Efficient Integer Square Root Computation Using Base-9 Decomposition

[Anshuman Singh]
[Oriental Institute of Science and Technology]
[Satna], [India]
[anshumans1320@gmail.com]

ABSTRACT

Computing the integer square root efficiently is critical for various applications, including numerical computing, cryptography, and computer graphics. Traditional methods such as Newton-Raphson and binary search operate in $O(\log_2 N)$ time. This paper introduces a novel approach, `nineTree`, which leverages a base-9 decomposition strategy to achieve a complexity of $O(\log_9 N)$, making it asymptotically faster. We present the algorithm, analyze its efficiency, and compare it against existing methods. Experimental results confirm that our approach significantly reduces computational steps while maintaining accuracy.

KEYWORDS

Integer Square Root, Computational Mathematics, Algorithm Optimization, Logarithmic Complexity

ACM Reference Format:
[Anshuman Singh]. 2025. Efficient Integer Square Root Computation Using Base-9 Decomposition. In . ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The computation of the integer square root of a number N is a fundamental problem in computer science, with applications in numerical analysis, cryptographic systems, and hardware optimization. Traditional methods such as Newton-Raphson iteration and binary search provide efficient solutions, operating in $O(\log_2 N)$ time. However, these methods involve division and floating-point operations, which may be costly in hardware-constrained environments.

This paper introduces the `nineTree` algorithm, which adopts a novel base-9 decomposition strategy to achieve $O(\log_9 N)$ complexity. By reducing the number of iterations required to compute the square root, our approach provides a more efficient alternative, particularly in scenarios where integer-only computations are preferred.

2 RELATED WORK

Numerous approaches exist for integer square root computation:

- **Newton-Raphson Method:** A widely used approach based on successive approximations, converging in $O(\log_2 N)$.
- **Binary Search Method:** Uses a divide-and-conquer approach to locate the integer square root in $O(\log_2 N)$.
- **Bitwise Methods:** Some implementations use bit-shifting techniques for optimization in hardware-based computations.

Despite these optimizations, existing methods still require a significant number of iterations for large N. Our work introduces a logarithmic-speedup technique based on base-9 decomposition.

3 MATHEMATICAL PROOFS

The algorithm uses inherent mathematical structures and insights to compute integer value of square roots of integers with high convergence rate while maintaining the correctness.

The mathematical insights forming the basis of algorithm are as follows,

3.1 result 1

For every N belonging to the set of all positive integers the value of N^2 is the sum of the first N consecutive odd integers starting from 1. Formally,

$$N^2 = sum_{n=1}^{n=N}(2 * n + 1) \tag{1}$$

where n belongs to the set of all natural numbers. Also,

$$N^2 = (N * (2 * 1 + (N - 1) * 2))/2 \tag{2}$$

that is N^2 equals to the arithmetic sum of N terms starting from 1 and having a common difference 2.

3.1.1 proof. Given N belonging to the set of all positive integers.

$$N^2 = (1 + N - 1)^2 \tag{3}$$

$$N^2 = 1^2 + (N - 1)^2 + 2(N - 1) \tag{4}$$

$$N^2 = 1^2 + 2 * N - 2 + (N - 1)^2 \tag{5}$$

$$N^2 = 2 * N - 1 + (N - 1)^2 \tag{6}$$

The equation 6 proves that every N belonging to the set of all positive integers can be expressed as a sum the Nth odd integer and the square of the integer $N - 1$. Also for every k belonging to the set of all positive integers less than N that is $N - k >= 1$ and $k >= 1$.

$$(N - k)^2 = (1 + ((N - k) - 1)^2 \tag{7}$$

$$(N - k)^2 = 1^2 + 2 * ((N - k) - 1) + ((N - k) - 1)^2 \tag{8}$$

$$(N - k)^2 = 1 + 2 * (N - k) - 2 + ((N - k) - 1)^2 \tag{9}$$

$$(N - k)^2 = 2 * (N - k) - 1 + ((N - k) - 1)^2 \tag{10}$$

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Equation 10 can be recursively solved for all values of k starting from 1 to $N-1$ as (for the sake of illustration only N is assumed to be a large integer which does not become less than 1 for the values of k used in the illustration).

$$N^2 = 2*N-1+(N-1)^2 \tag{11}$$

$$N^2 = 2*N-1+2*(N-1)-1+(N-2)^2 \tag{12}$$

$$N^2 = 2*N-1+2*(N-1)-1+2*(N-2)-1+(N-3)^2 \tag{13}$$

$$\vdots N^2 = 2*N-1+2*(N-1)-1+2*(N-2)-1+\cdots+1 \tag{14}$$

The equation 14 proves the consistency of result 1.

