**Algorithm Study Template**

**Algorithm**: Euclid’s Algorithm

**aka**: The Euclidean Algorithm (Euclid’s subtraction method)

**Techniques**: Subtraction

**Categories**: Number Theory, Integer Relations

**Problem**: Euclid’s algorithm is designed to find the greatest common divisor (GCD) between two positive integers. This GCD is the largest positive integer that divides the two numbers without a remainder. For example, the GCD of 16 and 20 is 4 because 4 is the largest number that evenly goes into both 16 and 20 with no remainder.

**Applications**: Euclid’s algorithm has many applications, such as solving Diophantine equations (polynomial equations that allow two or more variables to take integer values only). It is also a key element of RSA encryption and can be used to construct continued fractions in the Sturm chain method for finding real roots of a polynomial. More specifically, it can be useful in reducing fractions to be in lowest terms. For example GCD(42,56) = 14, so if we start with 42/56 and divide both the numerator and denominator by 14, we get ¾ as a reduced fraction. Also, two numbers are said to be relatively prime, or coprime, if their GCD is equal to one. For example, the numbers 9 and 28 are relatively prime because their GCD is one.

**References**:

* <http://en.wikipedia.org/wiki/Euclidean_division>
* <http://en.wikipedia.org/wiki/Greatest_common_divisor>
* <http://planetmath.org/EuclidsAlgorithm>
* <http://mathworld.wolfram.com/EuclideanAlgorithm.html>
* <http://www.math.rutgers.edu/~greenfie/gs2004/euclid.html>
* <http://www.cut-the-knot.org/blue/Euclid.shtml>

**Implementation details**:

* **Big Idea**: To find the GCD of two unequal positive integers, subtract the smaller number from the larger number and then keep comparing sizes and subtracting smaller from larger until the two numbers are equal. At this point, we take the first number to be the GCD.
* **Description**: We want to find the GCD of two positive integers. If they are equal from the start, the first number is taken to be the GCD and no more steps are needed. If the two numbers are different, the smaller number is subtracted from the larger number. Then, the two numbers are checked for equality again, and, if still not equal, the smaller number is subtracted from the larger number again. This process of comparison and subtraction continues until the two numbers are equal to each other. At this point, we take the first number to be the GCD and stop.
* **Pseudo-code**:

function EuclideanGCD(int A, int B){

while(A != B){

if(A > B)

{A = A - B}

else

{B = B - A}

}

return A

}

* **Specific implementation**: (see Euclidean.java)

**Correctness**:

**Theoretical**: Suppose that a and b are integers which are positive but not equal. Now suppose that d is a divisor of both a and b. There exist integers q1 and q2 such that a = q1\*d and b = q2\*d. It follows that a – b = q1\*d – q2\*d = (q1 – q1)\*d. Therefore, d is also a divisor of a-b and GCD(a,b) = GCD(a-b,b). From this argument, one can assume that d is also a divisor of b-a, such that GCD(a,b) = GCD(a,b-a).

**Empirical**: For testing purposes, my program allows the user to input two integers greater than zero. This is to ensure that the values being run through the algorithm are positive. The logic of the program enforces inputs greater than zero by utilizing a try-catch statement and an if-statement. If unacceptable input is found, the user is notified and the program terminates.

To ensure that my implementation of the algorithm was giving correct results, I utilized [this site](http://www.math.sc.edu/~sumner/numbertheory/euclidean/euclidean.html), which is a handy GCD calculator, to check my answers. I simply entered values for n and m, and the site told me what the GCD should be. Then I put those same values into my program to see whether or not I got the same answer. With that in mind, here are some test results from my program, verified by the above website.

