Computing square root of a large Positive Integer

Yumnam Kirani Singh,
C-DAC, IIPC Building,
NIT Campus, Silchar, Assam.
Email: yumnam.singh@cdac.in

***Abstract*—Dealing in large number is of interest in asymmetric key cryptography using RSA. The security of RSA is solely based on difficulty of factorization of a large number. Factoring a number requires finding all divisible primes less than or equal to the square root of the number. Proposed here is a new algorithm to compute the square root of large positive integer. The algorithm is based on the implementation of long division method also known as manual method we usually use to find the square root of a number. To implement the long division method, the given number is first represented in a radix-10 representa and then Bino's Model of Multiplication is used to systematically implement the long division method. A representa is a special array to represent a number in the form of an array so as to enable us to treat the representas in the same way as we treat numbers. This simplifies the difficulty of dealing large numbers in a computer. The proposed algorithm is applied to the RSA–challenge numbers for factorization. The square roots of the challenge numbers can be computed easily in less than a second. The square roots of first few challenge number and last few challenge number are also provided, which may be used for factorization of corresponding challenge number.**

***Index Terms*—Asymmetric key cryptography, Bino's Model of Multiplication, Large number manipulation, Long division method, Prime factorization, RSA challenge numbers, Representa, Square root computation.**

I. INTRODUCTION

Dealing in large number has been the interests of researchers working in asymmetric key cryptography, network and information security since the development of RSA. This is because RSA based public key crypto-system widely accepted in digital signature [8] and authentication on the web technology [5] requires dealing with large numbers. The security of RSA is solely based on difficulty of factorization of a large number. Factoring a small number is trivial but factoring a very large number is a computationally intensive and infeasible task to complete in a reasonable time frame. This is because factoring a large number in a computer has two main problems. The first problem is dealing of large numbers which is to be done in special ways as such large number cannot be represented as a value in a variable of a programming language and the programs or algorithm written for small numbers cannot be directly applied to such large numbers. The second problem is requirement of large memory space and (or) very long execution time for running a factorization algorithm. The first problem is somewhat simpler and can be solved by using the notion of representa [12]. Representa is a special way to represent number in the form of an array, which can be treated in the same way as we treat numbers. Representa simplifies the development of algorithms for arithmetic operations for large numbers. The second problem is more complex and its solution requires development of new efficient algorithms. The development of square root algorithm in this paper can be considered as a step towards developing efficient algorithm for factorization (and primality testing) of large number. To factorize a number, we have to find all primes less than the number, which divide the number. Finding all primes less than a number itself is computer intensive in terms time and memory requirement when the number is very large. We can improve the factorization if significantly, if we can find the square root of the number to be factorized. In [9], square root is used in the General trial division method, Fermat Factorization method for factoring a composite number into two primes of similar size to reduce the running time. Some of the algorithms suggested for factoring RSA numbers can be found in [1,2,4,11].

The most popular method for finding square root electronically is the iterative method based on Babylonian algorithm or Newton-Rapson method or its variants [3, 6,7,8]. But these algorithms are suitable for small numbers which are within the byte size limitations. These algorithms iteratively approximate the roots until a specified precision is achieved and at each iteration the given large number is divided by newly approximated root. Division of two large numbers is computationally difficult task which requires special algorithms. So, the direct implementation of iterative approximation method for finding square root is not possible.

A new algorithm is proposed here to compute the square root of an arbitrarily large number which does not require iterative approximation but at the same time gives best possible precision. The long division method, we manually use in finding square root is such an algorithm. This paper implements the long division method using the concepts of representa and Bino's Model of Multiplication.

The rest of the paper is organized into four sections. Section-II describes the squaring of a large number using Bino's model of Multiplication (BMM). This can be used to test the correctness of the result of square root and is used to explain why Long Division Method (LDM) works to find the square root of large number. Section III describes the long division method of computing square root of large integer. An algorithm to compute the square root based representa is presented. The experimental result is given in section IV and conclusion in section V.

