multiplications as shown below:

1. $a^2 = a^1 \times a^1$

2. $a^3 = a^2 \times a^1$

3. $a^6 = a^3 \times a^3$

4. $a^{12} = a^6 \times a^6$

5. $a^{15} = a^{12} \times a^3$

Here all the powers of a are not stored in memory. Only product and the last power of a are required. The algorithm does only O(log p) number of multiplications. In general we can write it as $a^p=(a^{p/2})^2 \times a^p \bmod 2$.

2.3. Window Method

In this subsection, we briefly describe Window method by Knuth[1]. The main idea of window method based on decreasing the number of multiplication. So some portion of the exponent (in binary) is taken as window. The window size is may be fixed or variable. The partition into windows depends upon the clustering of ones. That is in binary representation of windows where the series of 1 exist and the length of the sequence also supports the size of the window. The sequence is taken as a window. The windows are evaluated using multiplications and then by squiring (depending upon the number of 0s following it). It is added into its right place and then multiplied with the next window. By that manner the total exponentiation is evaluated. For example 800667069276199255 is taken as the large exponent, then the windows are look like,

2.4. M-Ary Method

The m-ary method is invented by Jong et al.[8] in 1989. The M-Array method scans the exponent from least significant bit to the most significant bit. Then groups the bits of the exponents such that each of length $\log_2 m$ (m is power of 2). The exponent is partitioned into 'p' partition. Each partition is length of $\log_2 m$ digits. If the length is less than $\log_2 m$ then the redundant bits are treated as zero. The M-ary method is depicted in Algorithm 2.

Algorithm 2: Calculate $a^p \bmod n$ using M-Ary Method

M-Ary Method(a,p,n)

```
1: initialize ZZ variables c,d;
2: initialize long variables k, i, temp;
3: k ← NumBits(p);
4: temp ← long (bit (p, k − 1));
5: c ← power (a, temp);
6: c ← c mod n;
7: for i = k − 2 to 0 do
8:          temp ← 2;
9:          c ← power(c, temp) mod n;
10:         if bit (b, i) ==1 then
11:             temp ← long (bit (b, i));
12:             d ← power(a,temp);
13:             c ← (c ∗ d)modn;
14:         end if
15: end for
16: return c;
```

2.5. Sliding Window Method

In this method, the window size is variable. The windows are partitioned on the basis of positioning of 0s. So the number of 0s windows is increased. Hence squaring increased rather than the number of multiplications. For sliding window method for each window the right most bit should be 1. The main problem of this procedure is to specify the window size. The time required for the calculation is totally dependent on the window size. The pseudo code of the method is described below. The method can be described in Algorithm 3.

Algorithm 3: Calculate a^p mod n using Sliding Window Method

```
SlidingWindow(a,p,n,windowsize)
1: ptr ← head1;
2: mn ← 2;
3: expo ← (ptr− > data2);
4: temp ← todouble (expo);
5: c ← power (a, temp);
6: c ← c mod n ;
7: ptr ← ptr− > next ;
8: while ptr! = NULL do
9:          temp ← ptr− > data1;
10:         pwr ← power (mn, temp);
11:         temp ← todouble (pwr);
12:         c ← c × power (a, temp);
13:          if ptr− > data2! = 0 then
14:              expo ← (ptr− > data2);
15:              temp ← todouble (expo);
16:              c ← (power(c, temp)) mod n;
17:          end if
18: ptr ← ptr− > next;
19: end while
20: return c;
```

3. PROPOSED METHOD

In the proposed method, we calculate a^b mod p depending upon the similar bits in b. First, the method is to find the maximal matched bits in the exponent b. After that, the decimal equivalent of matched bits are calculated, say (n_2, n_7, n_5, n_9, n_1, n_{10}, ...), where n_i represents a bit. These decimal numbers are arranged as (n_1, n_2, n_5, n_7, n_9, n_{10}, ...). Now by evaluating these decimal numbers as exponent of a, we can able to find the a^b mod p. In the proposed method, we use the strategy of divide and conquer method to find maximal matched consecutive bits in b. So we partitioned the b in two parts then find consecutive zeros after bitwise XOR operation of two parts. Following is our proposed method.

Algorithm 4: Algorithm for proposed method

1: Two parts are generated by right shifting one by one and the rest of digits in exponent as part two.

2: XOR the smaller one with another from most significant bit to least significant bit more than one times such that all bits are covered.

3: To store the part one and part two along with matched bit positions and for total number of matched bits, we use adjacency list.

4: From the adjacency list, we select the row corresponding to maximum matched bits.

5: From the list, the decimal values are calculated equivalent of the matched bits.

6: Each time the, decimal values are stored into a binary search tree.

7: The inorder operation is applied on the binary search tree. As a result, the decimal values become sorted. We do not use any other memory location to store the sorted decimal numbers. We only rearrange the tree.

8: An indexing procedure is used to reduce the complexity while calculating the a^p mod n. All the decimal values are treated as exponent and evaluated as power of a.

