Fields Institute Communications
Volume **00**, 0000

An Analysis of the Generalized Binary GCD Algorithm

Jonathan P. Sorenson
Department of Computer Science and Software Engineering
Butler University
Indianapolis Indiana 46208
USA
sorenson@butler.edu

Abstract. In this paper we analyze a slight modification of Jebelean's version of the k-ary GCD algorithm. Jebelean had shown that on n-bit inputs, the algorithm runs in $O(n^2)$ time. In this paper, we show that the average running time of our modified algorithm is $O(n^2/\log n)$. This analysis involves exploring the behavior of spurious factors introduced during the main loop of the algorithm. We also introduce a Jebelean-style left-shift k-ary GCD algorithm with a similar complexity that performs well in practice.

1 Introduction

The greatest common divisor, or GCD, of two integers u and v is the largest integer that evenly divides both u and v. Computing GCDs is a fundamentally important operation, and has applications in all branches of algorithmic and computational number theory, including almost all cryptosystems based on number theory[2, 14]. Perhaps the most famous algorithm for computing GCDs is the ancient algorithm of Euclid. On inputs u, v of at most n bits in length, Euclid's algorithm will compute $\gcd(u, v)$ using $O(n^2)$ bit operations [4, 5]. The fastest known algorithm for computing GCDs is that of Schönhage[16], which takes $O(n(\log n)^2 \log\log n)$ bit operations. This algorithm is based on FFT arithmetic and is only practical for very large inputs.

The algorithms that are used the most in practice today are probably the binary algorithm [12, 21] and Euclid's algorithm for smaller numbers, and either Lehmer's algorithm [13, 20] or Jebelean's version of the k-ary GCD algorithm [11, 19, 22] for larger numbers. The binary algorithm has an $O(n^2)$ running time, and both Lehmer's algorithm and Sorenson's version of the k-ary algorithm take at most $O(n^2/\log n)$ time. In practice, as we will show in §9, Jebelean's algorithm is

2000 *Mathematics Subject Classification.* Primary 11A05, 11Y16; Secondary 68Q25, 68W30, 68W40.

This work was supported in part by a grant from the Holcomb Research Institute.

competitive with Lehmer's algorithm and Sorenson's algorithm, yet only an $O(n^2)$ running time bound is known for this algorithm.

The main result of this paper is to reconcile, to some degree, the differences between theory and practice for Jebelean's algorithm. We prove that a slightly modified version of Jebelean's k-ary GCD algorithm runs in $O(n^2/\log n)$ time on average, while maintaining the $O(n^2)$ worst-case running time. The modifications are done to make the proofs easier; we consider the modified algorithm to be only a minor variant of the original.

After discussing our model of computation in §2, we review the original algorithm in §3, and give a quick heuristic analysis in §4. We then present the modified algorithm in §5, and we prove our main result in §6.

We then present a left-shift version of Jebelean's algorithm in §7, which we show performs quite well in practice in §9, where we also present data on spurious factors in addition to timing results. We discuss several techniques for how to prevent spurious factors in §8.

2 Model of Computation

Our model of computation is a RAM with potentially infinite memory that is addressed at the bit level (sometimes called the *naive bit complexity* model). Any basic operation on one or two bits takes constant time, as does indirect addressing and any basic flow of control operations. Let x, y be integers with $y \neq 0$. To compute $x \pm y$ or to compare x and y takes $O(\log x + \log y)$ time. Computing xy takes $O(\log x \log y)$ time. Computing $\lfloor x/y \rfloor$ and $x \bmod y$ takes $O(\log(x/y+1) \log y)$ time.

We make use of the following results on basic arithmetic.

Lemma 2.1 *Let x, y, and $k = 2^d$ be positive integers, with $y < x$. If $y \leq k$, then xy, $\lfloor x/y \rfloor$, and $x \bmod y$ can be computed in $O(\log x)$ bit operations. If $y > k$, then then xy can be computed in $O(\log x + (\log x \log y)/\log k)$ bit operations, and $\lfloor x/y \rfloor$ and $x \bmod y$ can be computed in $O(\log x + (\log(1 + \lfloor x/y \rfloor) \log y)/\log k)$ bit operations. These results require a precomputed table of size $O(k^2 \log k)$ bits. It requires at most $O(k^2 (\log k)^2)$ bit operations to compute this table.*

For proofs, see [19, 20].

