Algorithm Complexity Comparison between Methods Factorization and Euclidean Algorithm to Find the Greatest Common Divisor

Saskia Imani 13517142
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13517142@std.stei.itb.ac.id

***Abstract*—The greatest common divisor (GCD) is the largest integer capable of dividing two different integers. There are two commonly known methods to find the GCD, which are factorization and Euclidean algorithm, both utilizing the modulo operation. This paper aims to prove that using the Euclidean algorithm to find the GCD is more effective compared to using the factorization method. Each method is translated into an algorithmic notation, which complexities will be then be analyzed. The result will be compared, and the algorithm which is least complicated will be the more efficient algorithm.**

***Keywords*—Algorithm, complexity, GCD, Euclidean algorithm, factorization, modulo, remainder.**

I. Greatest Common Divisor

The greatest integer that divides both of two integers is called the greatest common divisor of these integers, or GCD for short [1]. The GCD is mathematically denoted as

$$GCD(a,b) = max\ of\ \{d\ \in\ Z;\ d \mid a\ and\ d \mid b\},$$

where d | a means a is divisible by d, d | b means b is divisible by d, and so forth. In the case that both the value of a and b are 0, the GCD of a and b will also equal to 0.

There are at least two known methods to find the GCD of two random integers, which is using factorization or the Euclidean algorithm.

A. Factorization Method

The first method, factorization, begins with factoring or dividing both the integers (let them be a and b) into smaller integers called factors, which when multiplied will form a and/or b respectively. For example, the factors of a = 3 are {1,3}, since 3 is a prime number; the factors of a = 25 are {1,5,5,25}; and the factors of a = 16 are {1,2,4,4,8,16}. After factoring the two integers a and b, we look for the largest number belonging in the factor set of both a and b, which is the GCD of a and b.

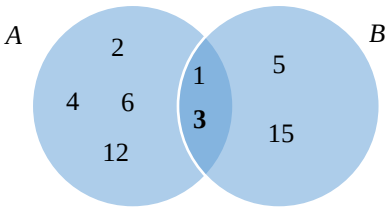

Image 1. A visualization of the factorization method.

A thorough example of this method is as such:

1. Let a = 12 and b = 15.
2. The factors of a = 12 are A = {1,2,3,4,6,12}, while the factors of b = 15 are B = {1,3,5,15}.
3. The largest number belonging in both A and B is 3, and thus 3 is the GCD of a = 12 and b = 15.

B. Euclidean Algorithm Method

The second method to find the GCD of two integers uses the Euclidean equation or Euclidean algorithm:

$$a = b \cdot q_1 + r_1,$$
$$b = r_1 \cdot q_2 + r_2,$$
$$\dots$$
etc.

where a, b, q_1, r_1, q_2 and r_2 are integers and a is bigger than b [1]. The equation above illustrates the modulo operation, where r_1 is the remainder of a divided by b, or the result of a mod b. For example, let a = 8 and b = 3. The integers are fitted into the equation as such:

$$8 = 3 \cdot 2 + 2$$

with q = 2 and r = 2.

The Euclidean equation's lemma concludes that

$$GCD(a,b)\ =\ GCD(b,r_1)$$

By using the Euclidean algorithm consecutively, we will eventually come across a time where the equation results in r_k = 0. In that very same equation, q_k is the GCD of a and b.

A thorough example of this method is as such:

1. Let a = 12 and b = 15.
2. Since b is the larger number, the modulo expression used is b mod a → 15 mod 12 → 15 = 12 * 1 + 3. Thus, the result is $r_1 = 3$.
3. Next, we use the results of the first equation to construct the expression 12 mod 3 → 12 = 3 * 4 + 0. Thus, the result is $r_2 = 0$.
4. Since r_2 equals to 0, then 3 is the GCD of a = 12 and b = 15.

It is important to note that the GCD obtained for a = 12 and b = 15 will always be the same, whether it is obtained using factorization or the Euclidean equation.