9: The next decimal value as the power of a is evaluated using the previous one and the previously evaluated values. For this, we use the indexing procedure.

Our experiment results to calculate a^p mod n based on our method for the different input sets as follows:

Set one:

a=564789569854756321

p=25565448896541236544775621364598756423654789564123654478965823654789659856214569874
5632145698745632586985471523654789569325874563215647895698563214758965874556552526544
5688798545632145745698547412365478965412547895463215698745632145647895698547563212364
5879654788524123652458796587452146523698547123654789568563214569874563212365478998523
6974123654789651235748967513624565555656632322365415935785265412365478935762154987456
32365124732145697875365214523657896 54712365478965412324566988556623255566

n=78965896532456987456321564789654236598745632156647789569853214563123647896547589653
2145698745632145698745632156478956865932569845632147569853214569856325698745632154698
7755669885566232555669987845875654123658974698456321475698532145698563256987586985471
5236547895693258745632156478956985632147589658745565525265445688798545632145745698547
4123654789654125478954632156987456321456478956985475632123645879654788524123652458796
5874521465647895686593256984563214756985321456985632569874563215469877546321547896547
89

Set two:

a=582365478965985621456987456321456987

p=25265445688798545632145745698547412365478965412547895463215698745632145647895698547
5632123645879654788524123652458796587452146523698547123654789568563214569874563212365
478998523697412 365478965123574896751362456555565663232236541593578526541236547893

n=21564789658965324569874563215647896542365987456321566477895698532145631236478965475
8965321456987456321456987456321564789568659325698456321475698532145698563256987456321
5469877546321547896547896542321456985632569856478598547123645698745632156478965471236
5478965412365

Set three:

a=12345698745632156478965896532456987456321564789654236598745632156647789569853214563
1236478965475896532145698745632145698745632156478956865932569845632147569853214569856
3254123654789357 621549874563236512

p=89652365478956325565448896541236544775621364598769854756321236458796547885241236524
5879569787536521452365789654712365478965412324566988556623255566998784587569877546321
5478965478965423214569856325698564785985471236456987456321564789654712365478965412365
6985474123654789654125478954632156987456321456478956985475632123645879654788524123652
458796587452146523698547123654789568563214569874563212365478998523697412365478965

n=89653214569874563214569874563215647895686593256984563214756985321456987965874521465
2369854712365478956856321456987456321236547899852369741236547896512357489675136245655
5565663232236541593578526541236547893576215498745632365124732145697875365214523657478

96582365478965983214756985321456985632569874563215469877546321547896547896321475698532145698563256987456321546987754632154789654789632147569853214569856325698745632154698775463215478965478965423214569856325698564785985471236456987456321564789654712365478965412365478965823658965478541236547896523654789563255654488965412365447756213645987564 2365478956412365447896582365478965985621345698563256985647859854712364569874563215647896547123654789654123657562136459875642365

p=236547896512357489675136245655556566323223654159357852654123654789357621549874563236512473214569787536521452365789654712365478965412324566988556623255566998784587565412365897458974562113244569877 44569877

n=456325869854715236547895693258745632156478956985632147589658745565525265445688798545632145745698547412365478965412547895463215698745632145647895698547563212364587965478852412365245879658745214652369854712365478956856321456987456321236547899852369741236547896512357489675136245655556566323223654159357852654123654789357621549874563236512473214569787536521452365789654712365478965412324566988556623255566998784587565412365897456211324456987745632154789654789654232145698563256985647859854712364569874563215647896547123654789654123654789658236589654785412365478965236547895632556544889654123654477562136459875642365

ISSN: 0973-6875

Set seven:

a=12345698745632156478965896532456987456321564789654236598745632156647789569853214563
12364789654758965321456987456321456987456321564789568659325698456321475698532145698563
25698745632154698775463215478965478965423214569856325698564785985471236456987456321564
64789654712365478965412365478965823658965478541236547896523654789563255654488965412365
54477562136459875642365

p=365478998523697412365478965123574896751362456555565663232236541593578526541236547893
57621549874563236512473214569787536521452365789654712365478965412324567562136459875642
36547895641236544789658236547896598562145698745632145698745632586985471523654789569325
87456321564789569856321475896587455655252655595878521654789215622145698745632145698745
63258745632156478956985632147589658745565525265559587852165478921562214569874563214569
87456325869854715236547895693258745632156478956985632147589658745565525265445688798545
63214574478965823658965478541236547896523654789563255654488965412365447756213645987564
78956412365447896582365478965985621456987456321456987456325869854715236547895693258745
63215647895698563214758965874556552526544568879854563214574569854741236547896541254789
5463215698745632145 647895698547563212364587965478852 3

