Integer GCD

Dillon VanBuskirk

June 25, 2016

Contents

1 Introduction

I implemented many different algorithms that calculate the greatest common denominator, hereafter written as gcd. The first gcd algorithm that I used was the standard Euclidean algorithm. This algorithm is typically implemented using recursion so it is not used as the integer inputs increased in size above 2^{16}. The Euclidean algorithm also has an extended version. This algorithm returns the gcd as well as two coefficients, a and b, which satisfy $au + bv = d$ where d is the gcd. A third algorithm I implemented is the binary gcd algorithm which uses reductions to speed up the process of finding the gcd. As shown by its name, it reduces the inputs by 2 while multiplying the gcd by 2 in order to decrease the time it takes to find the gcd. Finally, the third gcd algorithm that was implemented is the k-ary algorithm from Sorenson's 1994 paper. This uses the same idea as binary by reducing the inputs, but it is designed to reduce the inputs by a factor of k. I tested many k values to find the most efficient k for many different u, v inputs. This algorithm requires a pair of coefficients, a and b such that it satisfies $au + bv \equiv 0(modk)$. In Sorenson's paper, he

assumes that pre-computation is done to create a table of a's and b's for every k. However, I did not implement this - I implemented Jebelean-Weber algorithm which returns a and b that satisfy $au \equiv bv(modk)$. This could have been done using the extended Euclidean algorithm, but I was not getting correct a and b which satisfied the requirement.

2 Requirements and Results

The requirement of this assignment was to document the differences in time of each of the above algorithms as well as the relation of time to the size of k. I used std::chrono for the high resolution clock which allowed for high accuracy millisecond and microsecond timing. My findings were such that when inputs were small and relatively prime, $|u + v| < 2^{16}$, the standard Euclidean was the fastest. The time was closely followed by Extended Euclidean and binary gcd just after that. Sorenson's k-ary was the slowest, by a relatively significant margin. If the inputs were not prime and clear multiples of each other, Extended Euclidean was by far the fastest - such that it was less than 0.000 milliseconds. It was followed by standard Euclidean, then binary, and finally Sorenson. Even when inputs were large, prime or not, Extended Euclidean was the fastest. Binary gcd was about twice as slow, and the average k-value from Sorenson was twice as slow as Binary. It appears that the closer the k is to $\sqrt{|u+v|}$ the faster the algorithm runs. The further away the value was from that, the slower Sorenson was. In most cases, if k was prime, the gcd would be incorrect - but it was not consistent. In most cases, if k was not prime, the gcd would be correct - but again, it was not consistent. The inconsistency of k-ary with different k's made it very difficult to debug. Where the GCD was correct, the best k value was near or exactly $\sqrt{|u+v|}$, and outside the range of that, the smaller the k the faster it ran.

3 Pseudocode

3.1 Euclidean

The standard Euclidean is not used if the input is large, but I included the algorithm anyway. The Euclidean algorithm is typically implemented using recursion, but I have implemented it as such:

```
int u, v;
unsigned int x;
while (v)
{
        x = u % v;
        u = v;
        v = x;
}
```

```
return u;
```

3.2 Extended Euclidean

The Extended Euclidean algorithm was given to me by Dr. Beavers. It correctly returned the gcd, but sometimes would not work when it was used in Sorenson's algorithm in order to find the a and b that satisfied the Sorenson requirement. The algorithm was implemented so that it received u and v by value, and shared an a and b with the main function through reference. It looked as such:

```
a = 1;
int g = u;
int x = 0;
int y = v;
while (y > 0) {
        int q = g / y;
        int t = g % y;
        int s = a - q * x;
        a = x;
        g = y;
        x = s;
        y = t;
}
b = (g - u * a) / v;
return g;
```

3.3 Binary GCD

The Binary GCD algorithm was given in Sorenson's paper on page 3. It was implemented as such:

```
int g = 1;
while (u % 2 == 0 && v % 2 == 0) {
        g = 2 * g;
        u = u / 2;
        v = v / 2;
}
while (u != 0 && v != 0) {
        if (u % 2 == 0)
                u = u / 2;
        else if (v % 2 == 0)
                v = v / 2;
        else {
                if (u > v)
                        u = abs(u - v) / 2;
                else
```

```
                        v = abs(u - v) / 2;
        }
}
return (g * (u + v));
```

3.4 Sorenson's k-ary

Sorenson's k-ary algorithm is written on page 4 of his paper. It is shown as:

```
while (u != 0 && v != 0) {
        if (u is relatively prime to k) {
                u = u / gcd(u,k);
        }
        else if (v is relatively prime to k) {
                v = v / gcd(v,k);
        }
        else {
                find nonzero integers a, b satisfying au + bv \equiv 0 (mod k
                if (u > v) {
                        u = abs(a * u + b * v) / k;
                }
                else {
                        v = abs(a * u + b * v) / k;
                }
        }
}
if (u == 0)
        return v;
else
        return u;
```

3.5 Jebelean-Weber Algorithm

Jebelean-Weber's algorithm was used to find a and b that satisfied Sorenson's requirement, that is, it satisfied $au + bv \equiv 0(modk)$. It was implemented as such:

```
int c = (u / v) % k;
int f1[2] = { k, 0 };
int f2[2] = { c, 1 };
while (f2[0] >= sqrt(k)) {
        f1[0] = f1[0] - floor((f1[0] / f2[0])) * f2[0];
        f1[1] = f1[1] - floor((f1[0] / f2[0])) * f2[1];
        swap(f1, f2);
}
return f2;
```

4 Testing

Many test cases were created to test the different algorithms. One of the special cases that was tested was an input where u or v was zero. Test cases ranged from prime numbers with no gcd, other than 1, to repeat numbers where $u = v$. Test cases also ranged from 2 to 2 billion. I designed my input file to include the gcd answer as well as the u,v so it was easy to debug. Each algorithm was tested separately and each k value was tested on each u,v pair.

5 Efficiency

The complexity of the standard Euclidean and Extended Euclidean is based on the number of steps it takes to complete. This is given by an average of

$$T(a) = 1/a\Sigma T(a, b)$$

where the sigma is from $0 \leq b < a$.
The Binary GCD algorithm has been proven by Akhavi and Valle that it can be about 60% more efficient than the standard Euclidean algorithm. However, it is still a quadratic function of the number of input digits as well as Euclidean. Sorenson's algorithm is of

$$O(\frac{n}{\log k} + \log^2 n \log\log n)$$

if $\log n \leq k \leq 2^n$ and k is a power of 2 using a number of processors bounded by a polynomial in n and k. However, in my implementation, it was significantly slower than the other gcd algorithms. It was also incredibly inconsistent with different values of k.
Jebelean-Weber's algorithm is shown to have a time complexity of $O(n^2)$ where n represents the number of bits of the two inputs. This was found in Lavault and Sedjelmaci's paper, WorstCase Analysis of Webers GCD Algorithm.