DAA Assignment 2

Pratham Prakash Gupta (IHM2017003), Aditya Gupta (IHM2017005), Rohit Ranjan (IHM2017001), Priyal Gupta (IHM2017004)

February 2, 2018

1 Abstract

The problem is to first take out the integer part from a floating point number, and then to check whether the integer part is a Pythagorean hypotenuse. There are many algorithms to solve the above problem. We have to select the one which has least time complexity. The main part of solving the problem is that, we cannot use the multiplication, division or module operator. For this we use bit-wise operation to solve the problem of multiplication, division and modulo. Then we solve the problem of checking whether the number is Pythagorean Hypotenuse by using the these functions of multiplication, division and modulo.

2 Keywords

Pythagorean Hypotenuse - In a triangle, if there are three sides - a, b and c, and if a^2+b^2=c^2, then c is called the Pythagorean Hypotenuse.

3 Introduction

3.1 Question

For a given floating point number check whether the integer part is a Pythagorean hypotenuse.

3.2 Literature Survey

We have to develop an algorithm to check whether the integer part of a floating point number is a Pythagorean hypotenuse. There can be several ways to find the same. Our goal is to develop the most efficient algorithm with the least time complexity. We can either first create multiple loops to check weather the given number is of the form $a^2 + b^2 = c^2$. This is the brute force solution of the above problem. But this algorithm will take much time.
Another algorithm have a small loop of i from 1 to $\sqrt{N/2}$ and check if N-i^2 is a perfect square.
The third algorithm makes use of the fact that every Pythagorean Hypotenuse has a prime factor of the form 4k+1.

3.3 Idea

We first made functions of Multiplication, Division and Module with the help of Bit-wise operations. Each of these functions had a time complexity of O(log(n)).
In the main function of checking hypotenuse, we keep dividing n by 2 until it becomes odd, and then check if it is of the form 4k+1.

4 Algorithm Design

4.1 Part a: Brute Force - O(n^3)

4.1.1 Parameters :

n : Input number

4.1.2 Variables :

y : To Calculate power
i,j : To run loop
c,d : To store values

4.1.3 Pseudo Code :

Algorithm 1
power(x)

```
y← 0
for $i = 1$ to $x$
    y ← $y + x$
return $y$
```

Algorithm 2
checkpy(n)

```
n← $math.floor(n)$
for $i = 1$ to $n$
    for j = 1 to n
        if i!=j
            c ← $power(i) + power(j)$
            d ← $math.sqrt(c)$
            if (math.floor(d)-d = 0 and d = n)
                return true
return false
```

4.2 Part b: Using Bit-wise Operators

4.2.1 Parameters :

n : Input number

4.2.2 Variables :

i: Counters
a,b,x,y,dividend,divisor,sign,temp,res: To Store different value

4.2.3 Pseudo Code :

Algorithm 3
DIVIDE(dividend,divisor)

```
sign← $((dividend < 0)^(divisor < 0))? - 1 : 1$
$dividend \leftarrow abs(dividend)$
$divisor \leftarrow abs(divisor)$
$quotient \leftarrow 0, temp \leftarrow 0$
for $i \leftarrow 31$ to 0 do
if temp + (divisor ¡¡ i) ¡= dividend
then temp ← $temp + divisor << i$
quotient ← $quotient|1LL << i$
return $Multiply(sign, quotient)$
```

Algorithm 4
MULTIPLY(a,b)

```
res← 0
while $b > 0$ do
    if (b & 1)
        res ← $res + a$
    a ← $a << 1$
    b ← $b >> 1$
return $res$
```

Algorithm 5
MODULO(a,b)

```
x← $divide(a, b)$
$y \leftarrow multiply(b, x)$
return $a - y$
```

