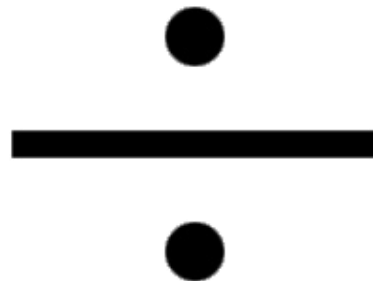


# **Analysis and Implementation of Common Division Algorithms**



# Problem statement

Let  $n$  be the dividend and  $d$  the divisor both unsigned real integers.

How can we find the quotient  $q$  and the remainder  $r$ ?

Can this procedure be optimized?

# Why Unsigned Real Integers

## **Unsigned = positive**

Negative numbers are a special case of positive numbers.

For negative numbers just put the minus sign in front of the quotient.

## **Real = no complex part**

Complex numbers are an algebraic combination and manipulation of real numbers.

Most real-life applications and tasks use real numbers.



## **Integers = no fractional part**

Fast algorithms do not bother with the fractional part as this makes them slower.

# Applications and Importance of Division Algorithms

- Complexity of the floating-point unit (FPU) – Standalone or Integrated
- High performance 3D graphics rendering systems (CAD)
- Image processing
- QR decomposition

# DIVISION NEEDED IN FPU AND CPU

Date	System	Picture
1980	Intel 8087	 <p>FPU</p>
1984	Motorola 68881, 68882	 <p>FPU</p>

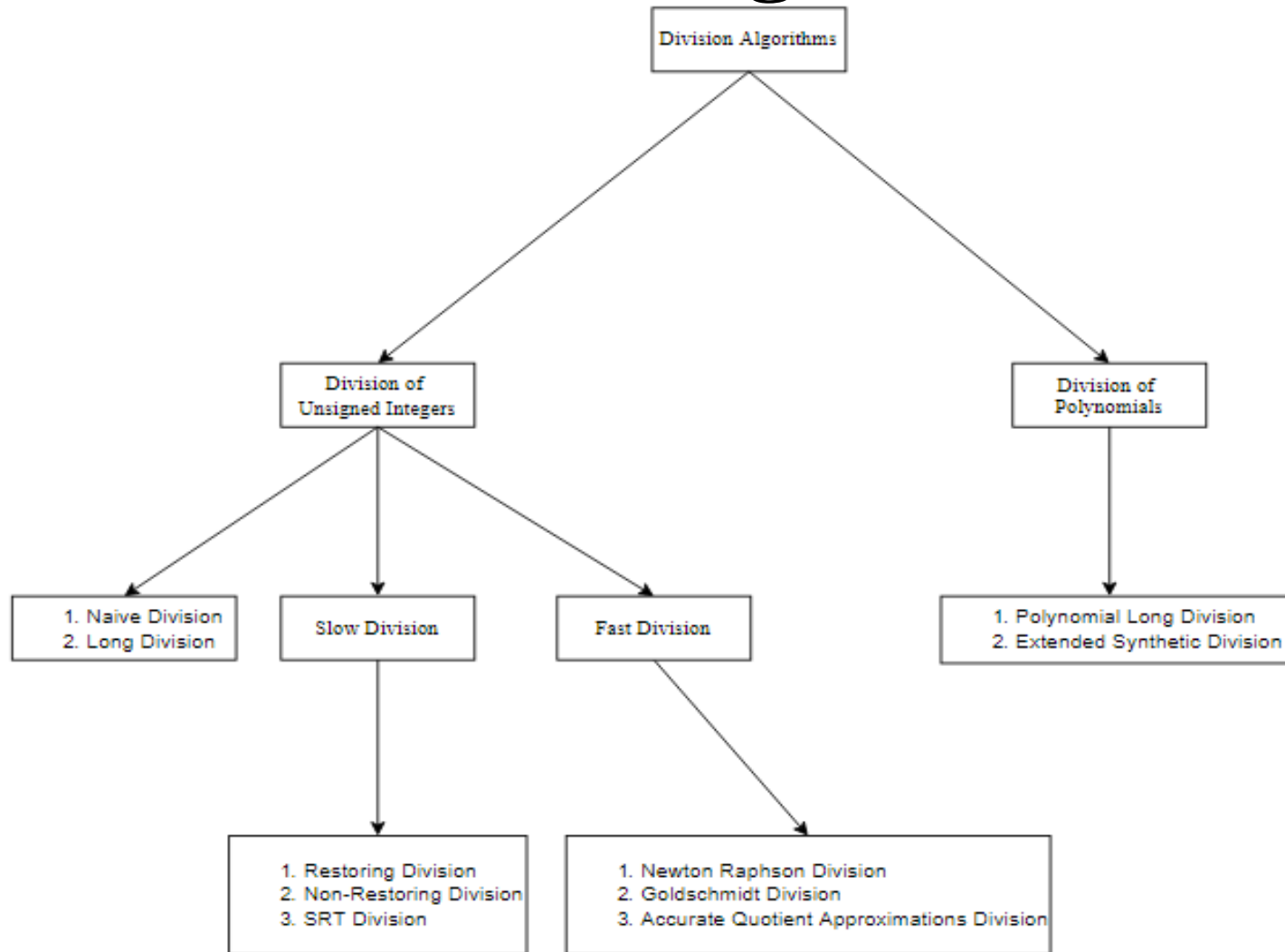
CPU



# The Need to Optimize Division

Microprocessor	1 Cycle time (ns)	Machine Cycles Required for $a \pm b$	Machine Cycles Required for $a \times b$	Machine Cycles Required for $a \div b$
Alpha 21164	2.0	4	4	22-60
HP PA 8000	5.0	3	3	31
Pentium 1	5.0	3	5	17
MIPS R10000	3.64	2	2	18
PowerPC 604	5.56	3	3	31
UltraSPARC	4.0	3	3	22
Pentium 4	0.67	5	7	23
AMD-K7	0.83	4	4	16

