

THE CITY COLLEGE OF NEW YORK

Revolutionizing the Division Operation:

Analysis and Implementation of Common Division Algorithms

Abstract

Division is the most time-consuming, resource expensive, and complex arithmetic operation. Hardware and software support for floating-point arithmetic is a must in modern central processor units. Even though division is used the least compared to addition and multiplication, it is becoming increasingly important in modern complex applications. The performance of such applications is affected and limited by the algorithms used for division. This report studies some of the most used division algorithms such as long division, restoring, non-restoring, SRT, Newton-Raphson, Goldschmidt, accurate quotient approximations, polynomial long division, and polynomial extended synthetic division. The report continues by comparing the fastest division algorithm with Python's built-in integer division, `divmod` function, and `polydiv` function. Finally, the report concludes that the Newton-Raphson division algorithm is the fastest for integers, and the polynomial extended synthetic division algorithm is the fastest for polynomials. Moreover, the report will illustrate that these two algorithms are faster and more efficient than Python's built-in algorithms.

Georgios Ioannou

CSC 30100

Professor Erik K. Grimmelmann

14 December 2021

Copyright © 2021

by

Georgios Ioannou

TABLE OF CONTENTS

1. Introduction	1
2. General Division Background	5
3. Naive Division Algorithm	6
4. Long Division Algorithm	7
5. Restoring Division Algorithm	9
6. Non-Restoring Division Algorithm	12
7. SRT Division Algorithm	14
8. Newton Raphson Division Algorithm	17
9. Goldschmidt Division Algorithm	19
10. Accurate Quotient Approximations Division Algorithm	21
11. Polynomial Long Division Algorithm	23
12. Polynomial Extended Synthetic Division Algorithm	25
13. Comparing With Python's Divmod Function and Integer Division	28
14. Conclusion	30

LIST OF FIGURES

1. Central processing unit with an integrated floating-point unit developed by Intel	2
2. Division algorithms studied in this report	4
3. Long Division Procedure	8
4. Restoring Division Algorithm Flowchart	11
5. Non-Restoring Division Algorithm Flowchart	13
6. SRT Division Algorithm Flowchart	16
7. Regular Synthetic Division Algorithm Procedure	26
8. Polynomial Extended Synthetic Division Algorithm Procedure	27

LIST OF TABLES

1. Floating-point units developed by Intel and Motorola	2
2. Microprocessor and arithmetic operations comparison	3

LIST OF EQUATIONS

1. How to compute the quotient	6
2. Euclid's Division Lemma	6
3. The final answer of a division	6
4. Condition required for the remainder	6
5. Slow/Digit Recurrence Algorithms Common Standard Equation	11
6. Restoring Division Algorithm Partial Remainder and Quotient	11
7. Non-Restoring Division Algorithm Partial Remainder and Quotient	13
8. SRT Division Algorithm Partial Remainder and Quotient	16
9. Definition of m and Q in the SRT Division Algorithm	16
10. Expressing the Quotient as a Product of the Dividend and the Divisor	18
11. Priming Function	18
12. Newton Raphson Equation	19
13. Apply Newton Raphson Equation on the Priming Function	19
14. Approximation to the Divisor's Reciprocal	19
15. Taylor-Maclaurin Series of the Goldschmidt Division Algorithm	20
16. Expansion of the Taylor-Maclaurin Series of the Goldschmidt Division Algorithm	20
17. Quotient of the Goldschmidt Division Algorithm	21
18. Calculating the New Function to Perform Synthetic Division	27

Revolutionizing the Division Operation:

Analysis and Implementation of Common Division Algorithms

INTRODUCTION

As technology evolves rapidly, computers are required to perform more complex tasks and applications that involve the basic operation of division. The most critical tasks include the complexity of the floating-point unit (FPU), high-performance 3D graphics rendering systems (CAD), image processing, K-means clustering, and QR decomposition. However, all these complex tasks depend and go back to the fundamental principle of computers which is arithmetic. The main goal of computers is to do arithmetic and as time progress, the arithmetic that is required to be done by computers consists of more complex and larger numbers. Computers perform arithmetic by converting every number into binary and then using the three basic arithmetic operations of addition, multiplication, and division. Subtraction is not mentioned because it is a special case of addition and therefore computers only consider addition.

One may argue that division is a special case of multiplication and that computers must also avoid it like subtraction. The truth is that compared to addition and multiplication, the division is used the least and must be avoided or being replaced by multiplications whenever possible. On the other hand, computers will experience performance degradation if the division is not used and there are cases where division is unavoidable. Floating-point computational performance and logic have long been essential components of high-performance systems. The

14 December 2021

performance of systems that have a high frequency of floating-point operations is often limited by the speed of the floating-point hardware which is the FPU. The FPU is responsible for all mathematical operations that have anything to do with floating-point numbers. The FPU is different from the arithmetic logic unit (ALU) which operates on integer binary numbers. The FPU can be a standalone processor or an integrated part inside the central processing unit (CPU). As the FPU has become more advanced year by year, the operations of addition and multiplication have become more efficient while the operation of division did not. Table 1 illustrates some of the FPUs developed throughout time. Figure 1 shows one of the first integrated FPUs developed by Intel in 1993.

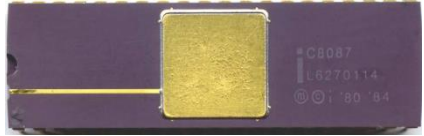


Date	System	Picture
1980	Intel 8087	
1984	Motorola 68881, 68882	 

Table 1. Floating-point units developed by Intel and Motorola



Figure 1. Central processing unit with an integrated floating-point unit developed by Intel

The optimization of division is as important as the optimization of multiplication because the division is more resource expensive and therefore the results in performance will be significant. Table 2 shows the significant difference in machine cycles between the four basic arithmetic operations. As Table 2 clearly shows, there is an immediate need for dividing faster. To perform fast division, we want algorithms that have low latency in machine cycles and a short time cycle. Moreover, the division is the hardest arithmetic operator to implement in hardware because it is the only one that is less defined and gives less exact answers. Furthermore, the division is the most complex arithmetic operator because its result is usually a rational number that cannot be represented exactly in binary with a fixed number of bits. Therefore, every result of a division will be an approximation to the exact answer.