Set eight:

a=564789569854756321

p=145698563256987456321546987754632154789654255654488965412365447756213645987564236547
89564123654478965823654789659855471523654789569325874563215647895698563214758965874556
55252655595878521654789215622321779566247863255598985222338148555145698745632123654789
98523697412365478965123574896751362456555565663232236541593578526541236547893576215498
74563236512473214569787536521452365789654712365478965412324567562136459875642365478956
41236544789658236547896598562145698745632145698745632586985471523654789569325874563215
64789569856321475896587455655252654456887985456321457456985474123654789654125478954632
15698745632145698745632586985471523654789569325874563215647895698563214758965874556552
52654456887985456321457456985474123654789654125478954632156987456321456478956985475632
12364587965478852 3

Set nine:

a=12345698745632156478965896532456987456321564789654236598745632156647789569853214563
12364789654758965321456987456321456987456321564789568659325698456321475698532145698563
25541236547893 57621549874563236512

p=896523654789563255654488965412365447756213645987698547563212364587965478852412365245
87956978753652145236578965471236547896541232456698855662325556699878458756987754632154
78965478965423214569856325698564785985471236456987456321564789654712365478965412365698
54741236547896541254789546321569874563214564789569854756321236458796547885241236524587
95697875365214523657896547123654789654123245669885566232555669987845875698775463215478
96547896542321456985632569856478598547123645698745632156478965471236547896541236569854
74123654789654125478954632156987456321456478956985475632123645879654788524123652

458796587452146523698547123654789568563214569874563212365478998523697412365478965478965823658965478541236547896523654789563255654488965412365447756213645987564236547895641212

n=47896582365896547854123654789652365478956325565448896541236544775621364598756423654789564123654478965823654789659856214569874563214569874563258698547152365478956932587456321564789569856321475896587455655252654456887985456321457456985474123654789654125478954632156987456321456 64789569854756321236458796547885236

Table 1: Comparisons with existing methods (where times are in milliseconds)

Size of Exponent (in bits)	Repeated Square Multiplication	M-Ary	Sliding Window (WS=3)	Sliding Window (WS=5)	Sliding Window (WS=7)	Our
1646	0.203	0.203	0.422	1.75	9.781	0.203
826	0.109	0.14	0.219	0.906	4.922	0.109
1392	0.218	0.235	0.5	1.985	10.5	0.203
449	0.11	0.078	0.172	0.672	3.75	0.78
558	0.078	0.078	0.071	0.688	3.734	0.078
630	0.156	0.125	0.297	1.203	6.266	0.093
1693	0.25	0.219	0.531	1.875	9.438	0.218
1692	0.531	0.188	0.328	1.602	5.453	0.203
1695	0.234	0.219	0.453	1.422	7.031	0.187

(SW=Sliding Window, WS=Window Size)

In Table 1, we compare our method with some existing methods, like repeated square, M-ary, Sliding Window methods. We note that some cases our method takes the same or less time than other existing ones, but sometimes it takes more time than other ones. In our method, we calculate all possible combinations of exponents to find maximal matching. So for some large exponent, it takes less time than other existing methods. It can be improved by applying parallel computation on ours maximal matching method.

4. CONCLUSION

In this paper, we introduce a new method of finding a^p mod n. Our proposed method becomes more efficient than the repeated square, M-Ary, Sliding Window methods. If we apply parallel computation procedure, pre computation for finding maximal matching will take less time and hence it will take less time to compute modular exponent. The method will be very much efficient when we get maximal matching on the exponent.

References

[1] D. E. Knuth: "The art of computer programming," Vol. 2, Seminumerical algorithms, Third Edition, Addison Wesley, 1996.

[2] P. De Rooij: "Efficient exponentiation using precomputation and vector addition chains," Advances in Cryptology Eurocrypt' 94, Lecture Notes in Computer Science, Vol. 950, Springer-Verlag, 389-399, 1995.

[3] P. Downey, B. Leong and R. Sethi, Computing sequences with addition chains, SIAM Jl. of Computing, Vol. 10, 103-121, 1981.

[4] Nadia Nedjah and Luiza de Macedo Mourelle: ”Fast pre-processing for the sliding window method using genetic algorithm,” International Journal of Computers, Systems and Signals, Vol. 4, No.2, 11-21, 2003.

[5] Noboru Kunihiro and Hirosouke Yamamoto:”New methods for addition chain,” IEICE TRANCE. FUNDAMENTALS, vol.E83-A, No.1, 60-67, 2000.

[6] Mathias Ravn - mr@daimi.au.dk Casper Lund Thomsen - casper@daimi.au.dk Lars Vadstrup Hansen lvh@daimi.au.dk:”Fast Exponentiation ”

[7] EDWARD G. THURBER:”Efficient generation of minimal length addition chains,” SIAM J. COMPUT, Society for Industrial and Applied Mathematics Vol. 28, No. 4, pp. 1247-1263, 1999.

[8] K. De Jong, and W. Spears: ”Using genetic algorithms to solve NP-complete problems. In Schaffer,” J. D., editor, Proceedings of the Third International Conference on Genetic Algorithms, pages 124132, Morgan Kaufmann, San Mateo, California, 1989.