Algorithm to Determine the whether a given Number is an Armstrong Number

Himank Goel iit2016061@iiita.ac.in

Siddhant Gautam iit2016069@iiita.ac.in

Akshat iit2016094@iiita.ac.in

Ankit Shrey iit2016050@iiita.ac.in

***Abstract*—In this paper we focus on determining whether a given input number is an Armstrong Number or not. To determine this we reviewed over the definition of an Armstrong Number. Further we have analyzed the algorithm by plotting graphs of time vs the length of the number.**

I. Introduction

This document is our final report for the Assignment 2 in the course of Design and Analysis of Algorithm. The report describes an algorithm check whether a given input number is an Armstrong Number or not.

We have implemented the above said algorithm in C and we have used GNU Plot to plot the time complexity analysis graphs and other relevant analysis related to our algorithm. For implementing the algorithm we have worked on the definition of Armstrong Numbers and have devised a way to calculate the power a number in a more efficient way.

II. Algorithm Design

The Armstrong number is defined as an n-digit number that is equal to the sum of the nth powers of its digits.

For instance, $1^4 + 6^4 + 3^4 + 4^4 = 1634$

Therefore, to implement the above definition what the algorithm does is to first calculate the order of a number i.e. the number of digits in the given number.

Next, we calculate the nth power of each digit in the number and sum them.

If this value 'sum' comes out to be equal to the original number we find that the number given was an Armstrong Number, else not.

Algorithm 1 To check whether a given number is an Armstrong Number

Input: number

Output: Returns True if 'number' is an Armstrong number.

1: *Initialization* :
2: $sum \leftarrow 0$
3: $size \leftarrow 0$
4: $temp \leftarrow$ number
5: **while** ($temp \geq 0$) **do**
6: $temp \leftarrow temp/10$
7: $size \leftarrow size + 1$
8: **end while**
9: **while** ($number \geq 0$) **do**
10: $sum \leftarrow sum + power$(number%10, size)
11: $number \leftarrow number/10$
12: **end while**
13: **if** ($number = sum$) **then**
14: $return\ true$
15: **else**
16: $return\ false$
17: **end if**

One major computation the above stated algorithm requires is the calculation of a^b quite frequently. The $pow()$ function in C returns a value of Double which when stored in an integer gets truncated and gives faulty answers.

To solve this problem we have used the concept of Binary Exponentiation.

Binary exponentiation works on the concept that given to calculate 3^{11}, 11 can be broken down into binary as 1011_2.

Therefore, $3^{11} = 3^{1011_2} = 3^8 \bullet 3^2 \bullet 3^1$

So we only need to find the sequence of numbers in which each one is a square of the previous one:

$3^1 = 3$
$3^2 = (3^1)^2 = 3 \bullet 3 = 9$
$3^4 = (3^2)^2 = 9 \bullet 9 = 81$
$3^8 = (3^4)^2 = 81 \bullet 81 = 6561$

To get the answer, we multiply three of them (skipping 3^4 because corresponding bit in 1011_2 is zero): $3^{11} = 6561 \bullet 9 \bullet 3 = 177147$.

Algorithm 2 Binary Exponentiation

Input: a, b

Output: a^b

1: *Initialization* :
2: $answer \leftarrow 1$
3: **while** ($b \geq 0$) **do**

```
    if ( b &1 ) then
        answer ← answer * a
    end if
    a ← a * a
    b ← b/2
end while
```

III. Analysis and Discussion

The above presented algorithm takes the same amount of time for a particular n, (here n being the number of digits in the given number). This is because the loop initially iterates over the number to calculate the number of digits and then calculates the individual number's power to n.
At no point in this algorithm does the value of given number affects the algorithm, therefore for a particular n the algorithm takes the same amount of time always.
Therefore, there is no best or worst case or we can say that in either of the cases we get same tight bounds, therefore the presented alogrithm is a θ algorithm.

$$f(n) = 3 + 4n + 1 + n * (2 + 6(1 + log(n))) + 4n + 1$$
$$f(n) = 6n * log(n) + 16n + 5$$
$$5n * log(n) < 6n * log(n) + 16n + 5 < 27n * log(n)$$
$$g(n) < f(n) < h(n)$$

Here $g(n)$ is the tightest lower bound and $h(n)$ is the lowest upper bound. Therefore, time complexity is $\theta(nlog(n))$

IV. Experimental Study

For testing the above algorithm we plotted the time vs n (where, n is the number of digits) for 180 cases where number of digits in given input number vary from 1 to 18. For each value of n we chose 10 possible numbers of n digits. The numbers randomly generated using the rand() function in C.

After this, we divided the inputs according to number of digits in the generated number and analyzed the T_{min}, T_{max}, and T_{avg} for the numbers of that particular digit. The tables and corresponding graphs generated are shown below.

Profiling of Algorithm

n	T_{max}	T_{min}	T_{avg}
1	22	22	22
2	49	49	49
3	77	77	77
4	113	113	113
5	150	150	150
6	179	179	179
7	222	222	222
8	261	261	261
9	311	311	311
10	345	345	345
11	401	401	401
12	413	413	413
13	473	473	473
14	509	509	509
15	575	575	575
16	597	597	597
17	668	668	668
18	707	707	707

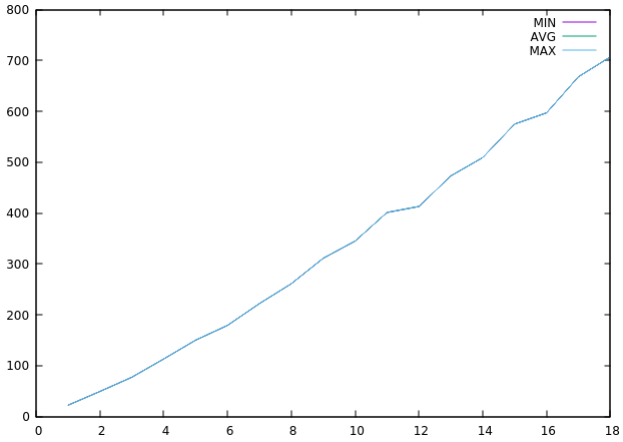

Fig. 1. Armstrong number algorithm Analysis

It is quite evident from the table and the correspondong graph that for a certain value of n (i.e. the number of digits) the algorithm takes equal amount of time irrespective of the actual value of the input.

V. Conclusion

In this paper, we devised an algorithm to to check whether a given number is an Armstrong Number or not in a time complexity of $\theta(nlog(n))$.
We also used a method of Binary Exponentiation to calculate power of a number. This improved time of the respective function from O(n) to O(log(n)).
Our experiments and analysis indicated that the algorithm is a θ algorithm because the worst and best case complexities are exactly the same.

REFERENCES

[1] *Algorithm, Binary Exponentiation* (n.d.). Retrieved from tps://e-maxx-eng.appspot.com/algebra/binary-exp.html