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

Table 2. Microprocessor and arithmetic operations comparison

Many algorithms have been developed to overcome the issues faced by the FPU and CPU however none of them is considered the best because each comes with its benefits and limitations. In this report, we will study the division algorithms that were developed for dividing huge unsigned real integers and the division algorithms developed for polynomials. We will only consider unsigned numbers because negative numbers are a special case of positive numbers by just putting the minus sign in the quotient at the end of the calculation. In addition to this, we will only consider integers because fast algorithms do not want to bother with the fractional part as this makes them slower. Moreover, we will only consider real numbers and not complex numbers because complex numbers are an algebraic combination and manipulation of real numbers. In addition to this, most applications happen and focus on real numbers. Figure 2 illustrates what division algorithms this report studies. We will then study the algorithms developed for polynomials. Finally, we will compare all algorithms with respect to the number of digits that they can handle and the time they need to execute. From the comparisons, we will conclude that the fastest division algorithm for real unsigned integers is the Newton Raphson division algorithm and for polynomials is the expanded synthetic division algorithm.

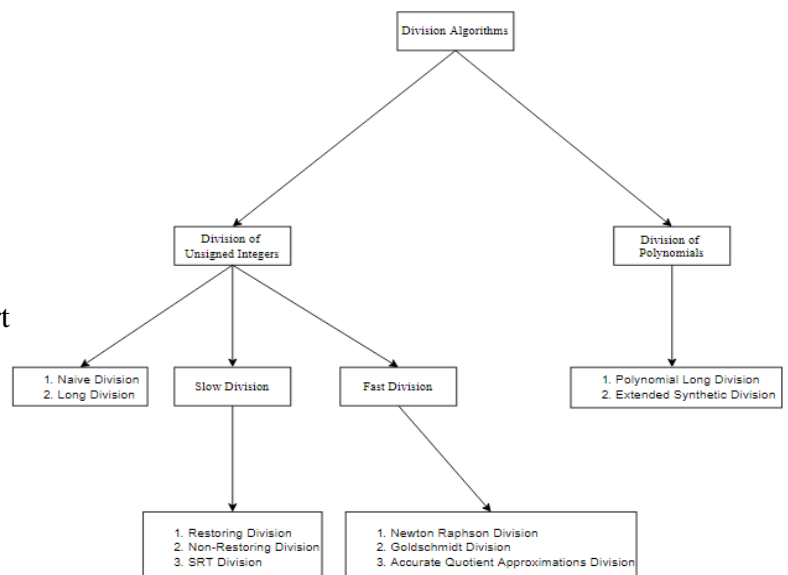


Figure 2. Division algorithms studied in this report

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

GENERAL DIVISION BACKGROUND

Before starting to study each division algorithm, there is a need to understand how division works. The quotient of a particular division can be calculated using Equation 1, where q represents the quotient, n represents the dividend also known as the numerator, and d represents the divisor also known as the denominator. Moreover, Equation 2 also known as Euclid's Division Lemma, shows how the dividend can be calculated, where r represents the remainder. In this report, both n and d are integers and d is never equal to zero because otherwise division is not defined. Equation 2 states that there exists unique q and r such that they satisfy the equation. The division algorithms in this report will focus on calculating the integer values of q and r as fast as possible for large values of n and d . The quotient is the integer part of the answer and the remainder is the fractional part as shown in Equation 3 and the remainder satisfies the condition in Equation 4, where ulp stands for the units in the last place. The precision of the quotient is given by the ulp and the ulp for an integer quotient is equal to 1 which simplifies things. To put everything together, let us say that $n = 10$ and $d = 3$, then $q = 3$ and $r = 1$, since $10 = 3 \times 3 + 1$. Furthermore, let $n = 10$ and $d = -3$, then $q = -3$ and $r = 1$, since $10 = -3 \times -3 + 1$. Finally, all division algorithms in this report have the characteristic of giving a quotient and a remainder and not an exact answer. Therefore, we always consider that the numerator is larger than the denominator because otherwise the result of the quotient will be zero and the remainder will be equal to the denominator. This is the simple case of just returning the denominator and as a result, we cannot compare the algorithms in terms of speed.

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

$$Q = \frac{n}{d}$$

Equation 1. How to compute the quotient

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

Equation 2. Euclid's Division Lemma

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

Equation 3. The final answer of a division

$$r < d \times \text{ulp}$$

$$0 \leq r < d$$

Equation 4. Condition required for the remainder

NAIVE DIVISION ALGORITHM

Now that we know what needs to be calculated let us start with the most obvious and simple method of performing division. This algorithm was used as proof for the greatest common divisor in Euclid's Elements Book VII, Proposition 1. The naive division algorithm finds the quotient (q) and the remainder (r) using repeated subtractions and comparisons. This algorithm takes $\Omega(q)$ steps where and this makes it the slowest division algorithm. In the naive division algorithm, we start with the numerator and keep subtracting the denominator until we get a number that is less than d and greater than or equal to zero. Code A1 in the appendix illustrates this procedure. The number of times that we successfully subtracted d from n represents the quotient and the remaining number represents the remainder. For instance, if we divide $\frac{111}{10}$ we will keep subtracting 10 from 111 until the remainder of 1 which is less than 10 . Since the process was repeated ten times, the quotient will be equal to 10 and the remainder will be equal to 1 . This algorithm is useful for small numerators and small denominators because they result in a small value of q . Moreover, this algorithm is preferred because is simple to implement. However, as Table A1 shows, by using the naive division algorithm we are limited to at most 9 denominator digits and 17 numerator digits. If the number of digits increases just by one, then it

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

takes more than 30 minutes for this algorithm to compute a result. Graph A1 which has an exponential pattern explains the rapid increase from 5 denominator digits to 9. Therefore, for tasks that require more digits, we must implement faster algorithms.

LONG DIVISION ALGORITHM

The long division algorithm is also known as the school book division or pencil and paper division and was invented by Henry Briggs in 1600 AD. This algorithm goes one step further than the naive division algorithm. This algorithm breaks the main problem into smaller simpler sub-problems. In this algorithm, the first step is to combine each digit of the dividend until they form a number that is greater than or equal to the divisor. Next, we find the greatest common multiple of the divisor that is less than the number that we formed above from the digits of the dividend. This multiple becomes the first digit or digits of the quotient. After, we multiply this multiple by the quotient and subtract the result from the dividend to find the remainder. This process continues until all digits of the dividend have been processed and either no remainder is left or the remainder is less than the divisor. Let us look at an example. Let us calculate the following division $\frac{97530}{30}$. This means that the dividend is equal to 97530 and the divisor is equal to 30. The following steps illustrate the procedure to divide these two numbers using the long division algorithm.

- Step 1: Take 9 from the dividend and divide it by 30. This gives 0 because 9 is less than 30. This is recorded in the quotient.

