JCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Volume-3, Issue-7 E-ISSN: 2347-2693

Square of an Enormously Large Number

Harsh Bhardwaj

Maharaja Agrasen Institute of Technology
GGSIPU, New Delhi - India

www.ijcseonline.org

Received: Jun/23/2015 Revised: July/06/2015 Accepted: July/21/2015 Published: July/30/ 2015

Abstract- In this world of innovative and effective manipulations and calculations, the requirements for developing a way to carry out the computation in a more algorithmic form arises. The requirement to understand the logic behind a computation doesn't come up as an issue unless you type in a really huge number, like a number including 100-digits or so. When this kind of number is required to be processed upon, a more algorithmic way of carrying out the computation is more presentable. A process for innovative and easily manipulated computations is required to be produced, in order to increase the productivity, convenience, ease of use and fairly deal with budgetary concerns. This document provides general practices, procedures and tools for creating an innovative computation of square of extremely large numbers. It is aimed for engineers, Mathematical gurus, algorithm geeks, who are assumed to possess basic knowledge regarding *what is meant by square of a number.* It addresses basic knowledge about how to calculate the square using conventional means and how the improvement can be made to produce an algorithmic way of doing the same.

Keywords— *Square of a number, Time-Complexity, Algorithm Design*

I. Introduction

Let's start with basics of exponentiation. Exponentiation can be referred to as the procedure of multiplying a given number 'n' by itself for 'm'-number of times. The number to be multiplied by itself can be called as 'base', i.e. 'n' here is 'base', while the number *how many times it is to be multiplied*, can be called as 'exponent', i.e. 'm' here is 'exponent'. The symbol required to represent the exponentiation can be given as '^', called as “carat”. Thus, exponentiation can now be shown in mathematical terms as 'n ^ m', also called as 'n to the power m'.

This simple concept of multiplication of a number by itself has played a very significant role in the world of mathematics. Polynomials are all based upon the base-exponent relationship of variables. These polynomials build up the infrastructure of Algebra.

When the exponent is made to be nothing but 2 (*two*), the exponentiation starts to be termed as calculating the square of a number. Now, the base 'n' becomes a k-digit number and 'm' becomes 2. This is represented as n ^ 2 and termed as 'n to the power two' or in more familiar words 'square of n'. From calculating the area of a circle, to equating popular Energy-mass equation of Albert Einstein, the exponentiation has developed its way through into being one of the very crucial parts of the whole mathematics.

In Computer Technology, the requirements for developing an algorithmic way of computing the square of a given number have evolved from various methods to various others. There exist many of them that are effective enough to compute the square of very large numbers, but they are not as efficient as the method we are about to discuss.

After having gone through this paper, you will be able to calculate the square of a given number with an ease never mentioned before and the algorithm discussed here would provide a way of computing the square of an extremely large number, i.e. consisting of several hundred digits.

II. Conventional Method

A. N to the power two

A very convenient method, as the name suggests is to multiply the number with itself. For example, if we have a number 'n' equals to 45 and it is supposed to be calculated square of, then we need to multiply 45 with itself, i.e. 45, in order to get the result 2025, as the square of this number. In this approach, it is to be understood what it actually means to multiply a number.

B. Multiplying a number

What it actually means to have a number multiplied is adding the number as many times as the number it is required to be multiplied with. Considering the previous example, 45 ^ 2 depicts that 45 is to be multiplied with 45 and in order to multiply 45 with another 45, it would have to be added with itself 45 times, i.e. 45 + 45 + 45 + … + 45 (45 times).

The square of a number can thus be calculated in this manner. But, it can be easily seen how hectic this procedure becomes when a number of sufficiently large number of digits is encountered. This conventional approach is the basic and the most absolute approach but computer technology requires an algorithmic way.

III. DEVELOPING A MEANS

It has to be seen how the advancement in computer technology has brought the computational system to algorithmic system. All the calculations and logics are now carried out in an algorithmic way. The programming language used here is Python and the module to keep track of the time element is 'time'. Recursive calling of function has been used in order to keep the complexity of the algorithm as low as possible.