This lemma can be viewed as a formalization of the notion of a machine's word size.

3 Jebelean's Algorithm

Below is a description of Jebelean's generalized binary GCD algorithm[11]. The `makeodd`(x) function removes all powers of 2 from x and returns as its function value the number of 2s removed. The `findab`(u,v,k) function returns a pair of integers (a, b) such that $au + bv \equiv 0 (\bmod k)$ with $0 < |a|, |b| \leq \sqrt{k}$. See [11, 6, 18, 22] for algorithms for `findab` that take $O((\log k)^2)$ bit operations. The `Euclid`(g,x,y) function returns $g = \gcd(x, y)$ using Euclid's algorithm.

Algorithm `GenBin`(g, u, v): `/** Returns` $g = \gcd(u, v)$. `**/`
  $k := 2^{32}$ or $k := 2^{64}$;
  $e := \min(\texttt{makeodd}(u), \texttt{makeodd}(v))$;
  $u := |u|$; $v := |v|$;
  $u_{\text{orig}} := u$; $v_{\text{orig}} := v$;

```
   while(uv ≠ 0) do:
      makeodd(u);
      if u < v then (u, v) := (v, u); endif;
      if ⌊log2 u⌋ − ⌊log2 v⌋ > log2 k then
         u := u mod v;
      else
         (a, b) :=findab(u,v,k);
         u := |a · u + b · v|;
      endif;
   enddo;
   w := u + v;    /** note either u = 0 or v = 0 **/
   /** Strip off the spurious factors and restore powers of 2 **/
   Euclid(g′,v_orig,w);    /** so g′ := gcd(v_orig, w); **/
   Euclid(g,u_orig,g′);    /** so g := gcd(u_orig, g′) = gcd(u_orig, v_orig, w); **/
   g := g · 2^e;
end GenBin;
```

Jebelean proposed using either an exact division step or a Euclidean step when u and v differ by a factor of k or more. We have chosen to present the algorithm with a Euclidean step. Removing that step would not affect the worst-case running time of the algorithm from a theoretical standpoint. In practice, it helps to include it.

4 Spurious Factors and a Heuristic Analysis

Definition 4.1 Let $s := w/g$ be defined as the *spurious factors* introduced by our k-ary reduction in the main loop.

In general, the reduction $(u, v) \to (v, |au + bv|/k)$ does not preserve the GCD, although we have $\gcd(u, v) \mid \gcd(v, au + bv)$, and in fact $\gcd(v, au + bv)/\gcd(u, v) \mid \gcd(v, a) \cdot t$ where t is a power of 2. Thus, at the conclusion of the main loop, w is a multiple of $g = \gcd(u_{\text{orig}}, v_{\text{orig}})$. The integer s is composed of primes that divide a values used during the algorithm. (In §8 we discuss some techniques for avoiding spurious factors.)

Definition 4.2 Let $P(x)$ denote the largest prime factor of x, and we say that an integer x is y-smooth if $P(x) \le y$. We also define $\Psi(x, y)$ to be the number of y-smooth integers up to x (see [10]).

Then we see that s is $\sqrt{k}$-smooth. We will take advantage of this observation in the following sections.

In practice, it has been observed that s is usually quite small, so that the two Euclidean GCD computations at the end of the algorithm, which compute $g' = \gcd(u_{\text{orig}}, w)$ and $g = \gcd(v_{\text{orig}}, g') = \gcd(v_{\text{orig}}, u_{\text{orig}}, w)$ are very quick. This is easy to see with a heuristic analysis. First we need the following lemma (see Knuth[12, 4.5.2 excercise 14]):

Lemma 4.3 (Knuth) *We have the following:*

- *Let u, v be positive n-bit integers chosen uniformly at random. Then the expected value of $\log \gcd(u, v)$ is $-\zeta'(2)/\zeta(2) \approx 0.56996$.*
- *Let u, v be positive n-bit odd integers chosen uniformly at random. Then the expected value of $\log \gcd(u, v)$ is $-\zeta'(2)/\zeta(2) - (\log 2)/3 \approx 0.33891$.*

If we make the heuristic assumption that the values of v and a that arise during the main loop of algorithm `GenBin` behave sufficiently like random numbers so that Lemma 4.3 applies, then we can conclude that there will be at most a constant number of bits of spurious factors accumulated during each iteration of the main loop.