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

- Step 2: 0 is multiplied by 30 and this gives 0.
- Step 3: Subtract 0 from 97 and this gives 0.
- Step 4: Combine 9 and 7 from the dividend to form the number 97. 97 is greater than 30.
Therefore, divide 97 by 30. This gives 3 and it is recorded in the quotient.
- Step 5: Multiply 30 by 3 and this gives 90.
- Step 6: Subtract 90 from 97 and this gives 7.
- Step 7: 7 is less than 30 and therefore there is a need to combine 7 with the next digit of the dividend which is 5.
- Step 8: 75 is larger than 30. Therefore, 75 divided by 30 gives 2.
- Step 9: Multiply 30 by 2 and it gives 60.
- Step 10: Subtract 60 from 75 and this gives 15.
- Step 11: 15 is less than 30. Therefore, combine 15 with the next digit of the dividend which is 3 to form the number 153.
- Step 12: 153 divided by 30 is 5.
- Step 13: Multiply 30 by 5 and this gives 150.
- Step 14: Subtract 150 from 153 and this gives 3.
- Step 15: 3 is less than 30. Therefore, combine 3 with the next digit of the dividend which is 0 to form the number 30.
- Step 16: Divide 30 by the divisor to get 1.
- Step 17: Multiplying 1 by 30 gives 30
- Step 18: Subtract 30 from 30 which gives 0 as a remainder.

			0	3	2	5	1
3	0	9	7	5	3	0	
	-	0					
		9	7				
	-	9	0				
			7	5			
		-	6	0			
			1	5	3		
		-	1	5	0		
					3	0	
				-	3	0	
						0	

Figure 3. Long Division Procedure

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

Therefore, the final result is all the digits of the quotient. These digits are 0, 3, 2, 5, and 1. Figure 3 clearly illustrates the above steps. As the steps and Code A2 show, the long division algorithm is a lot more complex than the naive division algorithm. However, the complexity and time needed to develop and implement this algorithm are worth as we can see from Table A2 and Graph A2. Table A2 shows that this algorithm can operate efficiently up to 8,193 digits for the denominator and up to 16,385 digits for the numerator. This makes the long division algorithm suitable for large-scale applications. Moreover, more digits than 32,769 for the denominator and 65,537 for the numerator will require more than 30 minutes for the long division algorithm to solve. Graph A2 which also has an exponential pattern like Graph A1 shows the huge increase in time after 16,385 digits for the denominator and 32,769 digits for the numerator. In this algorithm, the input is two unsigned integers and the output is the quotient and the remainder.

RESTORING DIVISION ALGORITHM

Restoring division algorithm is in the group of slow division algorithms also known as digit recurrence algorithms. Digit recurrence algorithms reserve a fixed number of bits for the quotient in every iteration. There are two types of algorithms that are part of the digit recurrence algorithms. These are the restoring division algorithms and the non-restoring division algorithms. All digit recurrence algorithms have something in common. They are all based on a standard recurrence equation shown in Equation 5, where R_j is the j^{th} partial remainder of the division, B is the base which will always be 2 because we are working in binary, $q_{n-(j+1)}$ is the digit of the quotient in position $n-(j+1)$, and D is the divisor. Furthermore, slow division algorithms produce

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

a single quotient bit/digit in each iteration by adding or subtracting a multiple of the divisor from the dividend. Finally, the restoring division algorithm requires both the dividend and divisor to be positive

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]. The following steps illustrate the process.

- Step 1: Initialize the register Q with the binary value of the dividend, the register M with the binary value of the divisor, register A with zeros, and counter with the binary digits of the numerator (dividend).
- Step 2: Shift the content of register A and Q left as if they are one unit.
- Step 3: Subtract the contents of register M from register A and store the result in register A.
- Step 4: Check if the most significant bit of register A is less than zero. If this is the case, then the least significant bit of register Q is set to zero, and the value of register A is restored by adding the content of register M which yields the value of register A before step 3. If the most significant bit of register A is not less than zero, then the least significant bit of register Q is set to 1.
- Step 5: Decrement the value of the counter (n) by one.
- Step 6: If the value of the counter (n) is zero we end the process. Otherwise, we go back again to step 2 and repeat the process.

- Step 7: At the end, register A contains the remainder and register Q contains the quotient.

Figure 4 clearly illustrates the above steps. What we conclude from this process is that restoring division can require up to $2n + 1$ additions/subtractions when dividing two n -bit numbers. This means $2n$ machine cycles to select all the quotient digits. This is n cycles for the trial additions/subtractions and n cycles for the restoration. The restoration process is what makes this algorithm slow. Let us look at the performance of the restoring division algorithm. This algorithm is fast up to 1,025 digits for the denominator and 2,049 digits for the numerator. Any more digits than 8,193 for the denominator and 16,385 for the numerator require more than 30 minutes for the restoring division to compute. The input of this algorithm is two unsigned real integers and the output is the quotient and remainder. The input is converted into binary inside the algorithm. Compared to the long division algorithm, the restoring division algorithm is slower in all cases. As we can see, the school book division outperforms the sophisticated restoring division algorithm. Moreover, Graph A3 is again exponential. Therefore, let us see if the restoration step is the reason that causes the issue. To do this we will implement a new division algorithm.