A. Time module of Python

This module can be used in order to keep track of the time taken by a particular instruction or a set of instructions in order to build up an exact scenario of how long does an instruction or a set of instructions take to execute. This can be shown to be done as:

```
time1 = time.time()
# Your instructions go here
time2 = time.time()
time2 = time2 – time1
```

It is clear that a variable called 'time1' holds the time instance that occurred just before instructions had started executing and 'time2' holds the time instance that occurred right after the execution of instructions was done. Subtracting the first variable from second gives the time required by the set of instructions to execute. In order to use this module, programmer has to import the module exclusively.

B. Recursion

Recursion can be defined as the way in which one method keeps passing a lower set of parameters to itself until a base case is encountered in which function returns primitive values that are predefined in the function or are too obvious to guess. Recursion in this manner can be thought of as a way in programming languages by which one method makes a call to itself with a lower set of values. This can be represented as:

```
def meth( param1, param2, … , paramk ):
    # here goes the base case
    meth( param_smaller1, param_smaller2, … , param_smallerk )
```

It is clear from the example above that 'meth' method is calling itself with lower set of values.

IV. GIVING THE ALGORITHM

A. Algorithm

Algorithm defines a set of sequential rules required to be followed in order to solve a particular problem. The algorithm for calculating the square of a given number can be given as:

1. *if the number is 1-digit, then*
2. *return the square of number*
3. *if modulo-10 of the number is 0, then*
4. *recursively call the algorithm over 1/10th of the number*
5. *return the result obtained after multiplying it with 100 or appending two zeroes at the end of the result*
6. *else*
7. *compute the nearest but smaller number that is exactly divisible by 10, save it in variable 'num'*
8. *recurse over the obtained number and store the result in a variable, say 'sq'*
9. *i = 1*
10. *while units digit of the number is greater than 0*
11. *sq = sq + num * 2 + i*
12. *increase i by 2*
13. *decrease units digit of the number by 1*
14. *return sq*

Above algorithm can be explained in the following manner:

Line 1 is the base case for the recursive algorithm, i.e. whenever the length of the number is 1, it should simply return the square of the number. This can be done as simple multiplication of the number by itself or by listing every single case from 0 to 9 and returning the corresponding value.

Now, there may occur two cases, i.e. when the number is exactly divisible by 10 and when the number is not exactly divisible by 10. When the number is exactly divisible by 10, then it is known by practice that the square of this number will be divisible by 100. Thus, we now only need to find the square of the 1/10th of the number, and when the result comes, we will multiply it with 100 for the correct answer. In programming language that support string operations, that most of the programming languages do, two zeroes can be appended at the end of the result obtained. This implementation can be pointed out in the algorithm on line 3, 4 and 5.

In *line 3*, modulo-10 of the number is checked. If it is found to be zero, a recursive call on 1/10th of the number is made and the result is simply returned back.

When the number is found to be not divisible by 10 and leaves a remainder, this remainder can be stored back for further utilization. Compute the nearest but smaller 10's multiple of the number as done in *line 7*. Recursive call can be made over this number and the result is to be stored in a variable, say 'sq'. This is done in *line8*.

In *line 9*, a variable called 'i' is assigned a value of 1. The loop on *line 10* sustains until the remainder that we had previously stored is greater than zero.

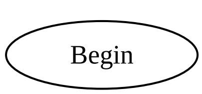
The main part of this whole algorithm resides in the next three lines from *line 11* to *line 13*. The explanation for this can be stated by an example.