II. SQUARING A LARGE NUMBER USING BMM

Bino's model of multiplication is generalized multiplication model for multiplication of numbers, polynomials and arrays. To multiply two numbers, numbers are represented in the form of special arrays called representa depending on a specific radix or base, which is power of 10. In a represent of radix-10, each element of the representa must be a reminder of 10. In a representa of radix-100, each element is a reminder of 100. In this way, in a representa of radix-1000, the elements are reminder of 1000. Representing a number in higher radix, saves, significant amount of memory and processing time. However, in this paper, we will be dealing with representa of radix-10, for easy understanding of the explanation. Once the numbers to be multiplied are represented in representa of same radix (10, here), multiplication terms are computed. The actual result of multiplication can be easily obtained from the multiplication terms adjusting the carries in the multiplication terms. More on representa arithmetic and Bino's model of multiplication can be found in [13].

According to the Bino's model of multiplication, when two representas each of length m is multiplied, the number of multiplication terms n = 2m-1, which is odd. That is, for an n-digit number, when n is odd, the number of digits in the integer part of its square root is (n+1)/2. But sometimes leftmost multiplication term becomes two digits, the total number of terms becomes even. In such a case, the number of digits for squaring a number becomes even. To take into account of such cases, we consider ceil of the half of n as the number of digits in the square root of an n-digit number.

Let us consider some squaring examples using Bino's Model of Multiplication.

A. *Squaring of 2-digit number*

Suppose b_1b_2 is a two digit number which we want to square. We first represent in a radix-10 representa as $B_1 = [b_1, b_2]$. Then, square of the 2-digit number will consist of three multiplication terms. Let us represent the array representing a multiplication term by T, and each element in the array by T_i, (denoting i-th element).

$B_1^2 = T$

Where $T = [T_1, T_2, T_3]$ and the respective terms are $T_1 = b_1^2,\ T_2 = 2b_1b_2,\ T_3 = b_2^2$.

The result of square of 2-digit number will be given by

$(b_1b_2)^2 = 100 \times T_1 + 10 \times T_2 + T_3$

Depending on the values of the two digits, the square of a 2-digit number will have minimum of 3 digits and maximum of 4- digits.

Example-1: Square of 23,

Multiplication Terms are

$T_1 = b_1^2 = 2^2 = 4$

$T_2 = 2b_1b_2 = 2 \times 2 \times 3 = 12$

$T_3 = b_2^2 = 3^2 = 9$

So, $23^2 = 100 \times 4 + 10 \times 12 + 9 = 529$, which is a 3- digit number.

Example-2: Square of 67

Multiplication Terms are

$T_1 = b_1^2 = 6^2 = 36$

$T_2 = 2b_1b_2 = 2 \times 6 \times 7 = 84$

$T_3 = b_2^2 = 7^2 = 49$

So, $67^2 = 100 \times 36 + 10 \times 84 + 49 = 4489$, which is a 4-digit number.

From these examples, we can be seen that any 3 or 4- digit number can be thought of as $b_1^2, 2b_1b_2, b_2^2$.

B. *Square of 3-digit number*

Let $B_1 = [b_1, b_2\ b_3]$ be a radix-10 representa corresponding to a 3- digit number $b_1b_2b_3$. Then, the square of B_1 will have 5 multiplication terms, which are as follows.

$B_1^2 = [T_1, T_2, T_3, T_4, T_5]$

Where, $T_1 = b_1^2,\ T_2 = 2b_1b_2,\ T_3 = 2b_1b_3 + b_2^2,\ T_4 = 2b_2b_3,\ T_5$

The actual result of the squaring the 3-digit number is given by

$(b_1b_2b_3)^2 = 10000 \times T_1 + 1000 \times T_2 + 100 \times T_3 + 10 \times T_4 + T_5$

As there are five multiplication terms, the number of digits in resulting square will be 5 or 6 depending on the values of first two multiplication terms. So, any five or six digit number can be thought of obtained from the five multiplication terms, i.e., $b_1^2,\ 2b_1b_2,\ 2b_1b_3 + b_2^2,\ 2b_2b_3,\ b_3^2$.

C. Square of m-digit number:

Let $B_1 = [b_1, b_2, b_3, \dots, b_m]$ be an m-digit number represented in a representa of base 10. Then, there will be 2m-1 multiplication terms.

$T_1 = b_1^2$,
$T_2 = 2b_1b_2$,
$T_3 = 2b_1b_3 + b_2^2$
$T_4 = 2b_1b_4 + 2b_2b_3$
$T_5 = 2b_1b_5 + 2b_2b_4 + b_3^2$
And So on.