From [19] we know that the number of main loop iterations is $O(n/\log k)$, where n is the number of bits in u and v. Thus we expect s to be roughly $u^{1/\log k}$ in practice. In fact, as we will see in §9, this is true. If we then allow k to grow with n such that $\log k \sim \log n$, this line of reasoning implies that Jebelean's algorithm should run in average time $O(n^2/\log n)$. This is what we will prove in §6, albeit through a different approach.

5 A Modified Algorithm

We make several modifications to algorithm `GenBin` as stated above. We would expect that in practice, our modified algorithm would not be as fast as Jebelean's original algorithm, and we see this is true is §9

1. For notational convenience, in the main loop we will keep track of the u and v values as u_i and v_i.
2. Rather than fixing k at 2^{32} or 2^{64}, we allow k to grow with the input size, so that k is a even power of two with $\log k$ roughly $0.4 \log n$.
3. To simplify the analysis, we perform a division, or Euclidean step, every iteration.
4. Under certain circumstances we use a recursive call in place of Euclid's algorithm to help remove spurious factors from w. This is to avoid the $O(n^2)$ running time of Euclid's algorithm which can ruin us if w is at all large.

Using Lehmer's algorithm[20] or Sorenson's version of the k-ary GCD algorithm[19] in place of Euclid's algorithm would make the analysis trivial, but we reject this on philosophical grounds. Such a "cop out" would demote Jebelean's algorithm to a glorified preprocessor, with the bulk of the work being done elsewhere. Our goal here is to analyse the behavior of Jebelean's algorithm, so it is crucial that we limit our choices here to either an $O(n^2)$ algorithm such as Euclid's algorithm, or recursion.

Algorithm `ModGenBin`(g, u, v): `/** Returns` $g = \gcd(u, v)$. `**/`
  `if` $u = 0$ or $v = 0$ `then`
    $g := u + v$; `return`;
  `endif`;
  `if` this is not a recursive call `then`
    $n := \lfloor \log_2(\max(u, v)) \rfloor + 1$;
    $d := \lfloor 0.4 \log_2(n) \rfloor + 1$; $d := d + (d \bmod 2)$; $k := 2^d$;
    Perform the precomputation for Lemma 2.1;
  `else`
    Use the caller's values for d, n and k;
  `endif`;
  $u_0 := |u|$; $v_0 := |v|$;
  $e :=$min(`makeodd`(u_0),`makeodd`(v_0));
  $i := 0$;

```
   while(u_i v_i ≠ 0) do:
      if u_i < v_i then (u_i, v_i) := (v_i, u_i); endif;
      (a, b) :=findab(u_i,v_i,k);
      t := |a · u_i + b · v_i|;
      makeodd(t);
      (u_{i+1}, v_{i+1}) := (v_i, t mod v_i);    /** Euclidean step **/
      makeodd(v_{i+1});
      i := i + 1;
   enddo;
   w := u_i + v_i;    /** Note either u_i = 0 or v_i = 0 **/
   u_f := u_0 mod w; v_f := v_0 mod w;
   /** Strip off the spurious factors **/
   if k < w ≤ u_0^{3/4} then
      ModGenBin(g',w,v_f);    /** Recursive call **/
   else
      Euclid(g',w,v_f);
   endif
   Euclid(g,u_f,g');
   g := g · 2^e;
end GenBin;
```

Remarks 5.1

- The Euclidean step will have no effect beyond a swap if t is smaller than v_i.
- We have $u_0 \geq v_0$, and in the recursive calls to `ModGenBin`, the first input (that is, w from the caller) is always odd, so that $e = 0$.
- The base cases for the recursion are if u or v is zero, or if $w \leq k$ or $w > u_0^{3/4}$ at the end of the main loop. These limits on w squeeze together in the recursion; it is easy to show there will be at most $O(\log n)$ recursive calls.
- We leave a proof of correctness to the reader. To appease the skeptical, we did implement this algorithm and it always gave correct answers on thousands of pairs of random inputs. See §9.