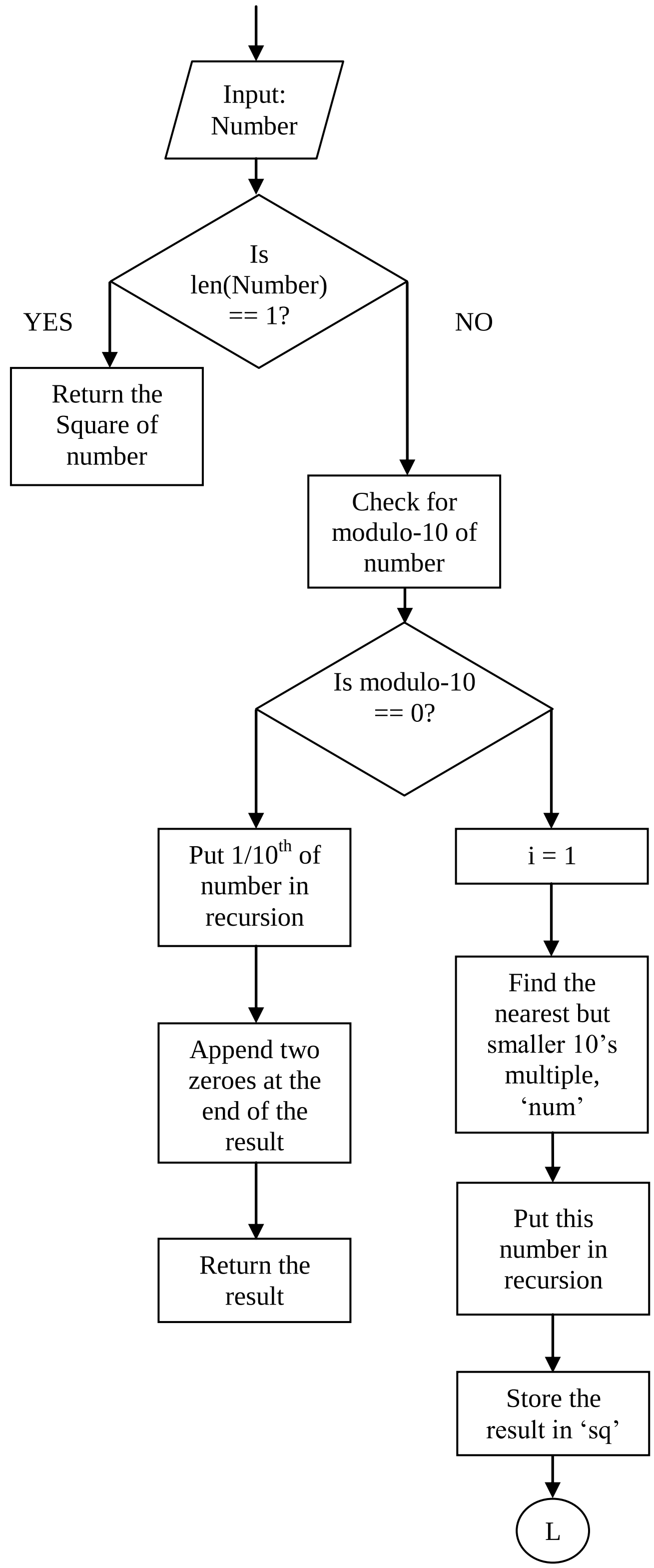
Let's consider a number, 34. The nearest but smaller 10's multiple to this number is 30. Square of 30 can be calculated using recursive calls as described above. In order to calculate the square of 34, four iterations are carried out inside the while loop. In first iteration, square of 30, i.e. 900, is added up with 'twice of 30' and 1, to yield the square of 31 as 961. In second iteration, 961 is added up with 'twice of 30' and 3, to yield 1024, i.e. the square of 32. In the next iteration, 1024 is added up with 'twice of 30' and 5, to yield the square of 33 as 1089. In the fourth iteration, 1089 is added up with 'twice of 30' and 7 to give the final answer of square of 34, i.e. 1156.