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

Equation 5. Slow/Digit Recurrence Algorithms Common Standard Equation

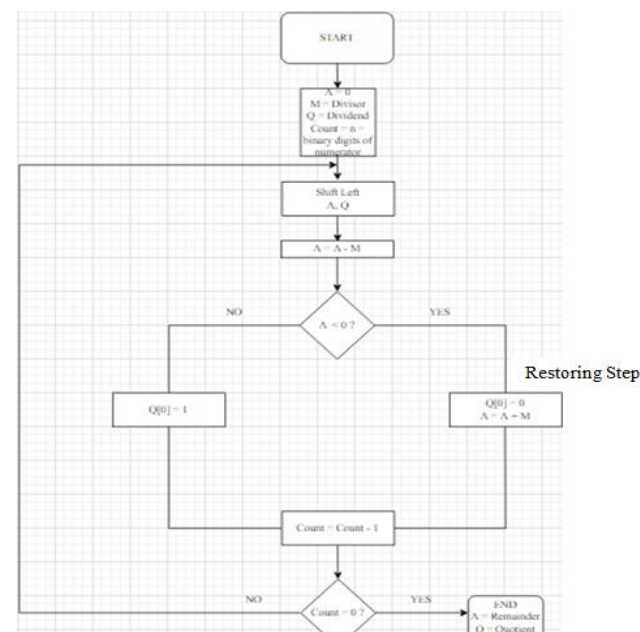
$$q_j = 0 \text{ iff } 2R_{j-1} < D_r$$

$$q_j = 1 \text{ iff } 2R_{j-1} \geq D_r$$

$$R_j = 2R_{j-1} - q_j \times D_r$$

Equation 6. Restoring Division Algorithm Partial Remainder and Quotient

Figure 4. Restoring Division Algorithm Flowchart



Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

NON-RESTORING DIVISION ALGORITHM

In an attempt to make the restoring division algorithm faster we eliminated the restoring step. We eliminated this step because we thought that this was the source of error and the one that made the restoring division algorithm slower than the long division algorithm. This was the idea of Robertson in 1958 and he developed the non-restoring division algorithm. First of all, the non-restoring division algorithm is less complex than the restoring division algorithm because the operations are simpler. This algorithm skips the restoration step of the remainder in every iteration and checks only the sign bit of the remainder. Therefore, the number of operations required is reduced by a factor of two. As with restoring division algorithm, the non-restoring division algorithm operates only with binary values. The following steps describe the process of the non-restoring division algorithm.

- Step 1: Initialize the register Q with the dividend, the register M with the divisor, the register A with zeros, and n with the number of bits of the numerator (dividend).
- Step 2: Check the sign bit of register A.
- Step 3: If the sign bit is one, then shift the left contents of registers A and Q and do the operation $A = A + M$. If the sign bit is zero, shift the left contents of registers A and Q and do the operation $A = A - M$.
- Step 4: Check again the sign bit of register A.
- Step 5: If the sign bit is one, then initialize Q[0] with zero. If the sign bit is zero, then initialize Q[0] with one. Q[0] represents the least significant bit of register Q.

- Step 6: Decrement the value of n by one.
- Step 7: If n is zero go to the next step. If n is not zero go to Step 2 and repeat the process.
- Step 8: Check again the sign bit of register A. If the sign bit is one, then do the operation $A = A + M$ and go to the next step. If the sign bit is zero go immediately to the next step without doing the operation.
- Step 9: Finally, register Q contains the quotient, and register A contains the remainder.

Figure 5 shows the flowchart of the non-restoring division algorithm and how the steps are performed. The Non-restoring division algorithm requires separate hardware for addition and subtraction for each iteration. This makes the non-restoring algorithm resource expensive. Equation 6 represents the partial remainder and quotient for each iteration, where R_j represents the partial remainder and q_j represents the quotient, and D_r represents the divisor. Finally, the digits of the quotient are from the set $[-1, 1]$.

$$\begin{aligned}
 q_j &= -1 \quad \text{iff } R_{j-1} < 0 \\
 q_j &= 1 \quad \text{iff } R_{j-1} \geq 0 \\
 R_j &= 2R_{j-1} + D_r \quad \text{iff } q_j = -1 \\
 R_j &= 2R_{j-1} - D_r \quad \text{iff } q_j = +1
 \end{aligned}$$

Equation 7. Non-Restoring Division Algorithm Partial Remainder and Quotient

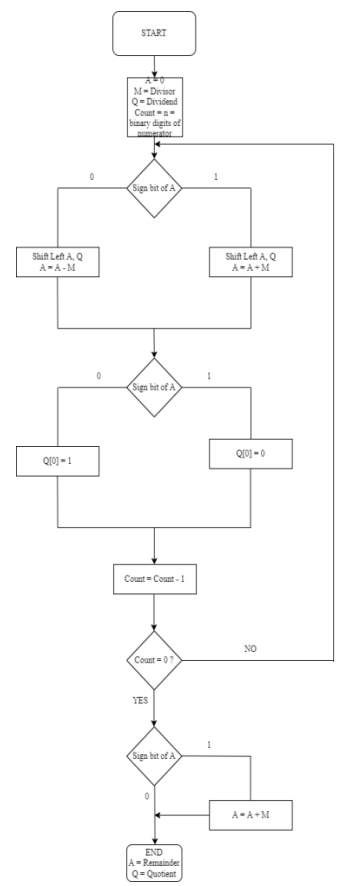


Figure 5. Non-Restoring Division Algorithm Flowchart

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

Let us now see the performance of the Non-restoring division algorithm. Table A4 shows that the algorithm is not performing well. The algorithm is slower than the Restoring division algorithm and the long division algorithm. However, the advantage of this algorithm is that the time remains constant for each trial. This means that the performance of the algorithm does not depend on the number of digits of the numerator or denominator. The factor that affects the performance of the non-restoring division algorithm is the number of bits used to store the numerator and denominator. In Table A4 and Graph A4, 3,394 bits were used. In Table A4.1 and Graph A4.1, 7,934 bits were used. The higher the number of bits the more digits can be used for the numerator and denominator, however, the algorithm becomes slower. For instance, an increase of 4,540 in the number of bits resulted in being 4.4 times slower. Even though it was expected that the non-restoring division algorithm will perform faster than the restoring division algorithm and the long division algorithm this was not the case. In fact, the non-restoring division algorithm was the slowest.

SRT DIVISION ALGORITHM

SRT is a special case of the previous algorithm the most common non-restoring division algorithm. SRT division algorithm was developed by Sweeney, Robertson, and Tocher and it is used in most microprocessors. The Intel Pentium processor (Figure 1) uses a base 4 SRT division algorithm where the quotient digits are from the set $[-2, -1, 0, 1, 2]$. Nowadays, most of the general-purpose central processing units use the SRT division algorithm for the division operator.

The SRT division algorithm generates a fixed number of quotient bits at every iteration. This algorithm uses subtraction as the fundamental operator to calculate a quotient bit/digit in each iteration. The main idea of the SRT division algorithm is to guess the quotient digit based on the most significant bits of the divisor and the partial remainder. The algorithm wants to find an answer close enough to the correct answer and then use redundant quotient bits/digits to correct subsequent steps. In the SRT division algorithm, an n -bit integer division requires, $\frac{n}{k}$, iterations. Equation 7 illustrates the partial remainder and quotient for each iteration in the SRT division algorithm. The digits of the quotient are from the set $[-m, -m + 1, \dots, -1, 0, +1, \dots, m - 1, m]$. The variable m is an integer that consists of k digits of base n and satisfies the equation as shown in Equation 8. Equation 8 also shows the final quotient denoted by Q . The SRT division algorithm reserves b bits of the quotient in each iteration and the base is usually 2. An algorithm of this form performs k iteration to get the quotient which is equivalent to k machine cycles. Therefore, the higher the base used, the lower the latency time. However, as the base increases, the selection of quotient digits is more complicated because the set increases and this increases the machine cycle time. As mentioned above, the quotient digits are guessed based on the most significant bits of the divisor and the partial remainder.