Test 1:

A = 10377

B = 873

Euclidean GCD (subtraction method) = 9

Test 2:

A = 1,000,000,000

B = 3775

Euclidean GCD (subtraction method) = 25

Test 3:

A = 1,000,000,000

B = 244

Euclidean GCD (subtraction method) = 4

Test 4:

A = 1,000,000,000

B = 776

Euclidean GCD (subtraction method) = 8

Test 5:

A = 1,000,000,000

B = 92

Euclidean GCD (subtraction method) = 4

**Performance**:

**Theoretical**: When computing GCD(a,b), the worst case is *O(*log *b*), the average case is  T(a)=12π2log2loga+O(1) (per Wikipedia), and the best case is *O*(1).

**Empirical**: The following time measurements are only of the execution time of Euclid’s algorithm. No printed output or other “noise” is intentionally included in the measurements. In addition, I recorded the number of iterations through the while-loop of the algorithm to show how many steps were needed to reach the answer. Please note that these times correspond to the same tests run in the “Empirical Correctness” section of this same report. The tests are numbered the same in both sections so that they can be more easily matched up by readers.

Test 1:

26 iterations performed by subtraction method

GCD computed in 0.006 milliseconds

Test 2:

264,929 iterations performed by subtraction method

GCD computed in 4.1 milliseconds

Test 3:

4,098,374 iterations performed by subtraction method

GCD computed in 6.19 milliseconds

Test 4:

1,288,672 iterations performed by subtraction method

GCD computed in 3.664 milliseconds

Test 5:

10,869,573 iterations performed by subtraction method

GCD computed in 10.866 milliseconds

(These performance results are compared to the performance of the modular division implementation of Euclid’s algorithm at the end of this document.)

**Anecdotes**: In *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 2nd Edition, Donald Knuth says “[Euclid’s Algorithm] is the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day.”

**History**: The first known appearance of the algorithm is in Euclid’s *Elements*, specifically in books seven and ten. The algorithm is not believed to have been discovered by Euclid, who compiled results from earlier mathematicians than himself in *Elements*. Mathematician and historian B.L. van der Waerden has suggested that Euclid’s use of the algorithm for integers is derived from a textbook on number theory, which was written by students in the school of Pythagoras. The algorithm was later independently discovered in India, China, and Europe, where it was largely used for solving Diophantine equations, amongst other uses. New uses for Euclid’s algorithm have been discovered as recently as the 20th century, such as the Ferguson-Forcade algorithm, published in 1979.

**Variations**: Euclid’s algorithm for GCD can also be performed using Euclidean division instead of subtraction. See the following report below for more information.

**Alternatives**: Other methods of finding GCD include listing all divisors of both numbers and identifying the largest common divisor, determining prime factorizations of the two numbers and multiplying common factors together, a binary method involving subtraction and division by two, as well as many others.

**Credits:**

* <http://xlinux.nist.gov/dads/HTML/euclidalgo.html>
* <http://www.proofwiki.org/wiki/Euclidean_Algorithm>
* <http://www.math.sc.edu/~sumner/numbertheory/euclidean/euclidean.html>
* <http://en.wikipedia.org/wiki/Euclidean_algorithm>

**Algorithm Study Template**

**Algorithm**: Euclid’s Algorithm

**aka**: The Euclidean Algorithm (optimized division method)

**Techniques**: Euclidean Division

**Categories**: Number Theory, Integer Relations

**Problem**: Euclid’s algorithm is designed to find the greatest common divisor (GCD) between two positive integers. This GCD is the largest positive integer that divides the two numbers without a remainder. For example, the GCD of 16 and 20 is 4 because 4 is the largest number that evenly goes into both 16 and 20 with no remainder.

**Applications**: Euclid’s algorithm has many applications, such as solving Diophantine equations (polynomial equations that allow two or more variables to take integer values only). It is also a key element of RSA encryption and can be used to construct continued fractions in the Sturm chain method for finding real roots of a polynomial. More specifically, it can be useful in reducing fractions to be in lowest terms. For example GCD(42,56) = 14, so if we start with 42/56 and divide both the numerator and denominator by 14, we get ¾ as a reduced fraction. Also, two numbers are said to be relatively prime, or coprime, if their GCD is equal to one. For example, the numbers 9 and 28 are relatively prime because their GCD is one.

