**Algorithm Study Template**

**Algorithm**: Binary GCD Algorithm

**aka**: Stein’s Algorithm

**Techniques**: Arithmetic shifts, Comparison, Subtraction

**Categories**: Number Theory, Integer Relations

**Problem**: The binary GCD algorithm is used for computing the greatest common divisor (GCD) between two non-negative integers. The GCD is the largest positive integer that divides the two numbers without a remainder. For example, the GCD of 10,377 and 873 is 9 because 9 is the largest number that evenly goes into both 10,377 and 873 with no remainder. Euclid’s algorithm does this as well, but the binary method is designed to be more efficient by use of simpler arithmetic operations like shifts and comparisons. For this assignment, I wanted to see which GCD algorithm would perform better.

**Applications**: Having the GCD 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. More generally, finding the GCD of two non-negative integers is important during RSA encryption and when solving Chinese Remainder Theorem problems due to GCD’s use for finding inverses in modular arithmetic.

**References**:

* <http://xlinux.nist.gov/dads/HTML/binaryGCD.html>
* <http://en.wikipedia.org/wiki/Greatest_common_divisor#Binary_method>
* <http://en.wikipedia.org/wiki/Euclidean_algorithm#Efficiency_of_alternative_methods>
* <http://stackoverflow.com/questions/8192314/binary-gcd-algorithm-vs-euclids-algorithm-on-modern-computers>
* <http://www.csie.nuk.edu.tw/~cychen/gcd/Two%20fast%20GCD%20algorithms.pdf>
* <http://perso.ens-lyon.fr/damien.stehle/downloads/recbinary.pdf>

**Implementation details**:

* **Big Idea**: Find the GCD of two non-negative integers by using arithmetic shifts and comparisons. The algorithm is considered binary because it only uses multiplication and division by two.
* **Description**:

The algorithm starts by determining if either non-negative integer input is equal to zero. If so, the other input is returned as the GCD and no further steps are required because the GCD of zero and any other non-negative integer is the other integer. By this logic, if zero were input for both numbers, the returned GCD would also be zero. Thus, for this implementation, only integers greater than or equal to zero are allowed as inputs.

Next, the algorithm uses bitwise conjunctions (A.K.A. bitwise ANDs) to determine whether one or both inputs, *u* and *v*, are even or odd. If *u* and *v* are both even, then gcd(*u*, *v*) = 2·gcd(*u*/2, *v*/2), because 2 is a common divisor. Note that the division by two is accomplished by a right shift and the multiplication by two is accomplished by a left shift. Continuing on, if *u* is even and *v* is odd, then gcd(*u*, *v*) = gcd(*u*/2, *v*), because 2 is not a common divisor. Similarly, if *u* is odd and *v* is even, then gcd(*u*, *v*) = gcd(*u*, *v*/2). Finally, if *u* and *v* are both odd, and *u* ≥ *v*, then gcd(*u*, *v*) = gcd((*u* − *v*)/2, *v*). If both are odd and *u* < *v*, then gcd(*u*, *v*) = gcd((*v* − *u*)/2, *u*). These are combinations of one step of the simple version of the Euclidean algorithm, which uses subtraction at each step. The division by 2 results in an integer because the difference of two odd numbers is even.

Because of the recursive nature of the algorithm, the steps at the beginning to check for either value being zero are run every time a new level of recursion is entered (this could conceivably be thought of as the next iteration of the algorithm, although not entirely accurately). When one of the values is eventually reduced to zero, the other value is returned as the GCD and the algorithm stops. Once again, we know the value being returned is the GCD because the GCD of zero and any other non-negative integer is the other integer. See [proofwiki.org](http://www.proofwiki.org/wiki/GCD_with_Zero) for further evidence of GCD with zero.

* **Pseudo-code**:

function binaryGCD(int u, int v) {

//simple termination cases

if v equals 0 : return u;

if u equals 0 : return v;

// If u and v are both even, then gcd(u, v) = 2·gcd(u/2, v/2), because 2 is a common divisor

if ((u is even) and (v is even)){

return binaryGCD(u >> 1, v >> 1) << 1;

}

// If u is even and v is odd, then gcd(u, v) = gcd(u/2, v), because 2 is not a common divisor

else if (u is even){

return binaryGCD(u >> 1, v);

}

// If u is odd and v is even, then gcd(u, v) = gcd(u, v/2)

else if (v is even){

return binaryGCD(u, v >> 1);

}

// If u and v are both odd, and u ≥ v, then gcd(u, v) = gcd((u − v)/2, v)

else if (u >= v){

return binaryGCD((u-v) >> 1, v);

}

// If both are odd and u < v, then gcd(u, v) = gcd((v − u)/2, u)

else{

return binaryGCD(u, (v-u) >> 1);

}

}

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

**Correctness**:

**Theoretical**: [Proofwiki.org](http://www.proofwiki.org/wiki/GCD_with_Zero) shows that the GCD of zero and any other non-zero integer is the other integer. The other steps of the algorithm are based on the following properties, deriving from the Fundamental Theorem of Arithmetic, as well as the basic rules of modular arithmetic and division:

1. If u and v are even, gcd(u, v) = 2 gcd(u/2, v/2)
2. If u is even while v is odd, then gcd(u, v) = gcd(u/2, v)
3. If both u and v are odd, then (since u-v is even) |u-v| < max(u,v). Replace the largest of the two with u-v and gcd(u,v)=gcd((u-v)/2,v).