# Common Division Algorithms



# Euclid's Division Lemma

$$n = q \times d + r$$

$$\frac{n}{d} = q + \frac{r}{d}$$

n = dividend

q = quotient

d = divisor

r = remainder



# Conditions on Remainder

$$\begin{aligned} r &< d \times \text{ulp} \\ 0 &\leq r < d \end{aligned}$$

# Slow Division/ Digit recurrence Algorithms

- Digit recurrence algorithms reserve a fixed number of bits for the quotient in every iteration.
- They produce a single quotient bit/digit in each iteration by adding or subtracting a multiple of divisor from the dividend.
- They are based on the following standard recurrence equation.

$$R_{j+1} = B \times R_j - q_{n-(j+1)} \times D$$

$R_j$  =  $j^{\text{th}}$  partial remainder of the division

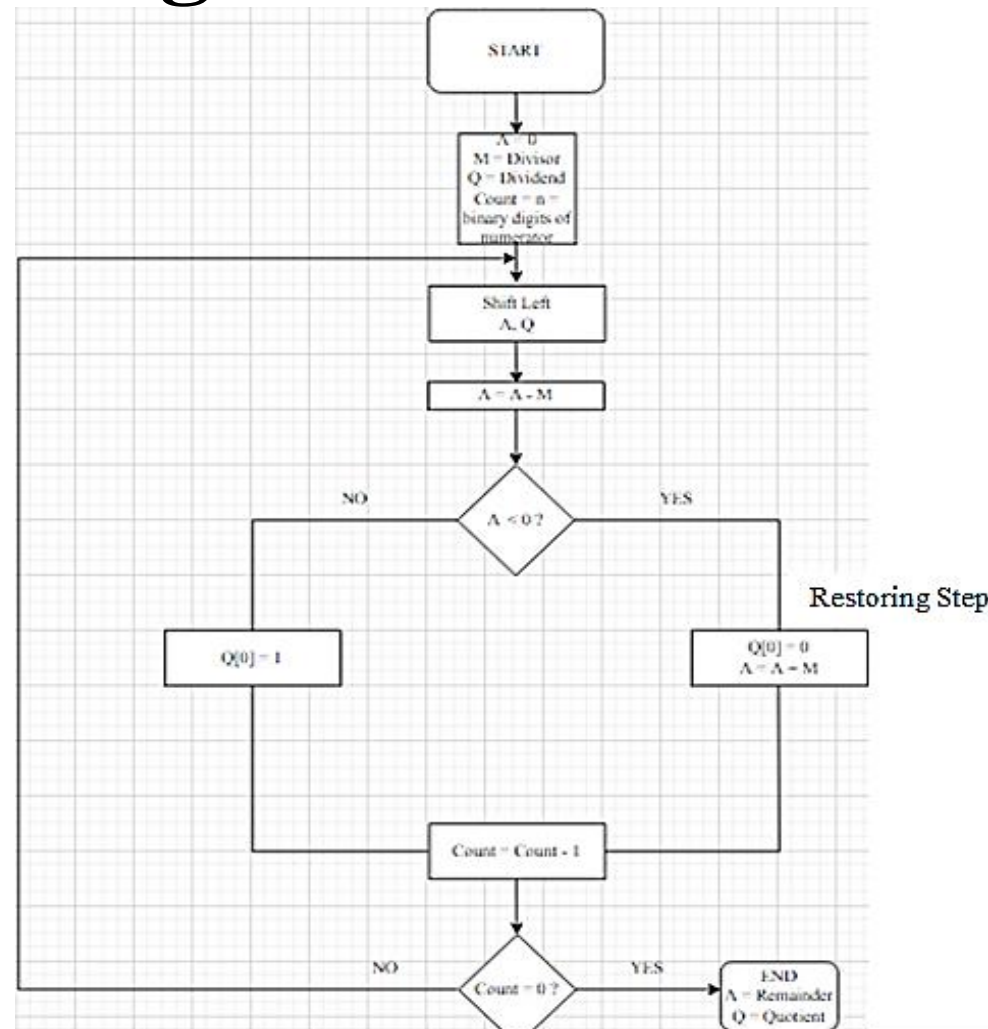
$B$  = base (2 for binary)