6 Complexity Analysis

Our purpose in this section if to prove the following.

Theorem 6.1 *Let u, v be positive integers of at most n bits in length. Then algorithm* `ModGenBin` *computes* $\gcd(u, v)$ *with a worst-case running time of $O(n^2)$, and an average case running time of $O(n^2/\log n)$, where the average is over all pairs of n-bit integers.*

We proceed as follows. First, we show that everything except the calls to `Euclid` and the recursive call takes $O(n^2/\log n)$ time. Using this lemma, we will then show that the worst-case time of our modified algorithm is $O(n^2)$. Note here that n is the number of bits in the larger of u and v; it is not the total length of the input, although the total length can be at most $2n$.

We then pin down the class of inputs that, when excluded, make it relatively easy to prove an $O(n^2/\log n)$ running time. We call such inputs *not well behaved*. We execute that proof, and then show inputs that are not well behaved have relative density of $o(1/\log n)$.

6.1 Quadratic Worst-Case Running Time.

Lemma 6.2 *Let u, v be positive integers of at most n bits in length. Then algorithm* `ModGenBin` *computes* $\gcd(u, v)$*, and the running time of all steps in the algorithm, excluding calls to* `Euclid` *and the recursive call, is* $O(n^2/\log n)$.

Again, correctness is left to the reader.

Proof A straightforward implementation of the `makeodd` function takes $O(n)$ time, and the `findab` function takes at most $O((\log k)^2)$ time.

From the analysis in [19], we can readily deduce that the main loop will run at most $O(n/\log k)$ iterations. From Lemma 2.1 we deduce that the complexity of each iteration is at most $O(n + (\log k)^2)$ bit operations, excluding the Euclidean steps (divisions), which we ignore for the moment. Plugging in the bound $\log k \leq 0.4(\log n) + O(1)$, we have a total time of $O(n^2/\log n)$ up to the end of the main loop, including precomputation, but excluding the division steps.

To handle the Euclidean steps, we borrow the analysis of Lehmer's algorithm [20], which gives a similar situation: $O(n/\log n)$ loop iterations, with a Euclidean step each iteration. Let r denote the number of loop iterations, and q_i the quotient computed by the Euclidean step that iteration. Observing that $\log v_i \leq n$ and $\prod q_i \leq 2^n$, and using Lemma 2.1 for division, the total time is bounded asymptotically by a constant times

$$\sum_{i=1}^{r} \log v_i (1 + \log q_i)/\log n = O(n^2/\log n). \tag{6.1}$$

Again using Lemma 2.1, computing u_f and v_f takes at most $O(n^2/\log n)$ time, as we know $\log w = O(n)$. □

Lemma 6.3 *Let u, v be positive integers of at most n bits in length. Then algorithm* `ModGenBin` *computes* $\gcd(u, v)$ *with a worst-case running time of* $O(n^2)$.

Proof The calls to Euclid's algorithm are at most $O(n^2)$ bit operations each. By the previous lemma, that leaves the conditional recursive call.

Recall that n is the number of bits in the larger of u and v. So if we are performing a recursive call, $\log w \leq (3/4)n$. This gives us a recurrence relation of the form $t(n) = O(n^2) + t(3n/4)$. This is clearly $O(n^2)$ (see, for example, [17, Chapter 2]), and our proof is complete. □

6.2 Bounds with Well Behaved Inputs.

Definition 6.4 Let u, v be positive integers of at most n bits chosen uniformly at random, with $u \geq v$. Let $g = \gcd(u, v)$, and let d, k be computed from n as given by Algorithm `ModGenBin`. We say that the input pair (u, v) is *well behaved* if it satisfies the following properties.

1. $g \leq n$.
2. Let h be the maximum divisor of v such that $P(h) \leq \sqrt{k}$. (In other words, v/h has no prime divisor below $\sqrt{k}$.) Then $h \leq 2^{d^2}$ (or $\log h = O((\log n)^2)$.