**References**:

* <http://en.wikipedia.org/wiki/Euclidean_division>
* <http://en.wikipedia.org/wiki/Greatest_common_divisor>
* <http://planetmath.org/EuclidsAlgorithm>
* <http://mathworld.wolfram.com/EuclideanAlgorithm.html>
* <http://www.math.rutgers.edu/~greenfie/gs2004/euclid.html>
* <http://www.cut-the-knot.org/blue/Euclid.shtml>

**Implementation details**:

* **Big Idea**: To find the GCD of two positive integers, first make sure neither is equal to zero. Then set the larger number equal to the remainder of itself divided by the smaller number. Continue checking for zero and finding remainders until one of the two numbers is zero. Then, return the maximum of the two numbers as the GCD.
* **Description**: First, check that neither of the two numbers is equal to zero. If either number is, return the maximum of the two numbers as the GCD and stop. If both numbers are greater than zero, set the larger number equal to the remainder of itself divided by the smaller number. Then, keep checking for zero and dividing until one of the two numbers is equal to zero. At this point, return the maximum of the two numbers (one of which is zero) as the GCD and stop.
* **Pseudo-code**:

function EuclideanGCD(int A, int B){

while((A != 0) && (B != 0)){

if(A > B)

{A = A % B}

else

{B = B % A}

}

return Max(A,B)

}

* **Specific implementation**: (see Euclidean.java)

**Correctness**:

**Theoretical**: (Adapted from [proofwiki.org](http://www.proofwiki.org/wiki/Euclidean_Algorithm))

Suppose that a and b are integers and that one of them is greater than zero. From the Division Theorem, a = qb + r, where zero is less than or equal to r and r is less than b. From GCD with Remainder, the GCD of a and b is also the GCD of b and r. Therefore, we may search instead for GCD(b,r). Since r is less than B and B is an integer, we will reach r = 0 after a finite number of steps. At this point, GCD(r,0) = r from GCD with Zero. Note that in this argument, q is quotient and r is remainder.

**Empirical**: For testing purposes, my program allows the user to input two integers greater than zero. This is to ensure that the values being run through the algorithm are positive. The logic of the program enforces inputs greater than zero by utilizing a try-catch statement and an if-statement. If unacceptable input is found, the user is notified and the program terminates.

To ensure that my implementation of the algorithm was giving correct results, I utilized [this site](http://www.math.sc.edu/~sumner/numbertheory/euclidean/euclidean.html), which is a handy GCD calculator, to check my answers. I simply entered values for n and m, and the site told me what the GCD should be. Then I put those same values into my program to see whether or not I got the same answer. With that in mind, here are some test results from my program, verified by the above website.

Test 1:

A = 10377

B = 873

Euclidean GCD (modular method) = 9

Test 2:

A = 1,000,000,000

B = 3775

Euclidean GCD (modular method) = 25

Test 3:

A = 1,000,000,000

B = 244

Euclidean GCD (modular method) = 4

Test 4:

A = 1,000,000,000

B = 776

Euclidean GCD (modular method) = 8

Test 5:

A = 1,000,000,000

B = 92

Euclidean GCD (modular method) = 4

**Performance**:

**Theoretical**: In 1844, Gabriel Lame published a proof showing that the number of steps in this version of Euclid’s algorithm can never be more than five times the number of digits (in base 10) of the smaller number. Thus, when computing GCD(a,b), no more than h\*5 steps can be performed, where h is the number of digits in b.

**Empirical**: The following time measurements are only of the execution time of Euclid’s algorithm. No printed output or other “noise” is intentionally included in the measurements. In addition, I recorded the number of iterations through the while-loop of the algorithm to show how many steps were needed to reach the answer. Please note that these times correspond to the same tests run in the “Empirical Correctness” section of this same report. The tests are numbered the same in both sections so that they can be more easily matched up by readers.