The process of each iteration in the SRT division algorithm is as follows. Figure 6 also illustrates the steps pictorially. The quotient selection table is extremely important and must be considered with accuracy because an incorrect table will produce bugs such as the Intel Pentium's floating-point division bug in 1994 and cost Intel \$475 million.

- Step 1: At the first iteration, the partial remainder is the dividend and then it is multiplied by the base n that is used. This is done by left bit shifting by k bits.
- Step 2: The result of the above step 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 some 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: The process starts from Step 1 until all the quotient bits are revealed.

$$\begin{aligned}
 q_j &= 1 \quad \text{iff} \quad 2R_{j-1} < -D_r \\
 q_j &= 0 \quad \text{iff} \quad -D_r \leq 2R_{j-1} \leq D_r \\
 q_j &= 1 \quad \text{iff} \quad 2R_{j-1} \geq D_r
 \end{aligned}$$

Equation 8. SRT Division Algorithm Partial Remainder and

Quotient

$$\begin{aligned}
 \frac{1}{2}(n-1) &\leq m \leq n-1 \\
 n &= 2^b, b \text{ is the base of the machine used} \\
 k &= \frac{x}{b}, x \text{ is the number of bits of the divisor} \\
 q &= \sum_{j=1}^k q_j n^{-j}
 \end{aligned}$$

Equation 9. Definition of m and Q in the SRT Division Algorithm

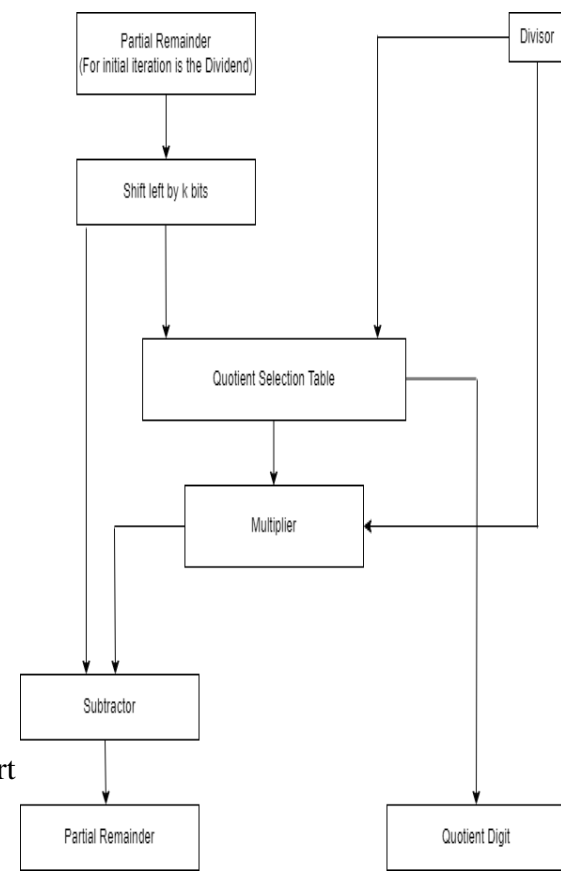


Figure 6. SRT Division Algorithm Flowchart

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

NEWTON RAPHSON DIVISION ALGORITHM

Newton Raphson division algorithm is one of the fastest division algorithms and belongs in the group of the fast division algorithms also known as functional iterative division algorithms or multiplicative division algorithms. The main difference between fast division algorithms and slow division algorithms is that fast division algorithms give more than one quotient digit in each iteration, they are based on multiplication and not addition or subtraction, and finally, they give only the quotient. This algorithm used the Newton Raphson method developed by Isaac Newton and Joseph Raphson and it was used in the IBM RS/600, Intel i860, and Astronautics ZS-1. The Newton Raphson division algorithm makes use of the fact that the quotient is a product of the dividend and the reciprocal of the divisor as shown in Equation 10. The reciprocal of the divisor will converge after several iterations.

The Newton Raphson division algorithm can be divided into three processes. The first process is responsible for finding the initial estimation of the divisor's reciprocal. The next process focuses on the iterative refinement of the divisor's reciprocal. The last process deals with the multiplication step between the dividend and the final convergent divisor reciprocal. The first step is the most difficult. The algorithm uses a priming function which is known to have a root at the divisor's reciprocal. Therefore, the priming function is found in Equation 11. The quadratically converging Newton Raphson equation is given by Equation 12. Equation 12 is applied to Equation 11 and evaluated at X_0 to give Equation 13. Finally, the results of Equation 13 are used to find successive approximations to the divisor's reciprocal. This is shown in

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

Equation 14. The error term is found to be of order $O(n^2)$ and this means by each iteration the error in the divisor's reciprocal decreases quadratically. Equation 14 also shows that in each iteration there are two multiplications and one two's complement operation resulting from the subtraction. Finally, using Equation 10 we find the quotient by multiplying the divisor's reciprocal with the dividend. As Code A5 shows, to enable the algorithm to work on huge numbers, the precision of estimating the divisor's reciprocal must be set extremely high.

Let us now look at the performance of the Newton Raphson division algorithm. Table A5 and Graph A5 display the results of this algorithm. As Table A5 shows, this algorithm is the best according to the previous algorithms. The Newton Raphson division algorithm is able to compute extremely fast divisions up to 131,073 digits for the denominator and 262,145 digits for the numerator. Moreover, it takes just 81 seconds to compute a division consisting of 1,048,577 digits in the denominator and 2,097,153 digits in the numerator. However, the algorithm takes more than 30 minutes to divide with more digits. Graph A5 illustrates these results but like the previous algorithms, this algorithm has exponential growth. The Newton Raphson division algorithm is so fast and works on huge numbers that it does not bother in calculating the remainder. This algorithm takes two unsigned integers as an input and outputs only the quotient.

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

Equation 10. Expressing the Quotient as a Product of the Dividend and the Divisor

$$F(X) = \frac{1}{X} - D_r = 0$$

Equation 11. Priming Function

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

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

Equation 12. Newton Raphson Equation

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

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

Equation 13. Apply Newton Raphson Equation on the Priming Function

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

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

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

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

Equation 14. Approximation to the Divisor's Reciprocal

GOLDSCHMIDT DIVISION ALGORITHM

Goldschmidt division algorithm is also a convergence-base algorithm like the Newton Raphson division algorithm. This algorithm was developed by Robert Elliott Goldschmidt and it is used in modern general-purpose central processing units. The algorithm was also used in the IBM 360/91 and the TMS390C602A Sparc FPU. Just like the Newton Raphson division algorithm, the Goldschmidt division algorithm is also divided into three steps. First, there is a need to generate a multiplication factor. Next, the dividend and the divisor are multiplied by this factor.