Assuming $b_i = 0 \; for \; all \; i < 1 \; and \; i > m$, we can write any multiplication term, T_k as

$T_k = 2\sum_{i=1}^{(k-1)/2} b_i b_{k+1-i} + b_{(k+1)/2}^2$ if k is odd.

$T_k = 2\sum_{i=1}^{k/2} b_i b_{k+1-i}$ if k is even.

This expression is handy for finding the first m-multiplication terms. However, to find remaining m-1 multiplication term, we can write the following expression.

$T_r = 2\sum_{i=1}^{m-(r+1)/2} b_{m+1-i} b_{r-m+i} + b_{(r+1)/2}^2$ if r is odd

$T_r = 2\sum_{i=1}^{m-r/2} b_{m+1-i} b_{r-m+i}$ if r is even.

Where r denotes any multiplication term greater than m.

If we observe carefully the multiplication terms, we see that except the m-th, there are similarities between first m-1 multiplication terms and last m-1multiplication terms as regards number of individual terms and additions required. The first multiplication term is similar last multiplication term, each of which consists of square term. The second multiplication term is similar to the second to last term, each of which consists of 2 times the product of first two and last two digits respectively. Similarly, third multiplication term is similar to the third to last term. Each term consists of product of product of two digits plus a square term. In general, we can say, any p-th (less than m) multiplication term is similar in form with (L-p)-th term, where L is the last term. We can conveniently use this similarity to develop a simpler and shorter algorithm for squaring a multi-digit number.

D. Algorithm for squaring m-digit number

Let B be radix-10 representa of m-digit number. That is, m=Length of the representa of base length 10.

```
T= an array of length L to store L multiplication terms
L=2*m-1;
For i=1 to m-1
lowerhalf=0;
upperhalf=0;
if (i%2)
        for j=1 to (i-1)/2
                lowerhalf=lowerhalf + B(j)*B(i+1-j);
                upperhalf=upperhalf + B(m+1-j)*B(m-1+j)
        end
T(i)=lowerhalf+B((i+1)/2)* B((i+1)/2);
T(L-i)=upperhalf + B((i+1)/2)* B((i+1)/2);
Else
        For j=1 to (i/2)
                lowerhalf=lowerhalf+ B(j)*B(i+1-j);
                upperhalf=upperhalf + B(m+1-j)*B(m-1+j)
        end
T(i)=lowerhalf;
T(L-i)=upperhalf;
End
```

The actual result of squaring is obtained from multiplication terms T by successively adding the carry from the last term to successive terms on the left till the first term.

If we observe the relation between length of a represent and the number of multiplication terms, when the representa is squared, we can predict that the square root of any n-digit number will be a k-digit number, where k =$\lceil n/2 \rceil$, i.e., ceiling of half of n. That is, square root of any 1 or 2-digit number will be a 1-digit number. Square root of any 3 or 4-digit number will be 2-digit number and so on.

III. COMPUTING SQUARE ROOT USING LDM

The long division method also known as manual method is the method, we generally use, to compute the square root of a number. This method has not been implemented as an algorithm to compute square root of a number. Instead other approximation or estimation method has been used. The main reason why this method has not been implemented as an algorithm is because the underlying theory why this method works has not been explained properly. Some tries to explain it using squaring of polynomials. But the polynomials representation and number representation are different and so, the explanation of the process is not clearly understandable for implementation purpose. BMM can be used to explain why how LDM works for finding the square root of a number.

Before explaining LDM, let us examine the following multiplication terms of squaring a 1, 2, 3, and 4-digit numbers in their representa forms.

$[b_1]^2 = [b_1^2]$,
(only one multiplication term, 2*1-1=1)
$[b_1, b_2]^2 = [b_1^2, 2b_1b_2, b_2^2]$,
(three multiplication terms, 2*2-1=3)
$[b_1, b_2, b_3]^2 = [b_1^2, 2b_1b_2, 2b_1b_3 + b_2^2, 2b_2b_3, b_3^2]$
(Five multiplication terms, 2*3-1=5)
$[b_1, b_2, b_3, b_4]^2 = [b_1^2, 2b_1b_2, 2b_1b_3 + b_2^2, 2b_1b_4 + 2b_2b_3, 2b_2b_4 + b_3^2, 2b_3b_4, b_4^2]$
(seven multiplication terms, 2*4-1=7)

If we observe carefully, we see that the square of digits occur in odd multiplication terms. That is, 1st, 3rd, 5th etc. multiplication terms contain b_1^2, b_2^2, b_3^2 etc. Finding square root

from the multiplication terms of a square of a number requires a systematic process to eliminate the square of the digits in their order of occurrence i.e. from left to right in long division method.