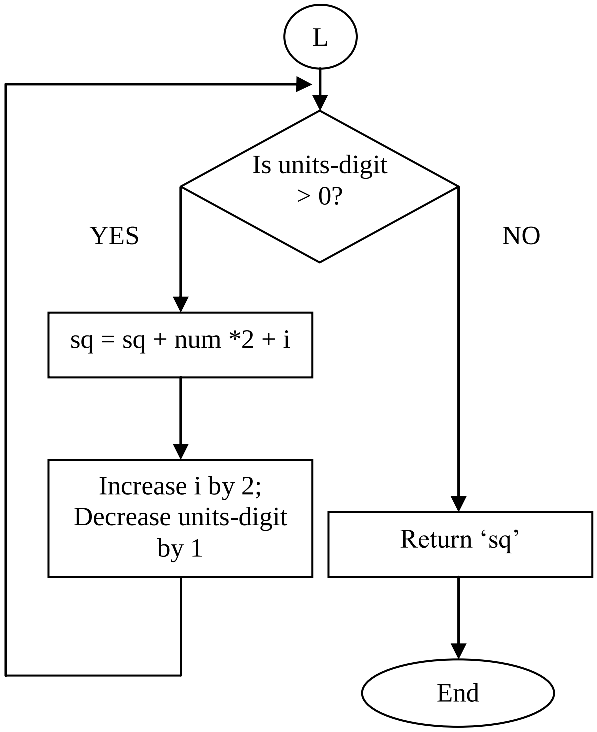
Thus, it can be shown that the algorithm applies to all sort of inputs and returns the square of number in recursive manner.

The square calculated can be returned as in *line 14*.

B. Flowchart







C. Corectness of the Algorithm

Correctness of the algorithm can be proved by having proved the applicability of the algorithm over all sort of inputs that the algorithm claims to be working upon. This is where the working of the inner of the 'while' loop is explained.

This method of computing the square of a number can also be implemented in thought process for human beings for faster calculation of squares of small numbers.

Let's consider a number, 57. The requirement is to find the square of this number. The unit's digit being 7 is required to be stored in a variable, say 'udigit'. The nearest but smaller 10's multiple is found to be 50. Calculate the square of 50. The square of 50 can be easily calculated as two zeroes are required to be placed after the square of 5. This is done in the first recursion in the algorithm given above. This ensures that the square of a number ending with a zero is calculated by eliminating the zero in the end and calculating the square of the remaining number by recursion and multiplying the result by 100 (or placing two zeroes in the end of the result in thought process).

Once the square of 50 is calculated as 2500, the while loop gets activated and the processing is done as follows:

In the first iteration, add 50 * 2 = 100 and 1 to 2500, to make it 2601, this is the square of 51. In the second iteration, add 50 * 2 and 3 to 2601, to make it 2704, this is the square of 52. By proof of induction it can be shown that proceeding in this manner, the square of a number can be calculated in an algorithmic way.

D. Implementation

Every algorithm is required to have a working model in order to test the applicability and complexity of it. The following program is written in Programming language and gives a working model of the algorithm given here.

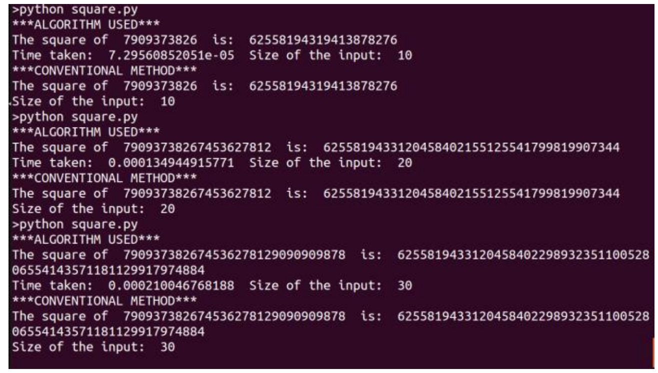
```
import time

num = 99999

def func(num):

        if num == 0:
                return 0

        if num == 1:
                return 1

        if num == 2:
                return 4

        if num == 3:
                return 9

        if num == 4:
                return 16

        if num == 5:
                return 25

        if num == 6:
                return 36

        if num == 7:
                return 49

        if num == 8:
                return 64

        if num == 9:
                return 81

        num1 = num % 10
        if num1 == 0:
```