3.2 result 2

The sum of first 3^k odd integers is equal to 9^k for all k belonging to the set of all natural numbers. Formally if,

$$S = sum_{n=1}^{n=3^k}(2n-1) \tag{15}$$

where n and k belong to the set of all natural numbers then it implies

$$S = 9^k \tag{16}$$

3.2.1 proof. Given k and n belonging to the set of all natural numbers. Assume the sum of the first 3^k is given by S then,

$$S = sum_{n=1}^{n=3^k}(2*n-1) \tag{17}$$

$$S = sum_{n=1}^{n=3^{k-1}}(2*a-5+2*a-3+2*a-1) \tag{18}$$

$$\tag{19}$$

where,

$$a = 3*n \tag{20}$$

$$S = Sigma_{n=1}^{n=3^{k-1}}(6*n-5+6*n-3+6*n-1) \tag{21}$$

$$S = sum_{n=1}^{n=3^{k-1}}(18*n-9) \tag{22}$$

$$S = sum_{n=1}^{n=3^{k-1}}9(2*n-1) \tag{23}$$

$$S = sum_{n=1}^{n=3^{k-2}}9^2*(2*n-1) \tag{24}$$

$$\vdots S = sum_{n=1}^{n=3^{k-k}}9^k*(2*n-1) \tag{25}$$

$$S = sum_{n=1}^{1}9^k*(2*1-1) \tag{26}$$

$$S = 9^k \tag{27}$$

Equation 27 proves the consistency of result 2.

4 ALGORITHM

4.1 step 1

Initialize an integer type variable N and store the input positive integer in it. Initialize two more integer type variables *temp* and *unitsize* with values N and 1 respectively.

4.2 step 2

Iteratively divide *temp* by 9 while *temp* >= 9 and for each iteration multiply *unitsize* by 3.

4.3 step 3

Initialize an integer type variable *startTerm* valued 1. Initialize an integer type variable *sum*1 storing the arithmetic sum of first *unitSize* terms starting from *startTerm* and having a common difference of 2 calculated as,

$$sum1 = (unitSize*(2*startTerm+(unitSize-1)*2))/2 \tag{28}$$

also initialize another integer type variable *sum*2 storing the arithmetic sum of first $2*unitSize$ terms starting from *startTerm* and having a common difference of 2 calculated as,

$$sum2 = (2*unitSize*(2*startTerm+(2*unitSize-1)*2))/2 \tag{29}$$

and finally initialize another integer type variable *squareRoot* valued 0.

4.4 step 4

Now there are two cases,

4.4.1 $temp >= 4$. In this case, Reassign *temp* as $temp = N - sum2$ and *squareRoot* as $squareRoot = 2*unitSize$.

4.4.2 $1 <= temp < 4$. In this case, Reassign *temp* as $temp = N - sum1$ and *squareRoot* as $squareRoot = unitSize$.

4.5 step 5

Reassign *startTerm* as $startTerm = 1+2*squareRoot$ and $unitSize = unitSize$

4.6 step 6

Reassign *sum*1 again as,

$$sum1 = (unitSize*(2*startTerm+(unitSize-1)*2))/2 \tag{30}$$

and also *sum*2 as,

$$sum2 = (2*unitSize*(2*startTerm+(2*unitSize-1)*2))/2 \tag{31}$$

4.7 step 7

Now following cases are possible,

4.7.1 $temp >= sum2$. In this case reassign *temp* as $temp = temp - sum2$ and *squareRoot* as $squareRoot = squareRoot + 2*unitSize$.

4.7.2 $sum1 <= temp < sum2$. In this case reassign *temp* as $temp = temp - sum1$ and *squareRoot* as $squareRoot = squareRoot + unitSize$.

4.8 step 8

Reassign $startTerm = 1+2*squareRoot$ and $unitSize = unitSize/3$.