$q_{n-(j+1)}$  = digit of the quotient in position  $n-(j+1)$

$D$  = divisor

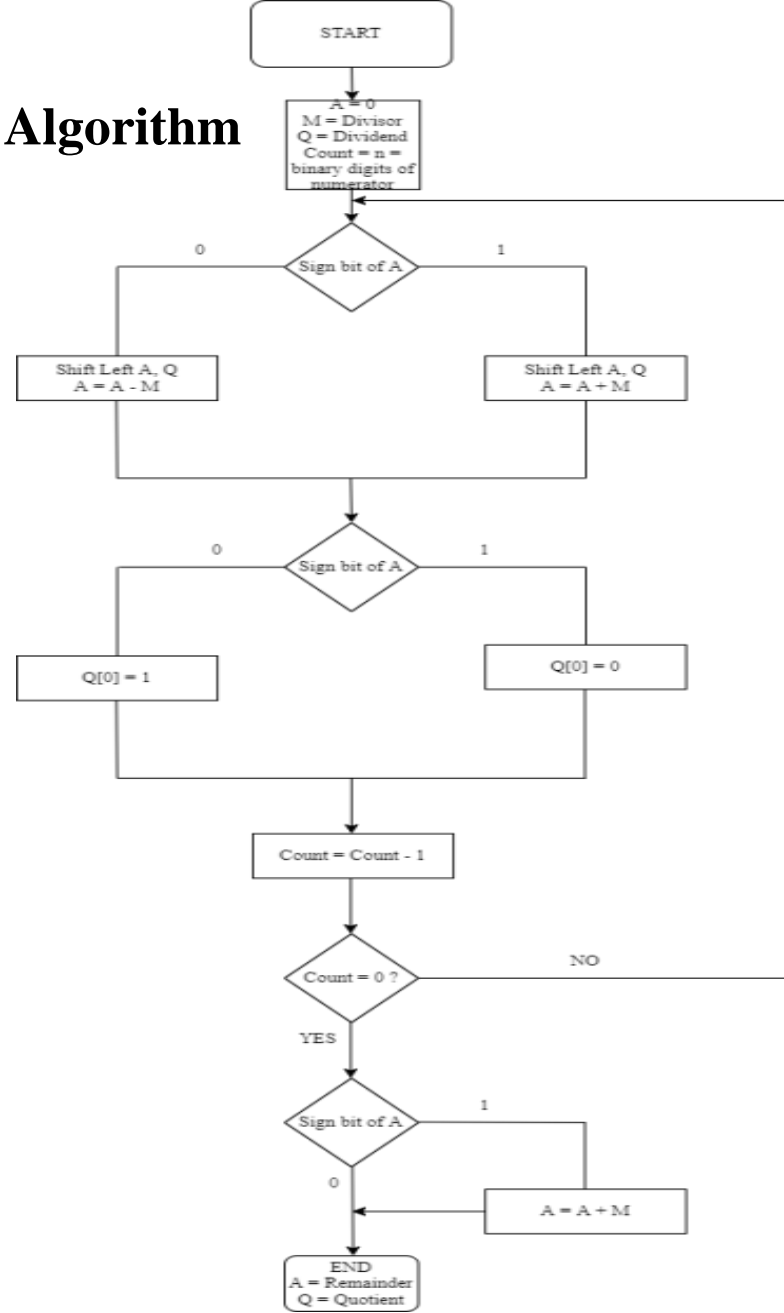
# Restoring Division Algorithm

- Restoring division algorithm got its name from the fact that it restores the value of A in each iteration.
- This algorithm works on the binary level using only 0 and 1.
- The digits of the quotient are from the set [0, 1].



# Non-Restoring Division Algorithm

- More complex than Restoring Division Algorithm.
- However, it avoids the Restoring Step.



# SRT Division Algorithm

- Application: General-purpose CPU
- The Intel Pentium processor (slide 5) uses a base 4 SRT division algorithm where the quotient digits are from the set  $[-2, -1, 0, 1, 2]$ .
- SRT is a special case of the non-restoring division algorithm.
- Generates a fixed number of quotient bits at every iteration.
- Uses subtraction as the fundamental operator to calculate a quotient bit/digit in each iteration.
- **Main Idea:** Guess the quotient digit based on the most significant bits of the divisor and the partial remainder.
- Next, find an answer close enough to the correct answer and then use redundant quotient bits/digits to correct subsequent steps.

# SRT Division Algorithm

**Step 1:** The partial remainder is multiplied by the base  $n$  that is used. This is done by left bit shifting by  $k$  bits.

**Step 2:** The result is given to the quotient selection table and the subtractor.

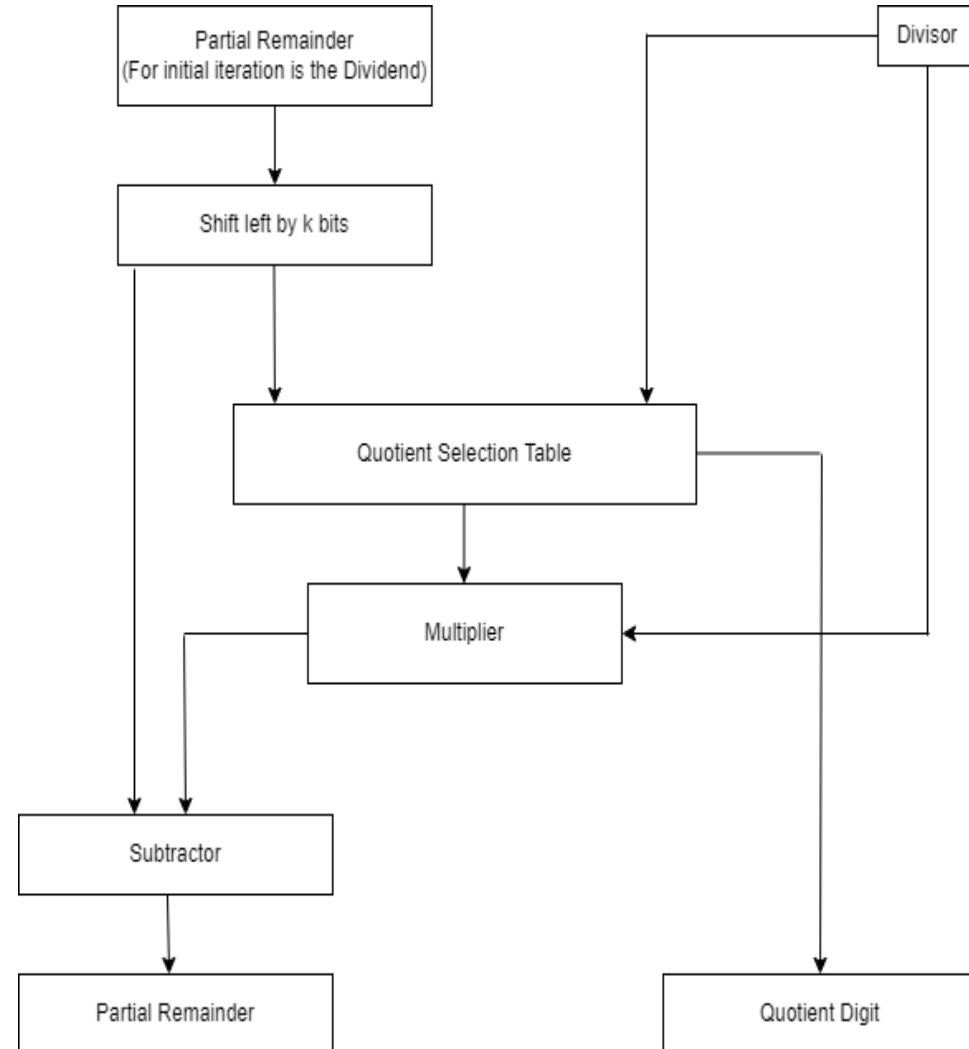
**Step 3:** The divisor provides the second input for the quotient selection table.

**Step 4:** Based on few of the most significant bits of the divisor and the partial remainder, the quotient digit for the next iteration is calculated. This is known as the multiplier.