Algorithm 6

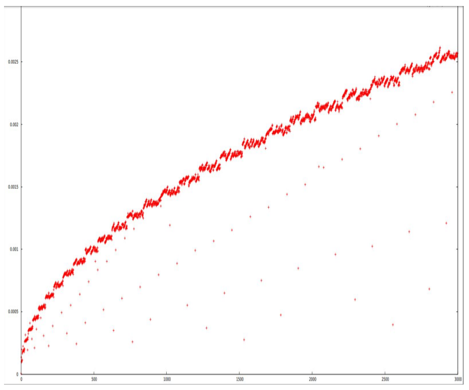
floorSqrt(int x)

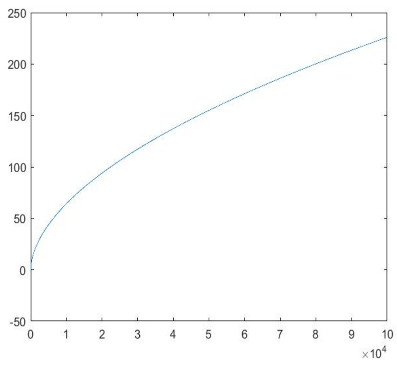
```
if x=0 or x=1
    return x
start ← 1, $end \leftarrow x$
while ($start <= end$) do
    mid ← $(start + end)/2$;
    if (multiply(mid,mid) = x)
        return mid;
    if (multiply(mid,mid) less than x)
        start ← $mid + 1$;
        ans ← $mid$;
    else end ← $mid - 1$
return $ans$
```

Algorithm 7

isHypotenuse(int n)

```
if n = 1 then
    return false
while (modulo(n,2)= 0)
    n ← $divide(n, 2)$
$temp \leftarrow floorSqrt(n)$
for ($i = 3; i <= temp; i \leftarrow i + 2$)
    if (modulo(n,i) = 0)
        if (modulo(i-1,4) = 0)
            return true
        while (modulo(n , i) = 0)
            n ← $divide(n, i)$
if ($n > 2$ and $modulo((n - 1), 4) = 0$)
    return true
else    return false
```





5 Analysis and Discussion

5.1 Time Complexity

The number of instructions executed in pre-computation are of the order of $O(\log_{10} n) + O(\sqrt{n}.(\log_{10}(\log_{10} n)))$ This includes the log(n) complexity for the functions of Multiplication, Division and Modulo. The remaining part of complexity comes during checking whether the number has a prime factor of form 4k+1. The overall time is much less than any other algorithm.

Best Case - It occurs when the number is a power of 2. So in the very first step, it keep dividing the number by 2, and reduces it to 1. The time complexity of best case comes out to be $O(\log_2 n)$

Worst Case - It occurs when the number is a Prime number, as the loop where we are calculating modulo(n,i)==0 will never return true. The time complexity in this case comes out to be $O(\log_2 n) + O(\sqrt{n}.(\log_2(\log_2 n)))$

5.2 Space Complexity

The space complexity of an algorithm is the maximum amount of space used at any one time, thus the space algorithm for this code is O(1).

6 Experimental Setup

Language Used :- C
Plotting tool :- matplotlib and MATLAB
Report Making Tool :- TextPortable(Latex)

7 Conclusion

Though it is easy to check whether a number is Pythagorean Hypotenuse or not, it can be complex to do it without use of Multiplication, Division or Modulo operators.
Thus, in order to solve this, we made use of Bit-wise operator, all of them having a time complexity of O(log(n)).
Then we made use of the fact that every Pythagorean Hypotenuse has a prime factor of the form 4k+1 to solve the problem.

8 References

1. https://www.geeksforgeeks.org/divide-two-integers-without-using-multiplication-division-mod-operator/
2. https://www.geeksforgeeks.org/russian-peasant-multiply-two-numbers-using-bitwise-operators/
3. https://www.geeksforgeeks.org/divide-two-integers-without-using-multiplication-division-mod-operator/
4. https://stackoverflow.com/questions/19332978/algorithm-to-determine-whether-a-given-number-n-can-become-hypotenuse-of-right-t
5. MATLAB