4.9 step 9

Iteratively repeat the steps from step 4 to step 8 until either *temp* or *unitSize* becomes smaller than 1.

4.10 EXPLANATION

Given a positive integer N, a copy of it is made as N_c so that N remains unaltered and consistent for referencing.Then N_c is divided by 9 while $N_c <= 9$, this is done to find to get upper and lower limits of the square roots. If it takes k iterations then we know for sure that $9^k <= N < 9^{k+1}$.

From result 1 and result 2 we know that the integer part of the square root of N say X lies in the range $3^k <= X < 3^{k+1}$.Now,

$$9^{k+1} = sum_{n=1}^{1} 9^k * (6 * n - 1 + 6 * n - 3 + 6 * n - 5) \quad (32)$$

$$9^{k+1} = 9^k * (1 + 3 + 5) \quad (33)$$

where

$$9^k * 1 = (3^k * (2 * 1 + (3^k - 1) * 2))/2 \quad (34)$$

$$9^k * 3 = (3^k * (2 * (1 + (3^k - 1) * 2) + (3^k - 1) * 2))/2 \quad (35)$$

$$(36)$$

and

$$9^k * (1 + 3) = (2 * 3^k + (2 * 1 + (2 * 3^k - 1) * 2))/2 \quad (37)$$

Now we check whether the reduced value of N_c is greater than or equal 4 which is equivalent to checking whether the initial value of N is greater than or equal to $9^k * 4$. We do it otherwise to eliminate a possible case of 'out of bounds' when $N < 9^k * 4$ and $9^k * 4$ is beyond the range of variable used for implementing the algorithm in a programming language.

If it is greater than or equal to 4 then we know for sure that $X >= 2 * 3^k$. If it is not then we know that $3^k <= X < 2 * 3^k$. If the first cases is true then we compute for $N_c = N - 4 * 9^k$ and the arithmetic sum is computed starting from $1 + 2 * 2 * 3^k$ that is starting from the next odd integer after the $2 * 3^k th$ odd integer. If the second case is true then we compute for $N_c = N - 9^k$ and the arithmetic sum is computed starting from $1 + 2 * 3^k$ that is starting from the next odd integer after the $3^k th$ odd integer. For either case we check for one smaller power 3 terms or $3^k - 1$ so that we can get more precise lower limit of the square root.

This process is repeated until the 3^k reduces to 0 after successive iterations and division of it by 3 in each iteration. When it reduces to 0 the exact integer part of the square root of the given integer N is obtained. There is a possibility that as we iteratively subtract the arithmetic sums from N_c it may become 0 which is the case when N is a perfect square.

5 METHODOLOGY

The `nineTree` algorithm follows a two-phase approach:

(1) **Unit Size Determination:** Iteratively divides N by 9 to find the largest base-9 unit size.
(2) **Refinement Step:** Successively refines the square root estimate by decrementing unit size in powers of 3.

The `nineTree` algorithm is designed to efficiently compute the integer square root of a given number N using a base-9 decomposition approach. It consists of two main phases:

5.1 Phase 1: Determining the Initial Unit Size

(1) Initialize a variable `unitSize` to 1 and a temporary variable `temp` to N.
(2) While `temp` is greater than or equal to 9:
   - Multiply `unitSize` by 3.
   - Divide `temp` by 9.
(3) At the end of this loop, `unitSize` represents the largest power of 3 that is less than or equal to $\sqrt{N}$.

5.2 Phase 2: Initial Square Root Estimation

(1) If `temp` is greater than or equal to 4:
   - Compute `doubleUnits` as $2 \times$ `unitSize`.
   - Compute `sum2` = `doubleUnits`^{2}.
   - Update `temp` = $N -$ `sum2` and set `squareRoot` to `doubleUnits`.
(2) Otherwise, if `temp` is at least 1:
   - Compute `sum1` = `unitSize`^{2}.
   - Update `temp` = $N -$ `sum1` and set `squareRoot` to `unitSize`.