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

Finally, if the divisor becomes one or close to one, then just return the dividend which will represent the quotient. If this is not the case, the process is repeated from step 1. The main idea behind this algorithm is the iterative parallel multiplication of the dividend and the divisor by multiplicative factors such that the final divisor will be one or close to one. The Goldschmidt division algorithm is based on the Taylor-Maclaurin series found in Equation 15 to find the multiplication factor required in the first step. In this algorithm, we want to find the expansion of the following equation $q = \frac{a}{b} = a \times f(x)$ where $f(x)$ is able to be computed by a numerical method. Therefore, we let $f(x)$ equal to Equation 15 and we expanded at $x = 0$ which results in the Maclaurin series found in Equation 16. To make $f(x)$ equal to $\frac{1}{b}$, x must be equal to $b - 1$. By doing this substitution in Equation 16, we get Equation 17 where b defines the following condition $0.5 \leq b < 1.0$. The expansion of Equation 17 can also be defined numerically using an approximate quotient as follows $q_i = \frac{N_i}{D_i}$. N_i and D_i are the values of the numerator and denominator respectively after the i^{th} iteration. By forcing D_i to be equal or closed to one, N_i converges towards q .

$$\frac{1}{1+x}$$

Equation 15. Taylor-Maclaurin Series of the Goldschmidt Division Algorithm

$$\frac{1}{1+x} = 1 - x + x^2 - x^3 + x^4 - \dots$$

Equation 16. Expansion of the Taylor-Maclaurin Series of the Goldschmidt Division Algorithm

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

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

$$q = a \times [(1-x)(1+x^2)(1+x^4)(1+x^8)(1+x^{16}) \dots]$$

Equation 17. Quotient of the Goldschmidt Division Algorithm

ACCURATE QUOTIENT APPROXIMATIONS DIVISION ALGORITHM

A subset of the fast division algorithms is named “Very High Radix Algorithms”. The accurate quotient approximations division algorithm is part of this group of algorithms and was developed by Derek C. Wong professor in the Electrical Engineering Department of Stanford University.

To be more precise, “Very High Radix” means dividers that use more than 10 bits of quotient in each iteration. Very high radix algorithms use multiplication for divisor multiple formations and make use of a look-up table to obtain an approximation to the divisor’s reciprocal. The purpose of the look-up table is to get an adjustment to the current quotient. Each adjustment is added to the previous cumulative estimate of the quotient and therefore each new estimate is a modified version of the previous. However, each estimate must always be less than or equal to the true quotient. This is because we want each new adjustment to be always positive. The overall description of the accurate quotient approximations division algorithm is as follows. First, it iterates by using the divisor’s reciprocal to find an approximation to the quotient. Next, the quotient is multiplied by the divisor and the result is subtracted from the dividend to find the remained dividend. This process is repeated until the true quotient is found.

Let us look at the ten steps of the accurate quotient approximations division algorithm. Note that this algorithm reduces the partial remainder by $m - 2$ bits in every iteration. This means, that an n bit quotient requires $\lceil \frac{n}{m-2} \rceil$ iterations. Let Q represent the estimated quotient, X the dividend, and Y the divisor such that they satisfy the following equation $Q = \frac{X}{Y}$. Let X_h be defined as the high order $m + 1$ bits of X concatenated with zeros (0s) to get an n -bit number. Let Y_h be defined as the high order m bits of Y concatenated with ones (1s) to get an n -bit number.

- Step 1: Initialize Q and the variable j to zero.
- Step 2: Approximate $\frac{1}{Y_h}$ from the look-up table using the top m bits of Y and return an m bit approximation. At the same time, perform the multiplication $X_h \times Y$.
- Step 3: Scale Y and scale the product $X_h \times Y$ by the approximation $\frac{1}{Y_h}$. This means doing the following two multiplications $\frac{1}{Y_h} \times Y = Y'$ and $\frac{1}{Y_h} \times (X_h \times Y)$
- Step 4: Let P represent the partial remainder. Let P_h be the truncated version of the partial remainder. Therefore, perform the general recurrence to get the next partial remainder using $P' = P - P_h \times \frac{1}{Y_h} \times Y$.
- Step 5: Calculate the new quotient using $Q' = Q + \frac{P_h}{Y_h} \times \frac{1}{2^j} = Q + P_h \times \frac{1}{Y_h} \times \frac{1}{2^j}$.
- Step 6: Normalize P' by doing left shifting to remove the leading zeros (0s).
- Step 7: The variable j is modified as follows $j' = j + m - 2$
- Step 8: Perform the following adjustments $j = j'$, $Q = Q'$, $P = P'$.

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

- Step 9: Repeat Step 3 up to Step 8 until $j \geq Q$.
- Step 10: Finally, the top n bits of Q give the true value of the quotient. The remainder is found by doing a right shift of $j - Q$ bits on the partial remainder P .

The double-precision quotients in the IEEE consist of 53 bits. To use the accurate quotient approximations division algorithm on these quotients, we will need to set $m = 11$ and a single look-u table will have $2^{11-1} = 1024$ entries each being 11 bits.

POLYNOMIAL LONG DIVISION ALGORITHM

Up to now, we were focusing on dividing unsigned real integers. However, many times we want to divide polynomials. Therefore, there is a need to develop algorithms that divide polynomials. One of these algorithms is called the polynomial long division algorithm. The polynomial long division algorithm is mainly used when we are given one common factor of two polynomials and we want to find the remaining factors. In addition to this, the algorithm can be used to find the equation of the line that is tangent to the graph of the function given by the polynomial $P(x)$ at a point $x = r$. Moreover, this algorithm is used to detect errors in transmitted messages and errors in raw data in digital networks and storage devices.

The polynomial long division algorithm is based on the long division algorithm that we studied before. This algorithm is also based on Euclid's Division Lemmas found in Equation 2. However, the polynomial long division algorithm requires that the remainder must be zero or the degree of the remainder to be less than the degree of the divisor. Having a remainder of zero

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

means that the corresponding dividend and divisor have a common factor. The five core steps of the polynomial long division algorithm are the following and are extremely similar to integer long division.