II. Algorithmic Notation

An algorithmic notation is more commonly called a pseudocode, and is a type of "language" used to illustrate program algorithms without using any specific programming language. The most obvious advantage of using an algorithmic notation is, once a convention has been created, programmers and developers hailing from different programming backgrounds and language can easily elaborate program structures and algorithms without relying on the others knowing their preferred programming language.

There is no standard for an algorithmic notation, and algorithmic notation does not adhere to any single programming language. However, certain groups of developers may create their own convention, tailoring certain aspects of code to match certain language(s) for further ease of use.

In this paper, we will use a specific algorithmic notation taught and used by the Bandung Institute of Technology, School of Electrical Engineering and Informatics. The algorithmic notation is used to illustrate how the two methods of finding GCD, factorization and Euclidean equation, are applied as program algorithms.

A. Factorization Method

In looking for GCD(a,b) using this method, there is a slightly different approach from the original factorization method, but it uses the same principles. The program will first compare between a and b to look for the greater integer between the two, resulting in x (greater) and y (lesser). The factorization method requires checking all the integers between n = 1 and y to find factors of y, which is every n meeting the condition n | y.

But since we are looking for a common divider, there is no need to factor both x and y. Because we are looking for the "greatest", we will start from the greatest integer between 1 and y, which is n = y. We will see if y is divisible by n. If it is divisible, we will then see if x is divisible by n. If it is divisible, then the integer n must be the GCD of a and b.

```
function Method1 (a, b: integer) → integer

LOCAL VARIABLES
    n: integer

ALGORITHM
    if (a > b) then
        n ← b
    else
        n ← a

    while (n > 1) do
        if (a mod n = 0) and (b mod n = 0)
        then
            → n
        else
            n ← n - 1
    → 1
```

Notation 1. The factorization method.

The algorithm above can be further elaborated as follows:

1. Declare local variable for the integer n.
2. Compare the integers a and b.
3. Assigning the lesser between a and b to n.
4. While n does not equal to 1, repeat the following process:
    a. Divide a by n using modulo, and check if the remainder is 0.
    b. Divide b by n using modulo, and check if the remainder is 0.
    c. If both (a) and (b) is true, return the value n as the GCD of a and b.
    d. If one of (a) and (b) is false, or both are false, decrement n by 1.
5. If n reaches 1, return 1 as the GCD of a and b.

B. Euclidean Algorithm Method

In looking for GCD(a,b) using this method, the program will compare the greater between a and b, resulting in x (greater) and y (lesser). Then the program will divide x by y and note the r (remainder). After that, the program will replace x with y and y with r, then divide the new x by the new y and note the next r. The program will repeat the last two steps until a remainder valued 0 is found. When r reaches the value 0, the last y value is the GCD of a and b.

```
function Method2 (a, b: integer) → integer

LOCAL VARIABLES
    x, y: integer
    r: integer

ALGORITHM
    if (a > b) then
        x ← a
        y ← b
    else
        x ← b
        y ← a

    r ← x mod y
```

```
while (r ≠ 0) do
    x ← y
    y ← r
    r ← x mod y
→ y
```

Notation 2. The Euclidean algorithm method.

The algorithm above can be further elaborated as follows:

1. Declare local variables for integers x, y, and r.
2. Compare the integers a and b.
3. Assign the greater between a and b to x, and the lesser to y.
4. Divide x by y using modulo and assign the remainder to r.
5. While r is not equal to 0, repeat the following process:
    a. Replace the value of x with y.
    b. Replace the value of y with r.
    c. Divide the new x by the new y using modulo and assign the remainder to r.
6. After r reaches the value 0, return y as the GCD for integers a and b.

Both notations for functions Method1 and Method2 have been applied, tested, and proven to yield expected results in a C language program.

III. Algorithm Complexity

While the algorithms of the factorization method and the Euclidean algorithm method do not seem to be much different from the amount of lines they take, and the numbered steps of both methods are quite similar, determining which of the two is a more efficient algorithm is not so simple.