Let us examine the some cases of finding square roots of some numbers.

Finding the square root of 1-or 2-digit number is trivial. It will be any digit from 0 to 9.

A. *Square root of 3 or 4-digit number*

Square root of 3 or 4-digit number will be a 2 digit number. We know that squaring 2-digit number results in three multiplication terms. So, the problem of finding the square-root of a 3 or 4-digit number is finding the two digits from the three multiplication terms.

Let us assume that the 3or 4-digit number is represented by three multiplication terms obtained when a 2-digit number is squared. The first term is b_1^2, which can be obtained by multiplying b_1 with b_1, as shown in the upper left part of Figure-1. Then, b_1^2 is subtracted from the multiplication terms and the first multiplication term is eliminated. So, b_1 becomes the first digit of the square root, which is written at the top. After eliminating the first multiplication terms, we are left with two multiplication terms, $2b_1b_2$, , b_2^2. They are brought down in the next line. The first digit of the root is multiplied by 2 which is a factor required to eliminate the second multiplication term. Also, at the same time we have to eliminate the third term. So, multiply $2b_1$ and $b_2$by b_2 to get the second and third term as $2b_1b_2$, , b_2^2. Subtract the terms from the second row and register b_2 as the next digit of the square root by writing it on the right side of the first digit. Thus, we can get the square root of 3 or 4- digit number.

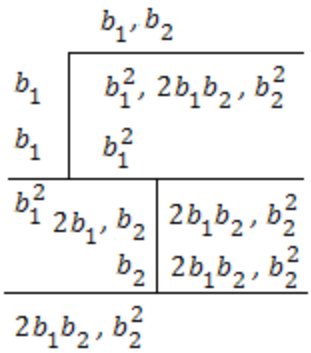

Figure-1: Process for finding square root of a 3 or 4-digit number.

B. *Finding square root of 5 or 6- digit number*

To find the square root of 5or 6-digit number, we can consider the number as the five multiplication terms when a 3-digit number is squared. The process remains the same as described in section III.A. First eliminate the first square term to get the first digit of the square root. Multiply the first digit of the root by 2 to get $2b_1$. The next square term to eliminate is b_2^2. So append the digit b_2 to $2b_1$ and multiply it by b_2 i.e., multiply $2b_1, b_2$ by b_2 to get $2b_1b_2$, b_2^2. Register b_2 as the second digit of the root and subtract $2b_1b_2$, b_2^2 from the second and third multiplication terms. The difference $2b_1, b_3$ corresponding to third term multiplication is term is brought down along with the fourth and fifth multiplication terms. Multiply the first two digits of the square root by 2 then append b_3 and multiply by b_3, i.e., multiply $2b_1, 2b_2, b_3$ by b_3 to get $2b_1b_3$, $2b_2b_3$, b_3^2, which the same three terms we are interested to eliminate. Register b_3 as the third digit of the root. Thus, we compute the square roots of the 5 or d- digit number. The described process is shown in figure-2.

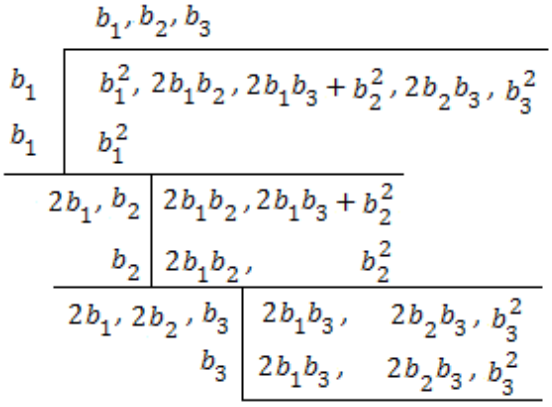

Figure-2: Process for finding square root of a 5 or 6-digit number.