Properties two and three are also true in vice versa, as seen in the above pseudo-code. By using the arithmetic shifts and subtraction in these three properties, one of the two numbers is eventually reduced to zero and returned as the GCD.

**Empirical**: For testing purposes, my program allows the user to input two integers greater than or equal to zero. This means that the case of a user trying to compute gcd(0,0)=0 is allowed. The only disallowed integers are those below zero. The logic of the program enforces this by utilizing try-catch statements and if-statements for each input. 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. I also included the modular division method of Euclid’s algorithm into my program for comparison purposes. I already had the code from a previous assignment, so I thought I could use it to verify that the GCD is being calculated correctly by the binary algorithm and to compare performance. With that in mind, here are some test results from my program, verified by the above website. Note that these tests are numbered because of their correspondence with the tests listed below in the “Empirical Performance” section.

Test 1:

U = 10,377

V = 873

Binary GCD = 9

Euclidean GCD = 9

Test 2:

U = 1,000,000,000

V = 3775

Binary GCD = 25

Euclidean GCD = 25

Test 3:

U = 1,000,000,000

V = 244

Binary GCD = 4

Euclidean GCD = 4

Test 4:

U = 1,000,000,000

V = 776

Binary GCD = 8

Euclidean GCD = 8

Test 5:

U = 1,000,000,000

V = 92

Binary GCD = 4

Euclidean GCD = 4

As one can see, the two GCD methods reached the same result, which was verified by the above website.

**Performance**:

**Theoretical**: The worst-case time is *O*(n^2), where n is the number of bits in the larger of the two numbers. Although each step reduces at least one of the numbers by at least a factor of 2, the subtraction and shift operations take linear time for very large integers (although they're still quite fast in practice, requiring about one operation per word of the representation).

**Empirical**: The following time measurements are only of the execution times of the Binary GCD algorithm and the modular division method of Euclid’s algorithm. No printed output or other “noise” is intentionally included in the measurements. Note that these tests are numbered because of their correspondence with the tests listed above in the “Empirical Correctness” section.

Test 1:

Binary GCD computed in 0.008 milliseconds

Euclidean GCD computed in 0.105 milliseconds

Test 2:

Binary GCD computed in 0.029 milliseconds

Euclidean GCD computed in 0.206 milliseconds

Test 3:

Binary GCD computed in 0.033 milliseconds

Euclidean GCD computed in 0.21 milliseconds

Test 4:

Binary GCD computed in 0.035 milliseconds

Euclidean GCD computed in 0.132 milliseconds

Test 5:

Binary GCD computed in 0.009 milliseconds

Euclidean GCD computed in 0.203 milliseconds

As one can see, the binary method outperformed the modular division method of Euclid’s algorithm in each test. This is due to the fact that the binary method utilizes arithmetic shifts to exploit the computer’s binary representation of numbers. Simply put, the modular division of Euclid’s algorithm is slower than the arithmetic shifts of the Binary GCD algorithm. The Binary GCD algorithm’s exact measure of superiority can be thrown off by different processor architectures, but on my computer the Binary GCD algorithm executed about twice as fast in almost every test.

**Anecdotes**: Akhavi and Vallée proved that the Binary GCD algorithm can be up to 60% more efficient than Euclid’s algorithm, but Knuth reported a 15% gain over Euclid’s algorithm in *The Art of Computer Programming* Volume 2, 3rd edition. The consensus seems to be that the increase in efficiency depends on the architecture of the processor in use.

**History**: The Binary GCD algorithm was first published by Israeli physicist Josef Stein in 1967, but may also have existed in 1st-century China. A method for finding the GCD of two numbers was described in the Chinese mathematics book *The Nine Chapters on the Mathematical Art*. The algorithm described there was intended to reduce a fraction, but the wording of the description can be interpreted as the Binary GCD Algorithm.

**Variations**: There is also an iterative method of implementing the Binary GCD algorithm. It starts by removing all common factors of two where both numbers are even, and then computes the GCD of the remaining numbers using cases where one or both numbers are odd, combining these to form the final answer. [Wikipedia](http://en.wikipedia.org/wiki/Binary_GCD_algorithm#Iterative_version_in_C) details this implementation in C.

**Alternatives**: There are at least three ways to implement Euclid’s algorithm and find the GCD of two non-negative integers. I have detailed two of these in a previous assignment, and used one of those to compare performance with the Binary GCD algorithm in this report. Other methods of finding GCD include listing all divisors of both numbers and identifying the largest common divisor, as well as determining prime factorizations of the two numbers and multiplying common factors together.

**Credits:**

* <http://en.wikipedia.org/wiki/Binary_GCD_algorithm>
* <http://www.cut-the-knot.org/blue/binary.shtml>
* <http://www.math.sc.edu/~sumner/numbertheory/euclidean/euclidean.html>