5.3 Phase 3: Refining the Square Root Estimate

(1) Set `startTerm` to $2 \times$ `squareRoot` + 1.
(2) Reduce `unitSize` by dividing it by 3.
(3) While `unitSize` is greater than 0 and `temp` is positive:
   - Compute:

$$\texttt{doubleUnits} = 2 \times \texttt{unitSize},$$
$$\texttt{sum1} = \texttt{unitSize} \times (\texttt{startTerm} + \texttt{unitSize} - 1),$$
$$\texttt{sum2} = \texttt{doubleUnits} \times (\texttt{startTerm} + \texttt{doubleUnits} - 1).$$

   - If `temp` is at least `sum2`, update:

$$\texttt{squareRoot} = \texttt{squareRoot} + \texttt{doubleUnits},$$
$$\texttt{temp} = \texttt{temp} - \texttt{sum2}.$$

   - Otherwise, if `temp` is at least `sum1`, update:

$$\texttt{squareRoot} = \texttt{squareRoot} + \texttt{unitSize},$$
$$\texttt{temp} = \texttt{temp} - \texttt{sum1}.$$

   - Reduce `unitSize` by dividing it by 3.
   - Update `startTerm` = 1 + (`squareRoot` $\times$ 2).

5.4 Algorithm Output

At the end of the process, the variable `squareRoot` contains the integer square root of N. The algorithm ensures that no floating-point computations are required, making it efficient for hardware implementations.

5.5 Algorithm Implementation

The algorithm is implemented as follows:

```
int nineTree(long long N) {
    long long temp = N;
    long long unitSize = 1;
    while(temp >= 9) {
        unitSize = unitSize * 3;
        temp = temp / 9;
    }
    long long doubleUnits, sum1, sum2;
    long long squareRoot = 0;

    if(temp >= 4) {
        doubleUnits = 2 * unitSize;
```

```
        sum2 = doubleUnits * doubleUnits;
        temp = N - sum2;
        squareRoot = doubleUnits;
    }
    else if(temp >= 1) {
        sum1 = unitSize * unitSize;
        temp = N - sum1;
        squareRoot = unitSize;
    }
    long long startTerm;
    startTerm = 2 * squareRoot + 1;
    unitSize = unitSize / 3;
    while(unitSize > 0 && temp > 0) {
        doubleUnits = unitSize * 2;
        sum1 = unitSize * (startTerm + unitSize - 1);
      sum2 = doubleUnits * (startTerm + doubleUnits - 1);
        if(temp >= sum2) {
            squareRoot = squareRoot + doubleUnits;
            temp = temp - sum2;
        }
        else if(temp >= sum1) {
            squareRoot = squareRoot + unitSize;
            temp = temp - sum1;
        }
        unitSize = unitSize / 3;
        startTerm = 1 + (squareRoot) * 2;
    }
    return squareRoot;
}
```

6 COMPLEXITY ANALYSIS

6.1 Time Complexity

The first phase determines the unit size by reducing N iteratively via division by 9. This takes $O(\log_9 N)$ time. The second phase refines the square root estimate, requiring another $O(\log_9 N)$ steps. Thus, the overall complexity is:

$$O(2 \log_9 N) = O(\log_9 N)$$

which is asymptotically faster than Newton-Raphson's $O(\log_2 N)$.

6.2 Space Complexity

The algorithm operates using a constant number of integer variables. It does not utilize recursion or additional data structures. Hence, the space complexity is:

$$O(1)$$

7 EXPERIMENTAL RESULTS

To evaluate performance, we tested `nineTree` against Newton-Raphson and binary search across a range of values up to 10^{18}. Key observations include:

- `nineTree` performed **fewer iterations** than Newton-Raphson for large N.
- Execution time was consistently **lower** in integer arithmetic environments.
- The absence of floating-point operations makes it more **suitable for embedded systems**.

8 CONCLUSION

This paper introduced `nineTree`, a novel base-9 decomposition algorithm for integer square root computation. The method achieves a theoretical speedup over Newton-Raphson and binary search, operating in $O(\log_9 N)$ time. Future work includes exploring hardware implementations and potential optimizations in alternative number bases.

9 REFERENCES

REFERENCES

[1] D. Knuth, *The Art of Computer Programming*, Vol. 2, 3rd Ed., Addison-Wesley, 1998.
[2] R. Brent, "Fast Integer Square Root Computation," *Journal of Computational Mathematics*, vol. 5, pp. 89–101, 2002.
[3] "Methods of Computing Square Roots," *Wikipedia*, https://en.wikipedia.org/wiki/Methods_of_computing_square_roots, Accessed Jan. 2025.