The efficiency of an algorithm is determined from the time and space needed to execute the algorithm. The amount of time and space required for an algorithm to work is called the complexity of the algorithm, and is measured by the amount of data the algorithm processes (n). Time complexity is denoted as T(n), while space complexity is denoted as S(n).

The time complexity of an algorithm can be further divided into three main categories:

- Tmin, which is the minimum amount of time for an algorithm to process a number of data in the best-case scenario.
- Tmax, which is the maximum amount of time for an algorithm to process a number of data in the worst-case scenario.
- Tavg, which is the average amount of time for an algorithm to process a number of data. Tavg is determined by the following equation:

$$Tavg = \frac{Tmin + Tmax}{2}$$

An efficient algorithm is one that fulfills its function using the smallest amount of time and space. Determining algorithm complexity will enable developers and programmers to save time and computer storage, which will improve the performance of the program overall.

The need for an efficient algorithm is especially important when the algorithm has exponential complexity. For example, let an algorithm's time complexity be $T(n) = 10^{-4} * 2^n$ seconds. With various amounts of data, we can observe the amount of time this algorithm takes to finish.

When n = 10, the algorithm takes about 1/10 seconds;
when n = 20, the algorithm takes about 2 minutes; and
when n = 30, the algorithm takes more than a day [2].

After creating the algorithmic notation of the two methods to find the GCD, we will now determine the complexity of the two algorithms. The units we will use will be ambiguous, as different computers may have different time required to run the exact same process.

A. Factorization Algorithm

When using the factorization algorithm, the best-case scenario is when n = y is the GCD of a and b, so that the algorithm will only run once. In this case, the time complexity is as such:

$$T_A min(n) = 1$$

Meanwhile, in the worst-case scenario of this algorithm, a and b are prime relatives, so GCD of a and b is 1. According to the algorithm, the value of n spans from y to 2, and if even 2 is not a common divisor of a and b, the answer will be 1. In this case, the algorithm will run for n-1 times, and the time complexity is written as

$$T_A max(n) = n - 1$$

The average time complexity of this particular process is therefore

$$T_A avg(n) = T_A \min(n) + T_A \max(n)$$

$$T_A avg(n) = \frac{1 + (n-1)}{2}$$

$$\boldsymbol{T_A avg(n) = \frac{n}{2}}$$

in which n equals the smaller of two integers a and b.

B. Euclidean Algorithm

Meanwhile, for the Euclidean algorithm, the best-case scenario is when the first x divided by y immediately results in remainder = 0, and thus the algorithm will only run once. In this case, the time complexity is written as

$$T_B min(n) = 1$$

The worst-case scenario, however, is quite complicated to analyze. In the worst-case scenario, the relationship of x and y is illustrated as $x = F_N$ and $y = F_{N-1}$, in which F_N is the Fibonacci sequence: {0,1,1,2,3,5,8,…}. In this case, the algorithm will result in

$$GCD(F_N, F_{N-1}) = GCD(F_{N-1}, F_{N-2})$$

The final result for any x and y part of the Fibonacci sequence will be GCD(x,y) = 1. To find the complexity of the Euclidean algorithm, we must observe the Fibonacci sequence.

$F_0 \rightarrow 0$
$F_1 \rightarrow 1$
$F_2 \rightarrow 1 = 1 * 1 + 0$

But only when we reach F_3 the pattern for the Fibonacci sequence is apparent:

$F_3 \rightarrow 2 = 1 * 2 + 0$ (base)
$F_4 \rightarrow 3 = 2 * 1 + 1$
$F_5 \rightarrow 5 = 3 * 1 + 2$
…
$F_N \rightarrow F_N = F_{N-1} + F_{N-2}$

Through mathematical induction from the equations above, we can conclude that, for the GCD of $x = F_N+1$ and $y = F_N$, it will take N-2 steps to reach the base of the Fibonacci sequence we have set.

However, up until here, we are still using N to describe the time complexity. To compare between the factorization algorithm and the Euclidean algorithm, N must be translated to n. For this, we utilize φ which is something called the "golden ratio".