**Step 5:** The multiplier becomes the second input of the subtractor.

**Step 6:** The subtractor generates the next partial remainder.

**Step 7:** Repeat process until all the quotient bits/digits are revealed



# Fast Division/ Functional Iterative/ Multiplicative Algorithms

- Give more than one quotient digit in each iteration.
- They are based on multiplication. Eliminate addition and subtraction as much as possible.
- Multiplication can be done using bit shifting.
- Bit shifting is faster than addition and subtraction.
- They give only the quotient. Do not bother with the remainder.

# Newton Raphson Division Algorithm

- Applications: 1. IBM RS/600  
2. Intel i860  
3. Astronautics ZS-1
- **Fact:** The quotient is a product of the dividend and the reciprocal of the divisor

$$Q = \text{dividend} \times \frac{1}{\text{divisor}} = D_d \times \frac{1}{D_r} = \frac{D_d}{D_r}$$

- The reciprocal of the divisor will converge after several iterations.

$D_d$  = dividend

$D_r$  = divisor



# Newton Raphson Division Algorithm

The Newton Raphson Division Algorithm is divided into three processes.

## Process 1:

- Find the initial estimation of the divisor's reciprocal using a priming function that has a root at the divisor's reciprocal.

$$F(X) = \frac{1}{X} - D_r = 0, F(X) = \text{Priming function}$$

# Newton Raphson Division Algorithm...

## Process 2:

- Focus on the iterative refinement of the divisor's reciprocal
- Apply the Newton Raphson Equation on the Priming function.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

- Evaluate at  $X_0$  to find successive approximations to the divisor's reciprocal. Error term of order  $O(n^2)$

$$f(X_0) = \frac{1}{X_0} - b$$

$$f'(X_0) = \frac{-1}{X_0^2}$$

# Newton Raphson Division Algorithm...

- Successive approximations to the divisor's reciprocal until it converges.

$$X_1 = X_0 - \frac{f(X_0)}{f'(X_0)}$$

$$X_1 = X_0 + \frac{\left(\frac{1}{X_0} - b\right)}{\left(\frac{1}{X_0^2}\right)}$$

$$X_1 = X_0 \times (2 - b \times X_0)$$

$$X_{i+1} = X_i \times (2 - b \times X_i)$$

# Newton Raphson Division Algorithm...

## Process 3:

- Multiply the dividend and the final convergent divisor reciprocal found in the previous step using the fact mentioned before.

$$Q = \text{dividend} \times \frac{1}{\text{divisor}} = D_d \times \frac{1}{D_r} = \frac{D_d}{D_r}$$

# Goldschmidt Division Algorithm

Applications: 1. General-purpose CPU

2. IBM 360/91

3. TMS390C602A Sparc FPU

**Main Idea:** The iterative parallel multiplication of the dividend and the divisor by multiplicative factors until the final divisor will be one or close to one

# Goldschmidt Division Algorithm...

## Process 1:

- Want to find  $q$  such that  $q = \frac{a}{b}$ .
- Generate a multiplication factor using the Taylor-Maclaurin series of  $\frac{1}{1+x}$ .
- Required to find the expansion of  $q = \frac{a}{b} = a \times f(x)$ .
- Let  $f(x) = \frac{1}{1+x}$  and expand at  $x = 0$  (Maclaurin)
- The result is:  $\frac{1}{1+x} = 1 - x + x^2 - x^3 + x^4 - \dots$
- Let  $f(x) = \frac{1}{b}$ .
- Let  $x = b - 1$  in the Maclaurin series expansion.
- The value of  $q$  is

$$\begin{aligned} q &= \frac{a}{b} = a \times f(x) = a \times f(b - 1) = a \times \frac{1}{1 + (b - 1)} \\ &= a \times \frac{1}{1 + x} = a \times (1 - x + x^2 - x^3 + x^4 - \dots) \end{aligned}$$

- $a$  is a common factor. Factoring the above equation yields the following equation  
$$q = a \times [(1 - x)(1 + x^2)(1 + x^4)(1 + x^8)(1 + x^{16}) \dots]$$

# Comparison of Common Division Algorithms

We will compare all division algorithms with respect to:

1. The number of digits that they can handle
2. The time they need to execute

The tests were done in the Python programming language with a machine having an AMD 3020e CPU and the following machine environment.

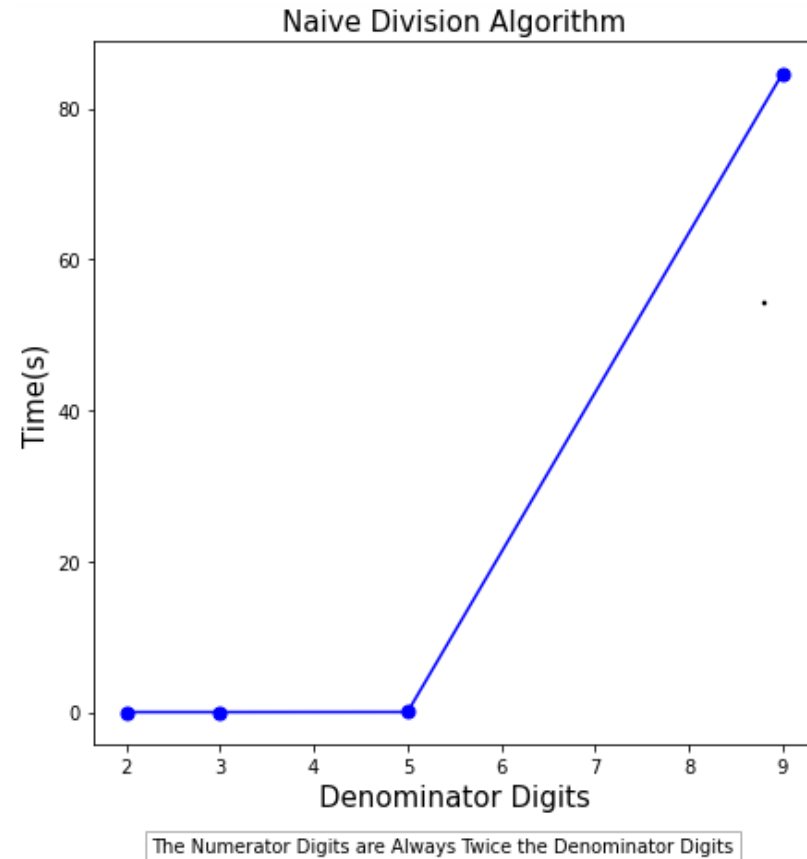
```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

