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

1 INTRODUCTION

The computation of the integer square root of a number N is a fundamental problem in computer science. 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. This 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. Then,

$$N^2 = (1 + N - 1)^2 \tag{3}$$
$$N^2 = 1^2 + (N - 1)^2 + 2(N - 1) \tag{4}$$
$$N^2 = 1^2 + 2 * N - 2 + (N - 1)^2 \tag{5}$$
$$N^2 = 2 * N - 1 + (N - 1)^2 \tag{6}$$

The equation 6 proves that every N belonging to the set of all positive integers can be expressed as a sum of the Nth odd integer and the square of the integer $N - 1$. Also for every k belonging to the set of all positive integers including 0 less than N that is $N - k >= 1$ and $k >= 0$.

$$(N - k)^2 = (1 + ((N - k) - 1))^2 \tag{7}$$
$$(N - k)^2 = 1^2 + 2 * ((N - k) - 1) + ((N - k) - 1))^2 \tag{8}$$
$$(N - k)^2 = 1 + 2 * (N - k) - 2 + ((N - k) - 1))^2 \tag{9}$$
$$(N - k)^2 = 2 * (N - k) - 1 + ((N - k) - 1))^2 \tag{10}$$

Equation 10 can be recursively solved for all values of k starting from 0 to $N - 1$ as (for the sake of illustration only N is assumed to be a large enough integer such that $N - k >= 1$ 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,

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

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

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

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

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

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

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

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

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 $sum1$ 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 $sum2$ 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 $unitSize = unitSize/3$

4.6 step 6

Reassign $sum1$ again as,

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

and also $sum2$ as,

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

4.7 step 7

Now following cases are possible for each iteration,

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.

5 METHODOLOGY

5.1 Concept

The algorithm works in two phases:

5.1.1 phase 1. In this phase the upper limit of the square root of the given integer N is determined by computing the value of k for which $9^k <= N < 9^{k+1}$ and from results 1 and 2 $3^k <= \sqrt{N} < 3^k$.

5.1.2 phase 2. In this phase the lower limit of the square root of the given integer N is computed iteratively until it can no longer be increased by a positive integer.
We compute the lower limit by computing the arithmetic sum of the first 3^x terms and the first $2*3^x$ where x is an integer. We begin with $x = k$ and repeat the process until $3^k = 0$.
To avoid an overflow case that might occur when $N <= 4 * 9^k$ we do the first estimation of the lower limit outside the iterating block. The comparison is based on the fact that for $9^k <= N < 9^{k+1}$ the value of the *temp* variable after iterative divisions is $1 <= temp < 9$ also,

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

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

where,

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

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

$$\tag{36}$$

and,

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

In each iteration, the respective value of *temp* is compared with the arithmetic sums of 3^x and $2 * 3^x$ terms for respective x of the iteration. The largest sum which is smaller than or equal to *temp* is subtracted from *temp* and the the *squareRoot* variable is incremented by the number of terms in the arithmetic sum which is subtracted from *temp*. Then computing is done for the new *temp* and for $x = x - 1$ starting the sum from the immediately next odd integer given by $1 + 2 * squareRoot$ for respective value of *squareRoot* variable for each iteration. When $3^x = 0$ we have the integer square root of N stored in the variable *squareRoot*.

5.2 Algorithm Output

The algorithm computes the integer value of the square root of the given integer N for N belonging to the set of all of positive integers.

5.3 Algorithm Implementation

The C++ implementation of the algorithm is given below.

```
int nineTree(long long N) {
//this function takes long long N and
// returns the integer value of its square root.

//Estimate the upper limit of the squaer root.
    long long temp = N;
    long long unitSize = 1;
    while(temp >= 9) {
        unitSize = unitSize * 3;
        temp = temp / 9;
    }
//Estimate the lower limit.
    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;
    }
//Refine the lower limit.
    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 the integer value of the square root of N.
    return squareRoot;
}
```

5.3.1 github repository url https://github.com/prem1620/NINETREE.

5.4 Computation of decimal square roots

This algorithm can be modified to give square roots precise to desired decimal places by combining with Newton-Rapshon. The C++ code for the modified algorithm is given below.

```
long double nineTree(long long N, int P) {
// this function takes a long long N and
// int P and prints the square root of N
// precise to P decimal places.

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;
}
//computing for decimal digits
double precision = 1.0;
for (int i = 0; i < P; ++i) precision /= 10;
long double guess = squareRoot;
while (true) {
    double newGuess = 0.5 * (guess + N / guess);
    if (abs(newGuess - guess) < precision) break;
    guess = newGuess;
}
return guess;
}
```

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 across a range of values up to 10^8. Key observations include:

- `nineTree` performed **fewer iterations** than Newton-Raphson for large N.
- Execution time was consistently **lower** than Newton-Rapshon method.
- `nineTree` computed integer square roots while Newton-Raphson produced decimal roots.

7.1 Newton-Raphson algorithm

The C++ code for the algorithm used for comparison is given below.

```
double sqrt_newton_raphson(long long N, int P) {
// this function takes long long N as the input integer
// and int P as the precision and
// returns the square root of N precise to
// P decimal places.
// Initial guess
double x = N / 2.0;

// Precision for comparison
double precision = pow(10, -P);

// Iterate until the difference
// is within the desired precision
while (fabs(x * x - N) > precision) {
    x = 0.5 * (x + N / x);
}
return x;
}
```

Newton-Raphson algorithm was used for benchmarking the `nineTree` algorithm as it is widely used for square root computation and libraries like `SciPy`, `Numpy` and `GNU scientific library` use it for its reliability and precision. Newton-Rapshon is also one of the fastest algorithm as far as the software is concerned. A custom

Table 1: Comparison of Cumulative Runtimes for Square Root Calculation

Range	nineTree	Newton-Raphson
1 - 10,000	0.000 354 s	0.000 968 s
Multiples of 100,000	0.000 001 s	0.000 002 s
Multiples of 1,000,000	0.000 001 s	0.000 002 s
Multiples of 10,000,000	0.000 001 s	0.000 002 s
Multiples of 100,000,000	0.000 001 s	0.000 002 s

Measurements were conducted on an 11th Gen Intel(R) Core(TM) i5-11320H @ 3.20GHz processor with 16GB RAM running Windows 11. Each integer in the specified range was tested for 10,000 trials, and the average runtime was recorded. The Newton-Raphson method used is described in the text and implemented using double precision (64 bits).

implementation of the algorithm was used for the comparison so that innate complexities of the two algorithms could be compared.

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 operating in $O(\log_9 N)$ time. Future work includes exploring hardware implementations.

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.