C. *Finding Square root of 7or 8- digit number*

A 7 or 8-digit number can be considered as the seven multiplication terms obtained when a four digit number is squared. The process of getting square root from the seven multiplication terms is the same as described in previous two subsections. First get the digit of the square root by finding suitable square of digit corresponding to the first multiplication term and eliminate the square term. Subtract the square term, and bring down the next two multiplication terms. Multiply the first digit of the root by 2, append the suitable digit corresponding next square term to eliminate and multiply by the digit. Subtract the result from the brought down terms in the next line. The process continues till all the multiplication terms are processed. The overall process of computing square root from 7 multiplication terms representing a 7 or 8-digit number is shown in figure-3.

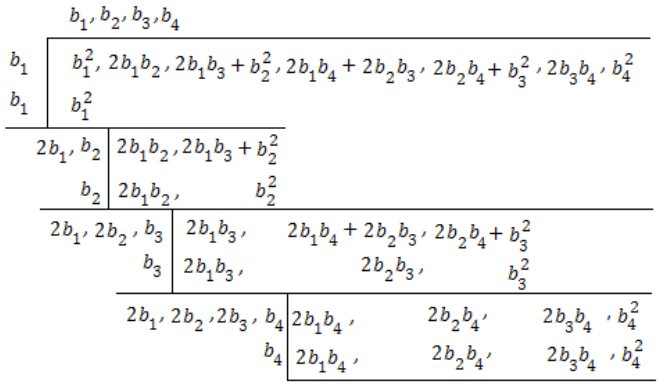

Figure-3: Process for finding square root of a 7 or 8-digit number.

D. Algorithm for computing square root

We have seen that while computing square root of a multi-digit number from multiplication terms, except the first term, we brought down two consecutive multiplication terms to find the next digit of the square root. That is why we usually mark the digits of multi-digit number in group of two starting from the right end before computing square root using long division method. Grouping the digits in a group of 2 can be done by representing the given number as radix-100 representa. So, the algorithm is as follows.

```
Let X= radix-100 representa representing the multi-digit
number whose square root is to be computed.
R=radix-10, representa representing result, i.e., square root.
C=radix-10, representa to find next digit of the root.
D=radix-10 representa to act as partial dividend.
Set D=X(1)
Assign R(1)=integer part of square root of D.
C=R(1)*R(1);
While (last element of X not processed)
        Compute D=Subtract( D, C);
        Append the next element of X to D.
        C=Multiply(R,2);
        For I=1 to 9
                Append I to C;
                C=Multiply(C,I);
                if C > D
                        Append I-1 to R
                        break;
                end
                if I==9
                        Append I to R;
                end
        end
end
```

In the algorithm, the first element of R is obtained using the square root function available to compute the square root of a small number. Since the first element of X will be a number less than 100, finding its square root would have no problem. The other digits are computed inside the while loop of the algorithm in the same way, as we do in the long division method.

IV. EXPERIMENTAL RESULT

Both the squaring and the square root algorithm has been implemented and tested using various inputs. It has been found that both algorithms give correct result as expected. The squaring algorithm has been useful to check the correctness of the square root values of large multi-digit numbers obtained when the proposed square root algorithm is executed. However, in the experimental result no squaring values are shown as squaring is not the main concern of the paper. The square root algorithm has been applied to all RSA to find their roots. But only square roots of some few and last few RSA factoring challenge numbers are given below. The number of digits in the square root of any RSA-challenge number is the half of the number specified in the challenge number. That is, the number of digits in the square root of RSA-100, is 50 and that RSA-110 is 55. Similar is true for all other challenge number except the last challenge number RSA-2048, where the 2048 is not the number of digits in the original challenge number. The number of digits in RSA-2048 is 617, which is odd. So, the number digits in the resulting square root is (617+1)/2=309, as expected.