```
machine epsilon= 2.220446049250313e-16
```

The tests were stopped at 30 minutes.

# Naive Division Algorithm

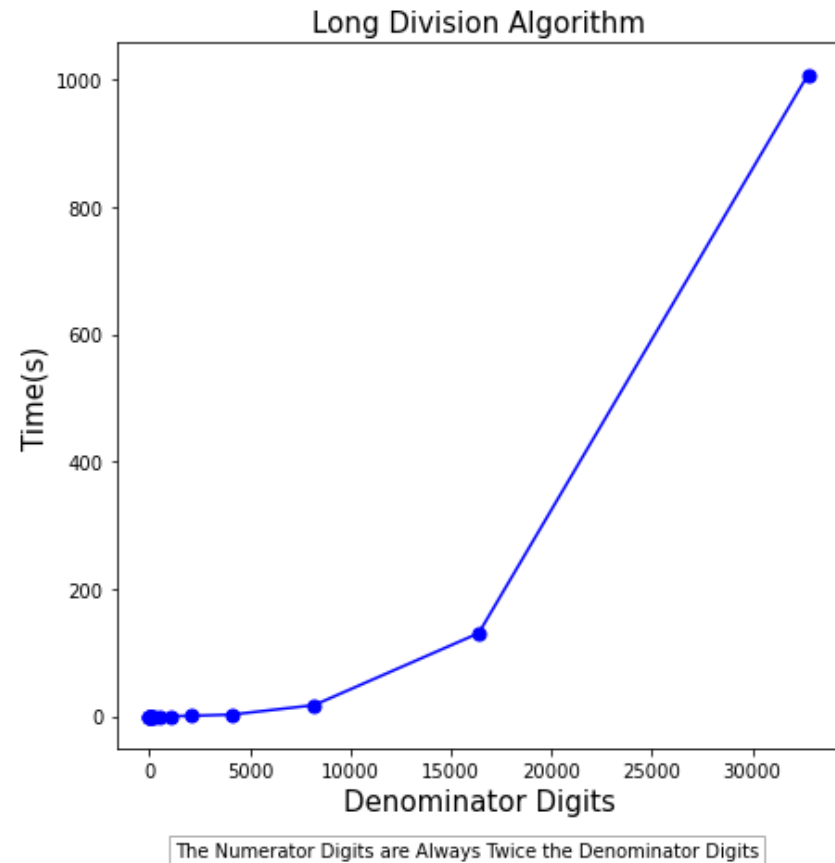
	Numerator Digits	Denominator Digits	Time(s)
1	3	2	0.000000
2	5	3	0.000000
3	9	5	0.010991
4	17	9	84.627516





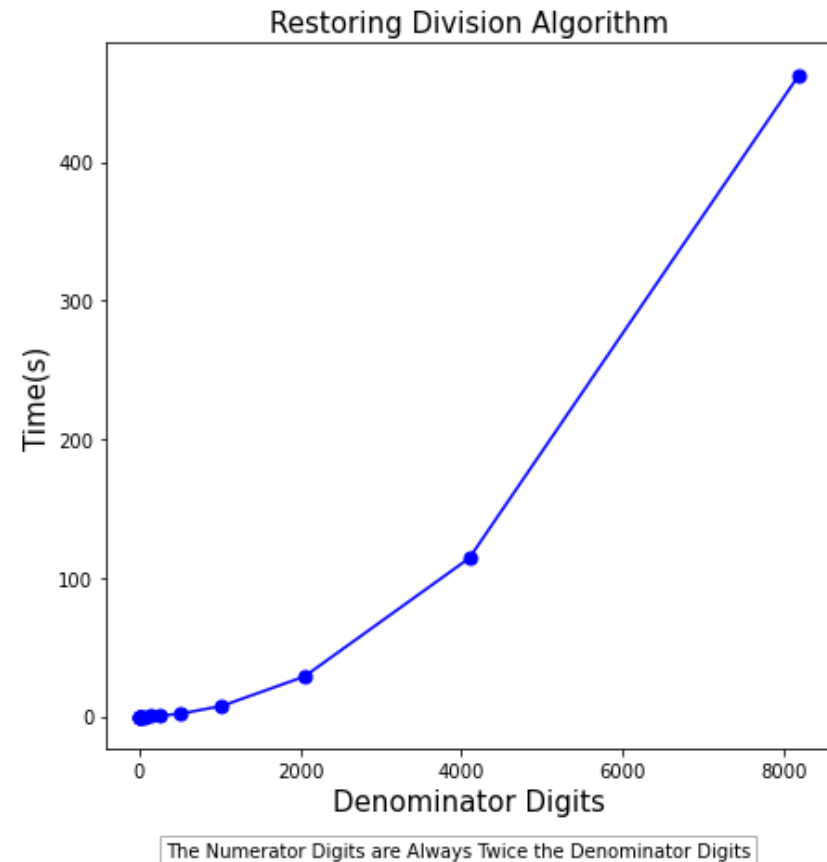
# Long Division Algorithm

	Numerator Digits	Denominator Digits	Time(s)
1	3	2	0.000000
2	5	3	0.000000
3	9	5	0.001000
4	17	9	0.000000
5	33	17	0.000000
6	65	33	0.000999
7	129	65	0.001997
8	257	129	0.003997
9	513	257	0.007992
10	1025	513	0.022979
11	2049	1025	0.084921
12	4097	2049	0.405623
13	8193	4097	2.424751
14	16385	8193	17.200045
15	32769	16385	130.346088
16	65537	32769	1007.628282



# Restoring Division Algorithm

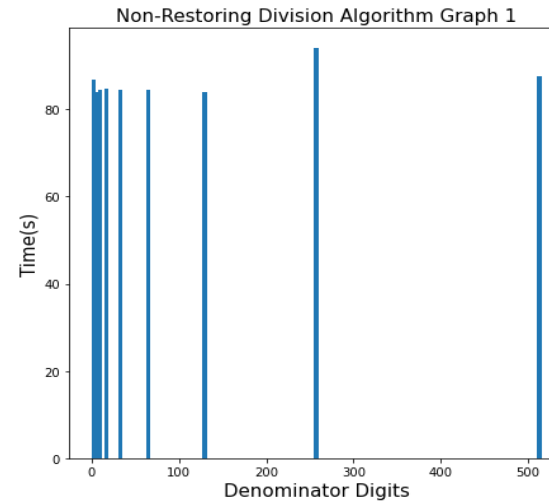
	Numerator Digits	Denominator Digits	Time(s)
1	3	2	0.001000
2	5	3	0.000000
3	9	5	0.000998
4	17	9	0.000999
5	33	17	0.003997
6	65	33	0.011989
7	129	65	0.036967
8	257	129	0.133874
9	513	257	0.483552
10	1025	513	1.853281
11	2049	1025	7.444095
12	4097	2049	28.663396
13	8193	4097	114.108163
14	16385	8193	462.880590



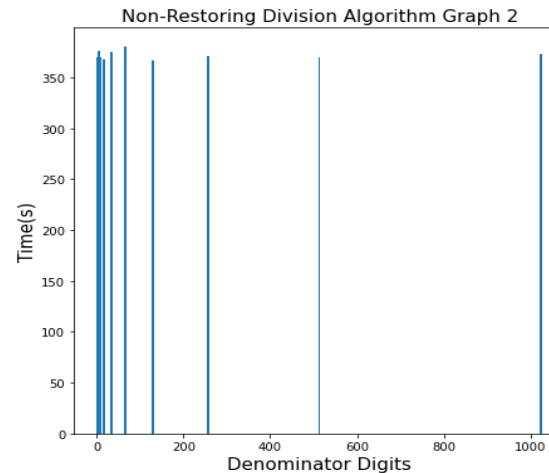
# Non-Restoring Division Algorithm

	Numerator Digits	Denominator Digits	Time(s)
1	3	2	86.672612