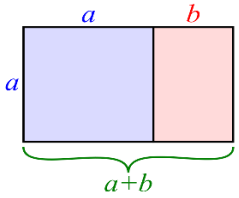

Image 2. A visualization of the golden ration theorem.
(Source: *Wikipedia*)

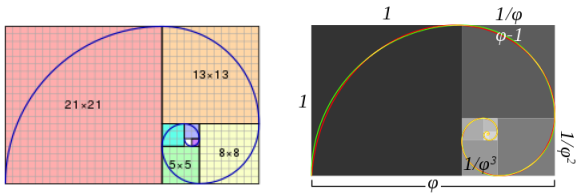

Image 3. The golden spiral, calculated using the Fibonacci sequence (left) and the golden ratio (right).
(Source: *Wikipedia*)

The golden ratio theorem boils down to

$$\frac{(a+b)}{a} = \frac{a}{b} = \varphi$$

Or in our case,

$$\frac{F_n}{F_{n-1}} = \frac{F_{n-1}}{F_{n-2}} = \varphi$$

If the program requires N-2 steps, then y is equal to N-1, which in turn is equal to φ^{N-3} [3]. Therefore,

$$y = \varphi^{N-3}$$
$$\log_\varphi y = N - 3$$

And since

$$\log_\varphi 10 > 1/5 \text{ [4]}$$

we can conclude that

$$\log_{10} \varphi \cdot \log_\varphi y > \frac{(N-3)}{5}$$
$$\log_{10} y > \frac{(N-3)}{5}$$

Thus,

$$5 \cdot \log_{10} y + 3 > N$$

For our complexity problem, this translates to

$$T_B max(n) < 5 \cdot \log_{10} n + 3$$

Therefore, the average time complexity for this algorithm can be written as

$$T_B avg(n) < \frac{5 \cdot \log_{10} n + 3 + 1}{2}$$

$$\boldsymbol{T_B avg(n) \leq \frac{5 \cdot log_{10} n}{2}}$$

in which n is the lesser integer between a and b.

C. Space Complexity

As visible from the algorithm notation of the factorization method and the Euclidean algorithm method, both require initial variable for two integers a and b. The factorization method requires one additional variable for integer n, which will be used to define the GCD.

$$S_A = 1$$

The Euclidean factorization method requires three additional variables for integers x, y, and r.

$$S_B = 3$$

Alternately, we can simple switch the values of a and b as needed, using one temporary variable for the switching process. In this case, the space complexity will be

$$S_B = 2$$

Seeing as there is not much difference between the space complexity of the factorization algorithm and the Euclidean algorithm, we will refrain from using these values for further comparison.

IV. Comparison of Complexity

To compare the two algorithms' complexity, we use the asymptotic complexity of both complexity values, also called the "Big-O" notation.

The definition of the Big-O notation is as follows:

$$T(n) = O(f(n))$$

which means the largest order of T(n) is f(n), if there exists a constant C and n_0 so that

$$T(n) \leq C \cdot f(n)$$

Observe the following table for an example of a time complexity $T(n) = n^2 + 1$:

n	T(n) = 2n² + 3n	n²
1	5	1
10	230	100
100	20300	10000
1000	2003000	1000000
10000	200030000	100000000

Table 1. The comparison of two complexity notations. (Source: [2])

It is clear that the time complexity T(n)'s growth is more similar to the notation n^2 rather than n. So we state that T(n) has the order of n^2 and we write the time complexity notation as such

$$T(n) = O(n^2)$$

A theorem for the Big-O notation is that if

$$T(n) = n^m \pm n^{m-1} \pm n^{m-2} \pm \cdots \pm n$$

then

$$T(n) = O(n^m)$$

This means that the term with higher order dominates the terms with lower orders. "Dominates" means that the growth rate of the time complexity is more similar to the dominating term rather than the other terms of the complexity equation.