Test 1:

7 iterations performed by modular method

GCD computed in 0.089 milliseconds

Test 2:

7 iterations performed by modular method

GCD computed in 0.198 milliseconds

Test 3:

7 iterations performed by modular method

GCD computed in 0.158 milliseconds

Test 4:

8 iterations performed by modular method

GCD computed in 0.112 milliseconds

Test 5:

6 iterations performed by modular method

GCD computed in 0.099 milliseconds

(These performance results are compared to the performance of the subtraction implementation of Euclid’s algorithm at the end of this document.)

**Anecdotes**: In *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 2nd Edition, Donald Knuth says “[Euclid’s Algorithm] is the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day.”

**History**:

On Euclid’s Algorithm in General:

The first known appearance of the algorithm is in Euclid’s *Elements*, specifically in books seven and ten. The algorithm is not believed to have been discovered by Euclid, who compiled results from earlier mathematicians than himself in *Elements*. Mathematician and historian B.L. van der Waerden has suggested that Euclid’s use of the algorithm for integers is derived from a textbook on number theory, which was written by students in the school of Pythagoras. The algorithm was later independently discovered in India, China, and Europe, where it was largely used for solving Diophantine equations, amongst other uses. New uses for Euclid’s algorithm have been discovered as recently as the 20th century, such as the Ferguson-Forcade algorithm, published in 1979.

On the Optimization Utilizing Euclidean Division:

I was unable to find out when and how this version of the algorithm came into being. However, I did find out that in 1844 Gabriel Lame proved that it never requires more division steps than five times the number of digits (in base 10) of the smaller integer. It is very likely that this version of the algorithm was in use long before this proof was written, though.

**Variations**: Euclid’s algorithm for GCD can also be performed using subtraction instead of Euclidean division. See the preceding report above for more information.

**Alternatives**: Other methods of finding GCD include listing all divisors of both numbers and identifying the largest common divisor, determining prime factorizations of the two numbers and multiplying common factors together, a binary method involving subtraction and division by two, as well as many others.

**Credits:**

* <http://xlinux.nist.gov/dads/HTML/euclidalgo.html>
* <http://www.proofwiki.org/wiki/Euclidean_Algorithm>
* <http://www.math.sc.edu/~sumner/numbertheory/euclidean/euclidean.html>
* <http://en.wikipedia.org/wiki/Euclidean_algorithm>

**Euclidean Methods Compared**

In my tests, I’ve seen a large difference in performance between the subtraction and Euclidean division methods of Euclid’s algorithm. When the input numbers are small, the subtraction method often executes faster, even though it performs more iterations than the division method. This is seen to be the case in test 1, where the division method uses 19 fewer iterations than the subtraction method but has an execution time twice as long. I believe this indicates that the modulus operation takes longer for the computer to perform. Even though there are more subtractions than modular divisions in this test, the subtractions are still few enough that they complete much faster.

In most of my tests, I chose to have the first number be very large and the second number be relatively small so as to maximize the number of iterations being performed by both implementations of Euclid’s algorithm. This was most successful in test 5, where the first number was one billion and the second number was ninety-two. The subtraction method used almost eleven million iterations and completed in about eleven milliseconds. The modular division method used six iterations and completed in about one tenth of a millisecond. This is where the strength of the division method is most clearly seen. Rather than subtracting ninety-two from one billion an enormous number of times, we just divide and get the remainder right away, which decreases the value of the larger number faster than the subtraction method. In this test, I believe the subtraction method performed poorly because, even though the subtraction operations themselves were quick, there were so many of them that the modular division method, whose individual operations are slower, easily outperformed it.

Overall, I would always recommend using the modular division method over the subtraction method because it performs fewer iterations. Even though the subtraction method can execute faster, using fewer iterations is more efficient and, most of the time, uses less processing power. The subtraction method was interesting to look at, especially knowing how old it is, but I wouldn’t recommend it for use in any kind of production environment due to its lack of efficiency.