2	5	3	83.805259
3	9	5	83.946112
4	17	9	84.373748
5	33	17	84.718411
6	65	33	84.425683
7	129	65	84.488609
8	257	129	83.954136
9	513	257	93.925871
10	1025	513	87.674854

	Numerator Digits	Denominator Digits	Time (s)
1	3	2	370.069701
2	5	3	368.594196
3	9	5	376.286671
4	17	9	370.354443
5	33	17	368.197445
6	65	33	375.336798
7	129	65	380.020501
8	257	129	366.533988
9	513	257	371.276589
10	1025	513	370.073704
11	2049	1025	373.608426



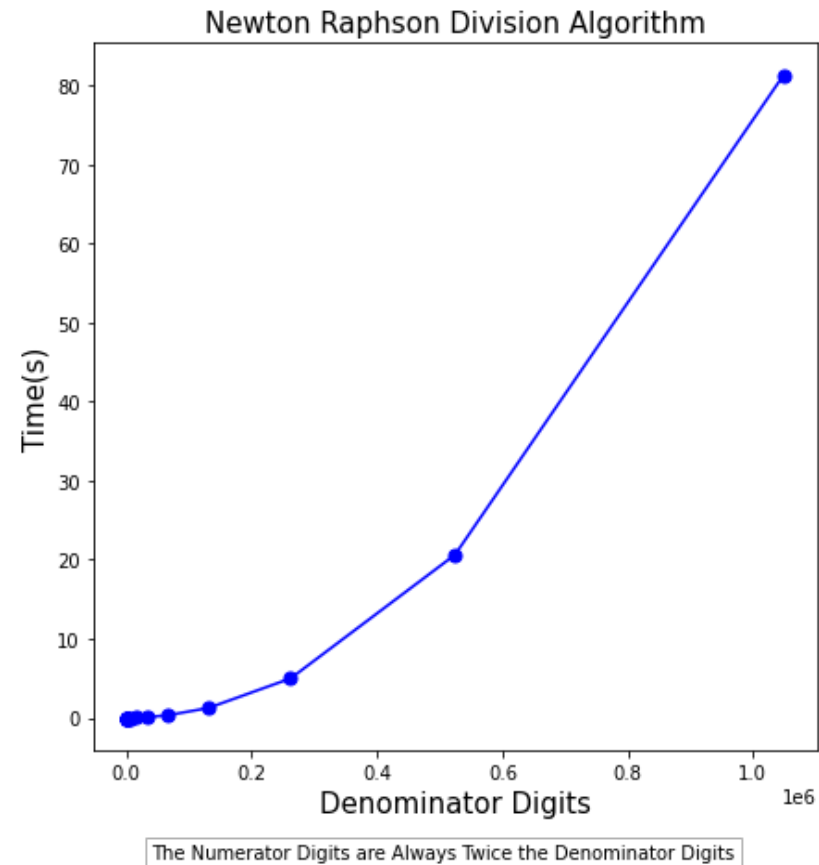
The Numerator Digits are Always Twice the Denominator Digits



The Numerator Digits are Always Twice the Denominator Digits

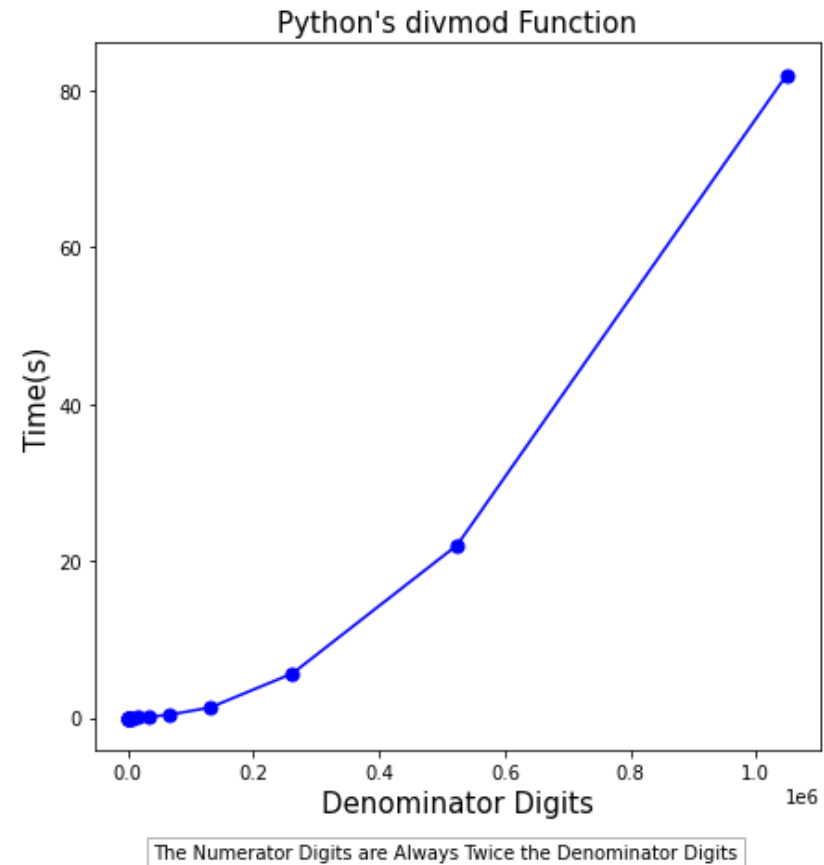
# Newton Raphson Division Algorithm

	Numerator Digits	Denominator Digits	Time(s)
1	3	2	0.000999
2	5	3	0.000000
3	9	5	0.000000
4	17	9	0.000000
5	33	17	0.000000
6	65	33	0.000000
7	129	65	0.000000
8	257	129	0.000000
9	513	257	0.000000
10	1025	513	0.000000
11	2049	1025	0.000000
12	4097	2049	0.000999
13	8193	4097	0.001998
14	16385	8193	0.004995
15	32769	16385	0.019983
16	65537	32769	0.077929
17	131073	65537	0.310712
18	262145	131073	1.243822
19	524289	262145	4.985398
20	1048577	524289	20.624876
21	2097153	1048577	81.322561



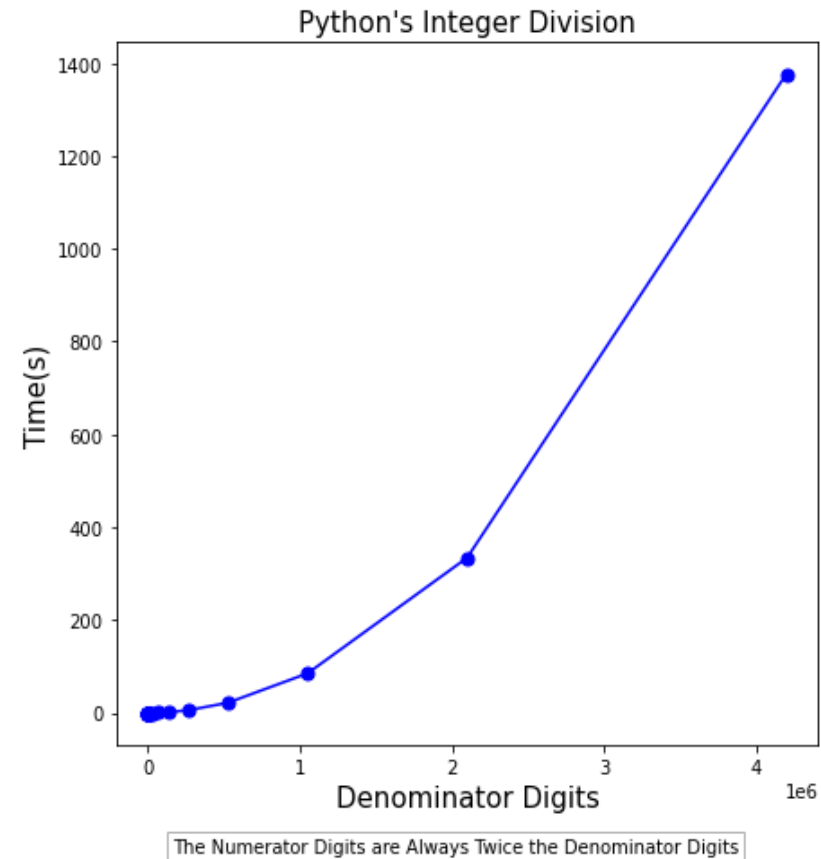