- Step 1: Divide when going up (writing digits to the quotient).
- Step 2: Multiply out when going down (writing digits to the remainder).
- Step 3: Subtract the result found in step 2 from the original remainder to compute the new remainder.
- Step 4: Carry down digits if necessary from the original remainder to be able to divide with the divisor
- Step 5: Repeat Step 1 up to Step 4 until either the remainder is zero or the degree of the remainder is less than the degree of the divisor.

In this report, we developed the polynomial long division algorithm illustrated in Code A8 to test its performance. We pushed the algorithm to its limits and as Table A8 shows the performance was brilliant. The algorithm was able to do a long division with 123,000-degree polynomials and having 123,000 terms in under one second. Furthermore, Graph A8 shows that the algorithm is linear and not exponential like the previous integer algorithms. There is only one point that is a little noisy however the difference is not big enough to be considered.

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

POLYNOMIAL EXTENDED SYNTHETIC DIVISION ALGORITHM

The extended synthetic division algorithm is a modification of the regular synthetic division algorithm. There is not much difference except from an additional step. Synthetic division is also known as short division because it is faster than long division. The main reasons for this are because synthetic division ignores all variables, it only deals with additions, it uses fewer calculations, and it uses fewer resources. Synthetic division is mostly used to find the roots of polynomials. The only limitation of regular synthetic division is that the divisor needs to be a binomial term of degree one such as $x - b$ and the coefficient of the variable in this case x must also be one.

These are the seven steps needed to divide using the regular synthetic division algorithm.

Figure 7 illustrates the steps by dividing, $\frac{x^3 - 2x^2 + 3x - 4}{x - 2} = x^2 + 0x^1 + 3x^0 + \frac{2}{x - 2}$.

- Step 1: Set the divisor ($x - b$) equal to zero to find its root and put it in the division box.
- Step 2: Write the coefficients of the dividend in descending order and if any terms are missing put zero to fill the gaps.
- Step 3: Bring the first number of the leading coefficient down.
- Step 4: Multiply the number from Step 3 with the divisor and put the result in the next column.
- Step 5: Add the number found in Step 4 with the coefficient directly above it and write the result below.

14 December 2021

- Step 6: Repeat Step 4 and Step 5 until there are no remaining coefficients in the dividend.
- Step 7: Finally, the numbers at the bottom are the final answer with the last number being the remainder.

+2				
+2	1	-2	3	-4
	1			
+2	1	-2	3	-4
	1	2		
+2	1	-2	3	-4
	1	0		
+2	1	-2	3	-4
	1	0	3	
+2	1	-2	3	-4
	1	0	3	2

Figure 7. Regular Synthetic Division Algorithm Procedure

Even though the regular synthetic division algorithm is fast and useful it only works when the divisor has the required format. Therefore, the polynomial extended synthetic division algorithm is based on the regular synthetic division algorithm however it can work by not putting any restrictions on the divisor and allowing it to be of any degree. To do this we need to divide the divisor by its leading coefficient. This is the additional step mentioned in the second sentence of the previous page. Let $f(x)$ be the dividend, $g(x)$ the divisor, and a the leading coefficient of

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

$g(x)$. Equation 18 shows the result after dividing $g(x)$ with its leading term to get $s(x)$. Next, we perform synthetic division using $s(x)$ as the divisor and dividing the quotient by the coefficient a to get the quotient of the original division. . Figure 8 illustrates the steps of using the expanded

synthetic division algorithm to divide $\frac{8x^4+2x^3-4x^2+10}{2x^2+4x-2} = 4x^2 - 7 + 16 + \frac{-78x+42}{2x^2+4x-2}$.

$$S(x) = \frac{g(x)}{a}$$

Equation 18. Calculating the New Function to Perform Synthetic Division

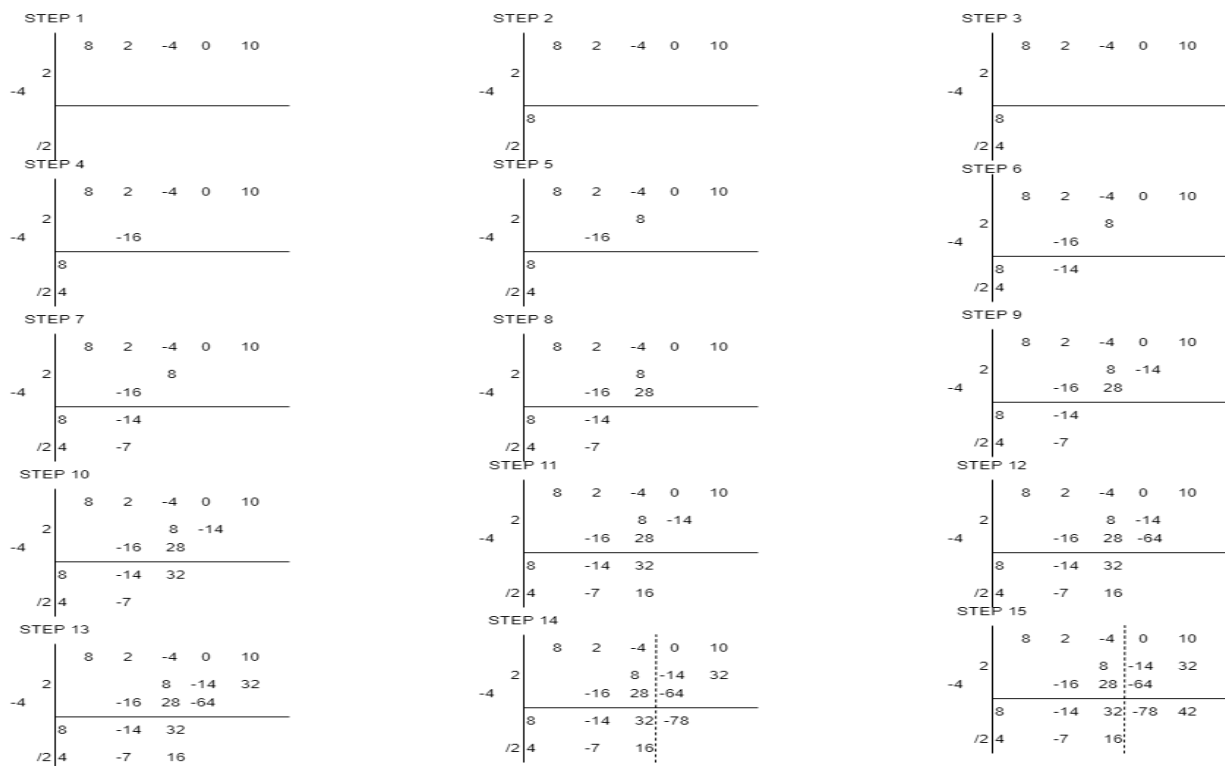


Figure 8. Polynomial Extended Synthetic Division Algorithm Procedure

Code A9 implements the polynomial extended synthetic division algorithm. Table A9 illustrates the amazing performance of this algorithm and shows that this algorithm is two times

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

faster than the simple polynomial long division algorithm. Furthermore, the polynomial extended synthetic division algorithm is a lot simpler to implement and this can be seen if we compare Code A8 and Code A9. Finally, Graph A9 shows that this algorithm is also linear. Once again, there is only one point that is noisy however it can be ignored as the difference is not big enough to affect the algorithm and the experiment.