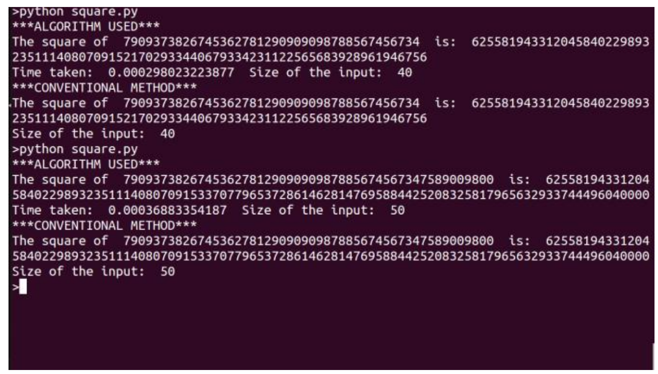
```
            num2 = num / 10
            num3 = func(num2) * 100

            return num3
        else:

            num2 = num - num1
            num3 = func(num2)

            sq = num3
            i = 1

            while num1 > 0:

                sq = sq + num2 * 2 + i

                i += 2

                num1 -= 1

            return sq
time1 = time.time()

result = func(num)

time2 = time.time()

time2 = time2 – time1


print "***ALGORITHM USED***"

print "The square of ", num, " is: ", result
print "Time taken: ", time2, " Size of input: ",
len(str(num))

result = num ** 2

print "***CONVENTIONAL METHOD***"
print "The square of ", num, " is: ", result
print "Size of input: ", len(str(num))
```

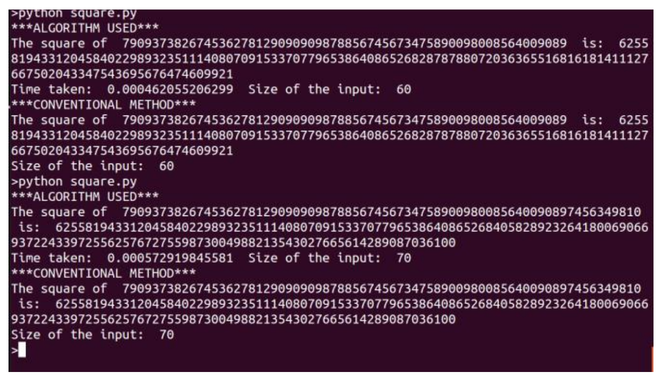
It should be clear from the program given above that time module of Python language has been used in order to keep track of the time taken by the algorithm.

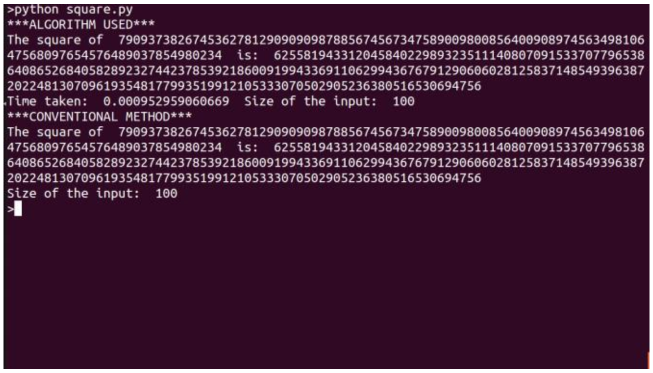
E. Observation

Following screenshots show the time taken by the algorithm to calculate the square of a sufficiently large number. We start with a number consisting of 10 digits and proceed by increasing the number by 10 digits every time.