# Python's divmod Function

	Numerator Digits	Denominator Digits	Time(s)
1	3	2	0.000000
2	5	3	0.000000
3	9	5	0.000000
4	17	9	0.000000
5	33	17	0.000000
6	65	33	0.000000
7	129	65	0.000000
8	257	129	0.000000
9	513	257	0.000000
10	1025	513	0.000000
11	2049	1025	0.000000
12	4097	2049	0.000000
13	8193	4097	0.001996
14	16385	8193	0.005996
15	32769	16385	0.024977
16	65537	32769	0.104904
17	131073	65537	0.361669
18	262145	131073	1.296821
19	524289	262145	5.649780
20	1048577	524289	22.032585
21	2097153	1048577	82.013895



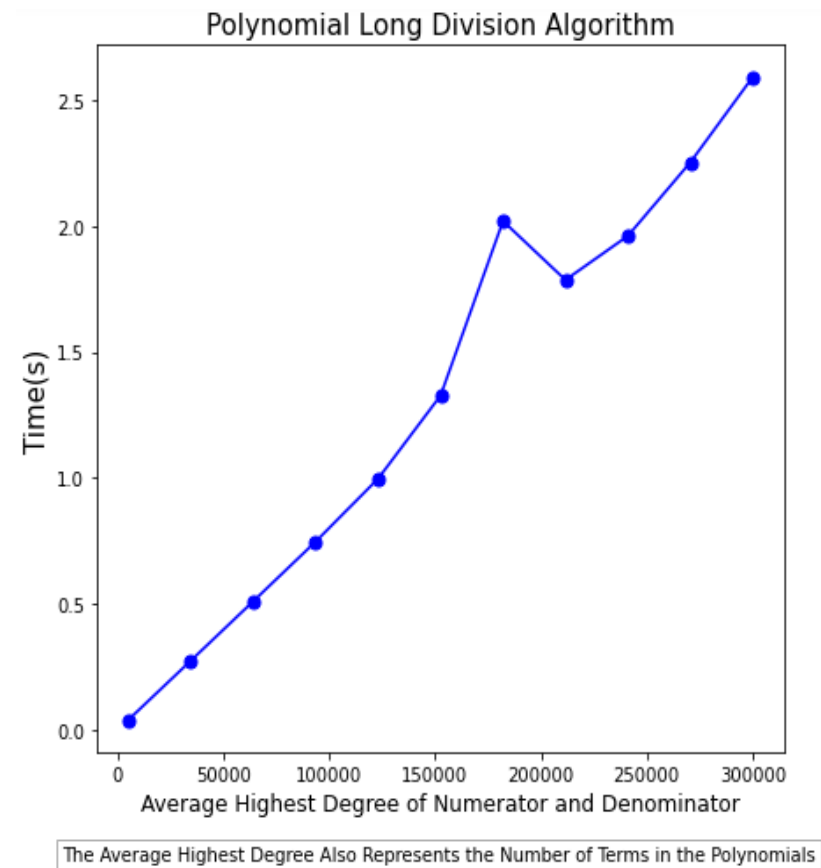
# Python's Integer Division

	Numerator Digits	Denominator Digits	Time(s)
1	3	2	0.000000
2	5	3	0.000000
3	9	5	0.000000
4	17	9	0.000000
5	33	17	0.000000
6	65	33	0.000000
7	129	65	0.000000
8	257	129	0.000000
9	513	257	0.000000
10	1025	513	0.000000
11	2049	1025	0.000000
12	4097	2049	0.000000
13	8193	4097	0.002991
14	16385	8193	0.007985
15	32769	16385	0.025974
16	65537	32769	0.079931
17	131073	65537	0.339664
18	262145	131073	1.276824
19	524289	262145	5.358030
20	1048577	524289	21.516048
21	2097153	1048577	84.886254
22	4194305	2097153	334.145011
23	8388609	4194305	1376.462213



# Polynomial Long Division Algorithm

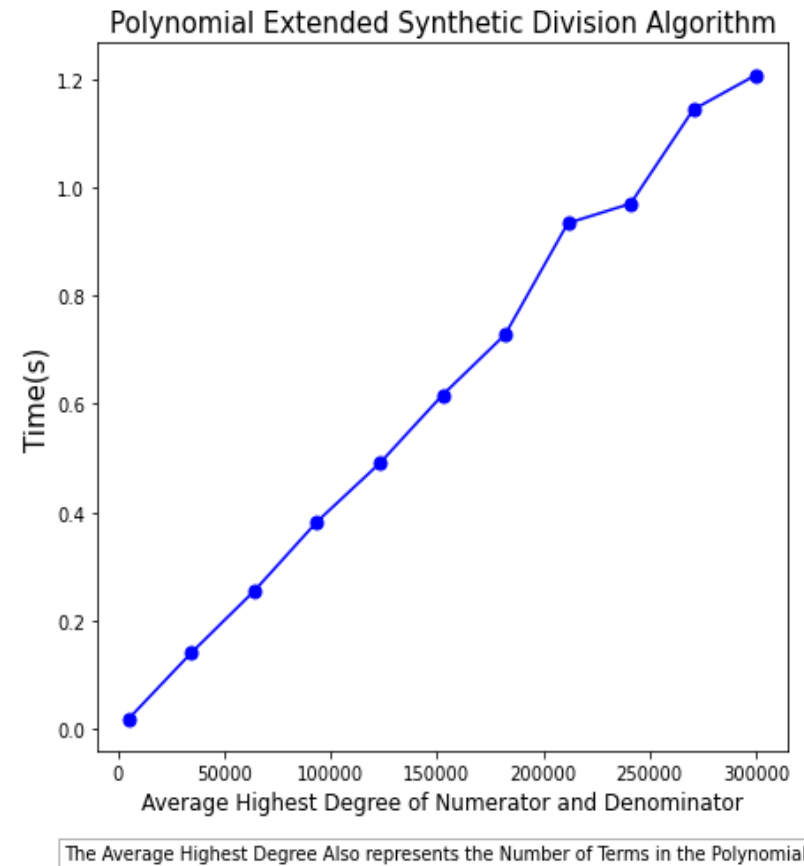
	Numerator Highest Degree	Denominator Highest Degree	Time(s)