RSA-100: (50 digit)
39020571855401265512289573339484371018905006900194
RSA-110: (55 digit)
59828282759683040041003178541182303136857938437236
09073
RSA-120:
47645616933295906619208683305765656673387250844370
0132335510
RSA-130:
42509788151523465407452697662505294703186899165086
434856734890373
RSA-200:
52912979420196447553235400217339415005716618867127
19898054010160087776775983652765778987171296799172
RSA-240:
35301609989024407676277184230754669625591247945328
83216133958719910993762631569718030063022214786064
77231562718336689398
RSA-500:
43556792050708080771412473257868843029627065400313
13841309486297616737283386446299363546474925651968
68686656924337842186248604188842298275798437781768
60622720538181254030819484471364309805836725916358
64958623804596201935416553105746452696931568061907

RSA-2048: (309 digits)
15873219105039120417448250866106300757935846344480
97157957266277535799700807499484042786432595681011
32671402056190021464753419480472816840646168575222
62893467140573921347743953387048979103897316683406
87362340203616648202669877269194533568241380073819
85796493621233035112849373047484148339095287142097
834807844

V. CONCLUSIONS

A generalized algorithm for computing the square root of a large number is proposed. Also, a new algorithm for computing the square of a positive integer is proposed. The algorithm is based on the long division method for finding the square root, implemented using represent and Bino's model of multiplication. The general notion that long division method is manual method for computing square root is proved incorrect. The long division method is effective method for computing square root of an arbitrarily large number giving the best precision of every digit computed. The proposed algorithm is simple and fast. It has been applied to RSA-challenge number, the square root of the first few and the last few of the challenge numbers are also provided.

REFERENCES

[1] B R Ambedkar, Ashwani Gupta, PratikshaGautam and SS Bedi, "An Efficient Method to Factorize the RSA Public Key Encryption" Proc. International Conference on Communication Systems and Network Technologies (CNST 2011) Katra, Jammu, India, pp. 108-112, 3-5 June 2011.

[2] Sattar J. Aboud and Evon M. Abu-Taieh, A New Deterministic RSA-Factoring Algorithm, IEEE J. J. Appl.Sci., 2006: Vol. 8, No. 1, 54-66.

[3] David Fowler and Eleanor Robson, "Square Root Approximations in Old Babylonian Mathematics," Historia Mathematica, 25 (1998), 366-378.

[4] Kostas Bimpikis and Ragesh Jaiswal, "Modern Factoring algorithms", (online) http://www.cs.columbia.edu/~rjaiswal/factoring-survey.pdf.

[5] Housley et al. "RFC 2459: Internet X.509 Public Key Infrastructure Certificate and CRL Profile." January, 1999. Available: http://www.faqs.org/rfcs/rfc2459.html

[6] Liang-Kai W, Schulte MJ. *Decimal Floating-Point Square Root Using Newton-Raphson Iteration*. 16th IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP). 2005: 309-315.

[7] Kosheleva O. *Babylonian Method of Computing The Square Root: Justifications Based on Fuzzy Techniques and on Computational Complexity*. Annual Meeting of the NorthAmerican Fuzzy Information Processing Society (NAFIPS). 2009: 1-6.

[8] Thomas J. Osler, "Extending the Bbylonian algorithm", Mathematics and Computer Education, Vol. 33, No. 2, 1999, pp. 120-128.

[9] Murat SAHIN "Generalized Trial Division", Int. J. Contemp. Math. Sciences, Vol. 6, 2011, no. 2, 59 – 64.

[10] R. L. Rivest, A. Shamir and L. M. Adleman, A method for obtaining digital signatures and public-key cryptosystems, IEEE Communications of the ACM (2)21(1978),120-126. J..

[11] Lal N., Singh A.P, Kumar S., "Modified trial division algorithm using KNJ-factorization method to factorize RSA public key encryption," Contemporary Computing and Informatics (IC3I), 2014 International Conference on , vol., no., pp.992,995, 27-29 Nov. 2014

[12] Yumnam Kirani Singh , "On Some Generalized Transforms for Signal Decomposition and Reconstruction " Ph.D. dissertation, CVPR Unit, ISI, Kolktata. 2006.

Brief about the Author:

Completed Master's Degree in Electronics Science from Guwahati University in 1997 and got Ph. D. degree from Indian Statistical Institute, Kolkata in 2006. Served as a lecturer in Electronics in Shri Shankaracharaya College of Engineering & Technology from Jan, 2005 to May, 2006. Joined CDAC Kolkata in May 2006 and worked there before coming to CDAC Silchar, in March 2014. Developed Bino's Model of Multiplication, ISITRA, YKSK Transforms and several other image binarization and edge detection techniques. Interested to work in the application and research areas of Signal Processing, Image Processing, Pattern recognition and Information Security. Published several papers in national and international journals and conferences