Other domination theorems for the Big-O notation includes:

- Exponential terms dominate random exponents, which means for every n > 1:

$$y^n > n^p$$

- Exponents dominate ln n, which means for every n > 1:

$$n^p > \ln n$$

- All logarithms have the same growth rate, which means

$$\log_a n = \log_b n$$

- The term n log n has a faster growth rate compared to n, but slower growth rate compared to n^2.

A summarization of the the domination chain for the Big-O notation is as follows:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < \cdots < O(2^n) < O(n!)$$ [2]

A. Factorization Algorithm

It has been concluded previously that the average time complexity of the factorization algorithm is

$$T_A(n) = \frac{n}{2}$$

To compare the time complexity, we translate this value to its equivalent Big-O notation, which is

$$T_A(n) = \frac{n}{2} = \boldsymbol{O_A(n)}$$

B. Euclidean Algorithm

Meanwhile, the average time complexity of the Euclidean algorithm is

$$T_B(n) \leq \frac{5 \cdot log_{10} n}{2}$$

We translate this value to its equivalent Big-O notation, which is

$$T_B(n) \leq \frac{5 \cdot log_{10} n}{2} = \boldsymbol{O_B(\log n)}$$

C. Comparison

Since we have obtained the Big-O notation of the time complexity for the factorization algorithm and the Euclidean algorithm and they have different orders, we can compare them by simple looking at the domination chain of the Big-O notation. From it, we can derive that

$$\boldsymbol{O_A(n) > O_B(\log n)}$$

which means that the time complexity for the factorization algorithm (A) is higher than the time complexity for the Euclidean algorithm (B).

Therefore, we can conclude that the Euclidean algorithm is a more efficient way of obtaining the GCD of two integers, compared to the factorization method algorithm.

V. Conclusion

The greatest common divisor, or the GCD, of two integers has many uses in programming. One of the most common use is to simplify algorithms for large integers where certain operations are involved, where retaining the original values of both integers are not necessary. It is also used in simplifying

fractions.

With the computer architecture development seemingly reaching a dead-end period, while computer science only grows more complex, the need for efficient algorithms is higher than ever to keep the physical aspect of program development from strain.

From this paper, we have proven that there are two algorithms used to obtain the GCD of two integers, which are the factorization method and the Euclidean algorithm method. We have translated both methods into an algorithmic notation and analyzed the complexity of each method's algorithm. The result is that the complexity of the Euclidean algorithm is lower than the factorization method's algorithm, and thus, the Euclidean algorithm is a much more efficient method of obtaining the GCD of two integers.

VI. Acknowledgments

The author would like to express her gratitude to God Almighty, for only because of His amazing grace the author is able to find the inspiration to begin this paper, and the ability to finish it. The author is also thankful to to Dr. Ir. Rinaldi Munir, M.T., as the beloved lecturer of IF2120 Discrete Mathematics of Class 01, for his dedication and enthusiasm, and unique little quirks in lecturing his students for this semester. The author also wishes to express gratitude to her parents, her sister and brother, and her friends for all their support and help during the writing process of this paper.

The author also remembers the teamwork her class has shown in making sure that every student's paper will be unique and considerably different from each other's. The efforts of the people involved in making the database of paper titles is highly appreciated.

References

[1] Rosen, Kenneth H. (2013). *Discrete Mathematics and Its Applications.* New York: McGraw-Hill, page 265.

[2] Munir, Rinaldi. (2010). *Matematika Diskrit Edisi 3 (Revisi Keempat).* Bandung: Informatika Bandung, page 496-497.

[3] Mollin, R. A. (2008). *Fundamental Number Theory with Applications (2nd edition).* Boca Raton: Chapman & Hall/CRC, page 21-22.

[4] Sloane, N. J. A. (1964). *The On-Line Encyclopedia of Integer Sequences – A000045*. Taken from https://oeis.org/A000045 on December 9, 2018.

Statement

I hereby state that the paper I have written is of my own writing, and not a copy, or a translation, of another person's paper, and is not a result of plagiarism.

Bandung, December 9, 2018

Saskia Imani 13517142