3. Let $j \geq 2$. Choose positive integers w_ℓ, $0 \leq \ell \leq j$ that satisfy the following equations:

$$\begin{aligned} w_0 &\leq v_0, && (6.2)\\ w_\ell &\leq w_{\ell-1}^{3/4} \quad \text{for } 0 < \ell < j, && (6.3)\\ w_j &\leq w_{j-1} && (6.4)\\ g \mid w_\ell &\quad \text{for } 0 \leq \ell \leq j, && (6.5)\\ P(w_\ell/g) &\leq \sqrt{k} \quad \text{for } 0 \leq \ell \leq j, && (6.6) \end{aligned}$$

and either $w_j \leq k$ or $w_j > w_{j-1}^{3/4}$. (These are intended to match the ws computed by the algorithm.) Also define $v_{f,0} = v_0 \bmod w_0$ and $v_{f,\ell} = v_{f,\ell-1} \bmod w_\ell$ for $0 < \ell \leq j$. Let h be a maximum divisor of $v_{f,j-1}$ with $P(h) \leq \sqrt{k}$. Then $\log_2 h \leq n/\log n$.

Lemma 6.5 *If the input pair (u, v) is well behaved, then Algorithm* `ModGenBin` *takes $O(n^2/\log n)$ time.*

To prove this lemma, we first require the following, which shows that if w is large, then so is g. This lemma is the motivation for the condition on the recursive call in algorithm `ModGenBin`.

Lemma 6.6 *Let $2/3 \leq c < 1$ and let $g = \gcd(u_0, v_0)$. If $w \geq u_o^c$, then $g \geq u_0^{3c-2+O(1/d)}$.*

Proof Let r denote the number of main loop iterations. We have

$$u_{i+2} = v_{i+1} \leq \frac{|au_i + bv_i|}{k} \bmod v_i \leq \frac{2\max(|a|,|b|)u_i}{k} \leq \frac{2u_i}{\sqrt{k}}. \tag{6.7}$$

Taking logarithms, we have $\log_2 u_{i+2} \leq \log_2 u_i - d/2 + 1$. Applying this $r/2$ times, we have $\log_2 w \leq \log_2 u_0 - (r/2)(d/2-1)$. Using our lower bound on w, we have $c \log_2 u_0 \leq \log_2 w \leq \log_2 u_0 - (r/2)(d/2-1)$, or

$$r \leq \frac{4(1-c)}{d-2} \log_2 u_0. \tag{6.8}$$

Now, let us write $w = g \cdot s$, where $g = \gcd(u, v)$, and s is the product of all the spurious factors introduced by the algorithm. Each time we perform the k-ary reduction spurious factors can be introduced. However, such factors must divide $\gcd(v_i, a)$. Thus, the number of bits added at such a step is bounded by $\log_2 a \leq \log_2(\sqrt{k}) = d/2$. Thus,

$$\log_2 s \leq r \cdot (d/2). \tag{6.9}$$

Plugging in our upper bound on r from above, we obtain

$$\log_2 s \leq \frac{d}{(d-2)}(2-2c)\log_2 u_0 = (2 - 2c + O(1/d)) \log_2 u_0. \tag{6.10}$$

(Recall $n \to \infty$ so $d, k \to \infty$.) Thus, $s \leq u_0^{2-2c+O(1/d)}$, and the result follows. □

Remarks 6.7

- If we were to add a second Euclidean step to each loop iteration, we would be able to prove a lower bound of $g \geq u_0^{2c-1+O(1/d)}$.

- From the work in §4, we would like to be able to show $w \leq u^{O(1/\log k)}$. In fact with this lemma and the assumption that g is small (the input is well behaved) we can only prove $u^{2/3+\epsilon}$, or $u^{1/2+\epsilon}$ if we were to add a second division step to the main loop.

Proof of Lemma 6.5:

First, we claim that we only need concern ourselves with the work done during the base cases for the recursion. To show this, we use Lemma 6.2, and we show that the value of g' that is returned by a recursive call to `ModGenBin` is small.