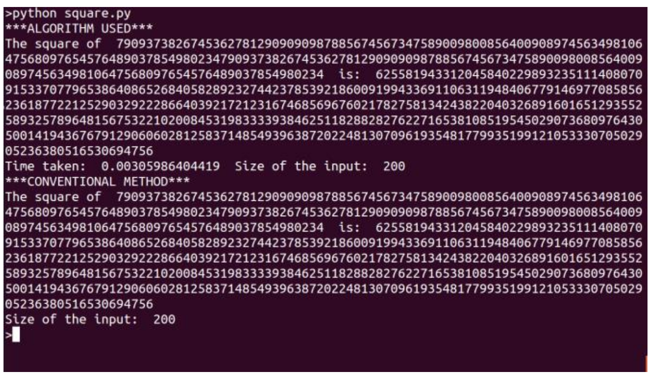


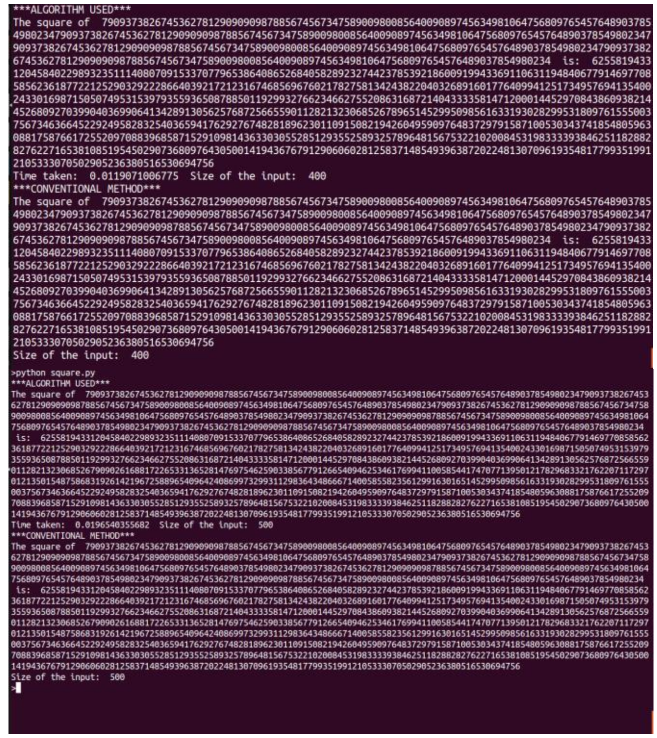




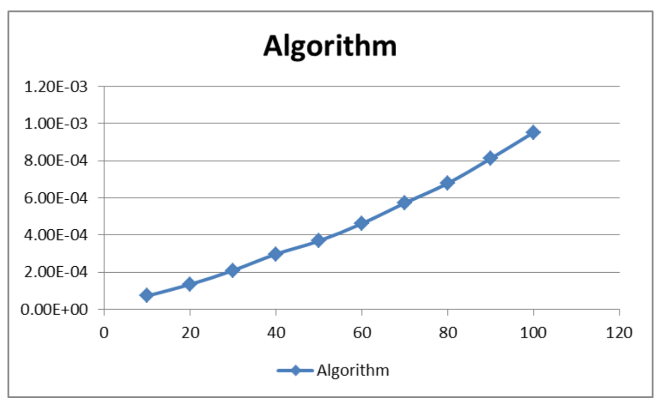


The algorithm also works for very large number, say numbers consisting of 100 digits and more.

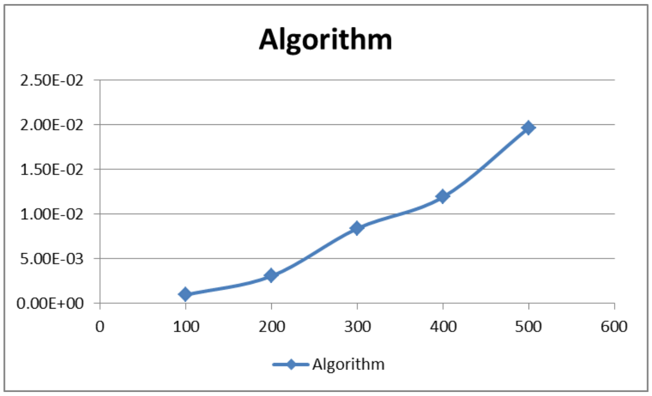




It is clearly seen in the above pictures that the square calculated through the algorithm mentioned in this paper is applicable on any length of number and works perfectly fine for enormously large numbers. A clear vision of the same can be deciphered by presenting a graph holding the amount of time taken on Y-axis with respect to the number of digits of the number on X-axis.