1	5002	5000	0.039962
2	34502	34500	0.274746
3	64002	64000	0.510526
4	93502	93500	0.747307
5	123002	123000	0.997053
6	152502	152500	1.326780
7	182002	182000	2.019138
8	211502	211500	1.785330
9	241002	241000	1.960184
10	270502	270500	2.251888
11	300002	300000	2.590598



# Polynomial Extended Synthetic Division Algorithm

Numerator Highest Degree    Denominator Highest Degree    Time(s)

1	5002	5000	0.018982
2	34502	34500	0.140893
3	64002	64000	0.254762
4	93502	93500	0.382647
5	123002	123000	0.490546
6	152502	152500	0.616430
7	182002	182000	0.728325
8	211502	211500	0.934136
9	241002	241000	0.970101
10	270502	270500	1.143923
11	300002	300000	1.207880





# Conclusion

Number of atoms in the universe =  $10^{78}$  to  $10^{82}$

Newton Raphson division algorithm = 2,097,153 digits (able to carry the most complex tasks)

Polynomial extended synthetic division algorithm = 300,002<sup>nd</sup> degree polynomials

Newton Raphson division algorithm is the fastest and faster than Python's built-in functions

None of the division algorithms is considered as the “**best**”.

There is a trade-off in choosing what division algorithm to use and it depends on the task needed to be done.

# QUESTIONS