COMPARING WITH PYTHON'S DIVMOD FUNCTION AND INTEGER DIVISION

When finding out that the Newton Raphson division algorithm was the fastest based on the collected data there was a need to compare it with Python's built-in division functions.

The Python programming language provides the ability to users to perform division using the divmod function. This function returns a tuple containing the quotient and the remainder of the two arguments passed in the function. This function is extremely similar to the division algorithms in this report because all of them take two parameters as input namely the dividend and the divisor and produce a quotient and a remainder. The divmod function was also tested to its limits and the final results are displayed in Table A6. Python's divmod function is also slower than the Newton Raphson division algorithm however when comparing the divmod function with the integer division no conclusion can be drawn. This is because there are times when the divmod function is faster and there are other times that the integer division is faster.

Another built-in function found in the Python programming language is the integer division. This report pushed Python's built-in integer division to its limits and the results are

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

shown in Table A7. The results are extremely similar to the Newton Raphson division algorithm found in Table A5. However, it can be seen that in all cases the Newton Raphson division algorithm is a fraction of a second faster than Python's integer division. The comparison was made with only the integer division because all the division algorithms studied in this report take only integers as input.

The Python programming language also has a built-in function that we can use to compare the two polynomial division algorithms that were mentioned in this report. This function is called `polydiv` and takes two np arrays as input and returns the quotient and remainder. Unfortunately, these arrays are memory expensive and they need powerful machines. When I attempted to create np arrays with 300,000 elements the system crashed as it run out of memory. On the other hand, the polynomial long division algorithm and the polynomial extended synthetic division algorithm did not have such an issue. Therefore, the `polydiv` function is not considered a good solution for dividing extremely high degree polynomials if the machine is not powerful enough.

It is important to mention that all the division algorithms that were discussed in this report must go back to Python's division if they need to find an exact answer. This is easily done by using the returned quotient and remainder of each algorithm and plugging them in Equation 3.

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

CONCLUSION

Overall, the division is the slowest and most complex arithmetic operation. An efficient machine must implement an efficient divider. Over the years lots of algorithms were developed to make it faster as it is essential for the FPU and many complex tasks. Some of these algorithms were successful and some were not. For comparing and identifying which is the best it depends on several circumstances. None of the discussed division algorithms is considered as the “best”. There is a trade-off and the choice of the best algorithm depends on the application, task, and machine used.

From this report and based on Table A5 and Graph A5, we concluded that the fastest division algorithm for large-scale complex tasks is the Newton Raphson division algorithm. However, it is also expected that the Goldschmidt division algorithm and the accurate quotient approximations division algorithm will also be fast. The Newton Raphson division algorithm is so fast that the Python programming language may want to override its integer division algorithm with this algorithm. For smaller scale and simpler tasks, the long division algorithm can also be used.

For polynomial division it is better to implement the polynomial extended synthetic division algorithm rather than the polynomial long division algorithm because it is faster, simpler, and uses fewer resources. The Python programming language may also consider in overriding its polydiv function with the polynomial extended synthetic division algorithm because it solves the issue of memory.

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

This report can be extended by implementing the rest of the division algorithm in the Python programming language so that we can compare more data. Moreover, this report can allow the division algorithms to run more rather than limiting them to 30 minutes. This will generate more solid conclusions and give a clearer picture. Finally, it is also extremely important to mention that there are 10^{78} to 10^{82} number of atoms in the universe and if we compare this number with the 2,097,153 digits used in the Newton Raphson division algorithm and with the 300,002-degree polynomials used in the polynomial extended synthetic division algorithm we may proudly say that we can achieve and calculate the impossible with these division algorithms.

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

REFERENCES

- DJ Anita, Hongal S. Rohini. “Comparative Study of Different Division Algorithms for Fixed and Floating Point Arithmetic Unit for Embedded Applications”, International Journal of Computer Sciences and Engineering, September 2016, Accessed 11 November 2021.
- Draper Jeff, Kwon Taek-Jun, Sondeed Jeff. “Floating-Point Division and Square Root using a Taylor-Series Expansion Algorithm”, Marina del Rey, U.S.A. Accessed 11 November 2021.
- Flynn J. Michael, Oberman F. Stuart. “An Analysis of Division Algorithms and Implementations”, July 1995, Stanford University. Accessed 11 November 2021.
- Koel Ants, Patankar S. Udayan. “Review of Basic Classes of Dividers Based on Division Algorithm”, IEEE Access, January 2021. Accessed 11 November 2021.
- “Chapter 5 Division”, Stanford University. Accessed 11 November 2021.
- “Division algorithm”. Wikipedia, Wikimedia Foundation, 30 October 2021, https://en.wikipedia.org/wiki/Division_algorithm. Accessed 12 November 2021.
- “Non-Restoring Division For Unsigned Integer”, 04 October 2018, <https://www.geeksforgeeks.org/non-restoring-division-unsigned-integer/>. Accessed 13 November 2021.

Professor Erik K. Grimmelmann

CSC 30100

14 December 2021

“Polynomial Long Division”, <https://www.chilimath.com/lessons/intermediate-algebra/polynomial-long-division/>. Accessed 14 November 2021.

“Restoring Division Algorithm For Unsigned Integer”, 22 April 2020,
<https://www.geeksforgeeks.org/restoring-division-algorithm-unsigned-integer/>. Accessed
12 November 2021

“Synthetic division”. Wikipedia, Wikimedia Foundation, 25 August 2021,
https://en.wikipedia.org/wiki/Synthetic_division. Accessed 12 November 2021.

Note: All the division algorithms that were implemented in this report are based on the
pseudocodes found in the following article.

“Division algorithm”. Wikipedia, Wikimedia Foundation, 30 October 2021,
https://en.wikipedia.org/wiki/Division_algorithm