In the graph shown below, the comparison of number of digits with respect to the time taken for calculating the square of a number by using the algorithm is shown. It can be seen how the time taken increases with the increase in number of digits. It should be noted here that input numbers are reasonably large, i.e. consisting of 100, 200, 300, 400 and 500 digits.



F. *Advancement in the Algorithm*

The algorithm can also be modified for the computation of higher exponential powers. This can be achieved by strategically repeating the algorithm until the required result is obtained. A more elaborative explanation for this fact can be given by an example.

Let a base number be '46' and the exponent be '9'. The operation '46 ^ 9' is required to be carried out. This can be done by calculating the square of 46 using the algorithm. This gives '46 ^ 2'. Calculating the square of the result obtained will give '46 ^ 4'. Applying the same algorithm again for calculating the square will give '46 ^ 8'. Now, this result is simply required to be multiplied by another 46 in order to yield the result for '46 ^ 9'. In this manner, higher exponential operations on a number can be dealt with.

G. *Future Scope*

1. The algorithm being used here functions on a pre-determined set of hardware that keeps it working in a conventional way of execution of operations that limits the effectiveness of the algorithm. The speed of calculation can be improved by providing the algorithm with proper hardware. With proper hardware capable of holding the variables and function and heaps used, the algorithm can be proved to be even faster.
2. It can be seen from the fact of execution of the algorithm, that for all the numbers, the nearest but smaller 10's multiple is calculated. For numbers that are more than 5 units more to the 10's multiple, i.e. 57 is 50 (10's multiple) + 7 (unit's digit), the inner loop executes for seven times. This can be reduced by having considered the nearest but larger 10's multiple, i.e. 60 for 57. Now, square of 60 can be calculated as 3600 and twice of 60 can be subtracted from 3600 and 1 is added to get the square of 59, i.e. 3481. In order to find the square of 58, from 3481, twice of 60 has to be subtracted again and 3 has to be added up to the result, to get 3364. In the next iteration, subtracting twice of 60 from 3364 and adding 5 will give the square of 57, as 3249. This reduces the number of iterations from 7 to 3 and thus leads to a better and efficient method of calculating the squares.

V. CONCLUSION

The requirements for computing the square of a given number in an algorithmic manner using recursion have been fulfilled. It can now be concluded that a square of a number can be subjected to be considered as a problem that can be solved by dividing the problem into sub-problem of calculating the square of the nearest but smaller 10's multiple of the given number. This implementation is done in programming by using the concept of recursion.

VI. ACKNOWLEDGMENT

I would like to thank my parents for providing me all sorts of means to have a considerable environment for proper studies. I would also thank the Almighty God, for all the protection and care he has been giving to me. A special thanks to my father, who could not see his son's first authored paper. I would also thank my dearest friend, Yogender Bhardwaj, for introducing me to IJCSE and promoting me to develop this paper. A special thanks to my dearest friends Moksh Gaur, Akash Raman, Aakrisht Aman, for encouraging me to publish my work.

VII. REFERENCES

I would mention here that this discovery is individual to the author only and contribution of no other person or any source is involved. The following references contribute to the development of the program and not the algorithm:

[1] Dive Into Python, Mark Pilgrim, Published 20th May 2004

AUTHORS PROFILE

Name: Harsh Bhardwaj
Education: Pursuing B. Tech. in Computer Science And Engineering from Guru Gobind Singh Indraprastha University, New Delhi
Certifications : Certificate of Accomplishment in Usable Security, Data Mining, Internet Security
Interests : Python Programming, Ruby Programming, Ruby on Rails, Algorithm Design and Analysis, System Secutiry, Cyber Security