First note that, unless $w \leq k$ or one of the inputs is zero, at least one recursive call will be made. The base case of $w > u_0^{3/4}$ cannot happen. This is because u, v are well behaved, we deduce that g is sufficiently small that w in turn is small by Lemma 6.6. If $w \leq k$ or one of the inputs is zero, we have no concerns as all our numbers at the end of the main loop are all quite small.

In the recursive call to `ModGenBin`, $g' = \gcd(w, v_f)$ that is computed will be the product of $g = \gcd(u_0, v_0)$ (which is at most n) with spurious factors shared by $v_f = v_0 \bmod w$ and w. However, because our inputs are well behaved, these are bounded by 2^{d^2}. In other words $g'/g \leq 2^{d^2}$. Thus, the total number of bits in $g' = \gcd(w, v_f)$ is $O((\log n)^2)$.

This implies that the final call to `Euclid` will take at most $O(\log u_f \log g') = O(n(\log n)^2)$ time. (Recall that Euclid's algorithm computes the $\gcd(x, y)$ using $O(\log x \log y)$ bit operations [2, 12].) Hence, the total work done, except for the recursive call to `ModGenBin`, is at most $O(n^2/\log n)$.

We now address the work done in the base case reached from the recursive calls. We add a subscript ℓ to our variables in the algorithm to indicate the recursion level, with $\ell = 0$ indicating the original invokation of `ModGenBin`. So, for example, $u_{0,0}$ and $v_{0,0}$ are the original values of u_0 and v_0, and $u_{0,1}$ is the value of u_0 in the first recursive call to `ModGenBin`. When we omit the ℓ subscript, we are referring to values in the first call to `ModGenBin` (the one not called recursively). The value $\ell = j$ occurs when the recursion hits a base case.

By our condition on the recursive calls, we know that the largest input to a recursive call has at most $(3/4)^\ell n$ bits. The question, however, is when the recursion bottoms out, is the work that remains sufficiently small? There are two cases to consider: $w_j \leq k$, or $w_j > u_{0,j-1}^{3/4}$. In the first case ($w_j \leq k$), the time remaining is bounded by $O((\log k)^2) = O((\log n)^2)$, as all our numbers at this point are bounded by k.

In the second case, by Lemma 6.6, we know $g_j \geq u_{0,j}^{1/4+o(1)}$. Thus the length of g_j is within a constant factor of the length of w_j. If g_j and w_j are at most $O(n/\log n)$ bits in length, then the total work is clearly $O(n^2/\log n)$ at the bottom of the recursion. If our inputs are well behaved, this is in fact true. By definition $g_j = g'_{j-1} = \gcd(w_{j-1}, v_{f,j-1})$, and in particular g_j/g_0 divides $v_{f,j-1}/g_0$, and g_j/g_0 is $\sqrt{k}$-smooth. But by our assumption that u, v are well behaved, g_j/g_0 can therefore be at most $O(n/\log n)$ bits in length. But we saw earlier that g_0 has only $O((\log n)^2)$ bits in length. We therefore conclude that the work in the base case of the recursion is bounded by $O(n^2/\log n)$.

This gives us a recurrence relation of the form $t(n) = O(n^2/\log n) + t(3n/4)$. This is clearly $O(n^2/\log n)$ (again, see [17, Chapter 2]), □

6.3 Inputs that are not Well Behaved are Scarce. The following lemmas show that input pairs (u, v) that are not well behaved are rare. Our first lemma implies that the proportion of inputs with $g = \gcd(u, v)$ where $g > n$ is $O(1/n)$.

Lemma 6.8 *Let u and v be positive n-bit integers chosen uniformly at random. Then the probability $\gcd(u, v) > m$ is $O(1/m)$.*

Proof We have

$$\text{Prob}\left(\gcd(u,v) > m\right) \quad \leq \quad \frac{1}{2^{2n}} \sum_{\ell > m} \left\lfloor \frac{2^n}{\ell} \right\rfloor^2 \quad \leq \quad \sum_{\ell > m} \frac{1}{\ell^2} \quad = \quad O\left(\frac{1}{m}\right). \tag{6.11}$$

□

Next we show that the proportion of inputs v with maximum divisor h satisfying $P(h) \leq \sqrt{k}$ is $o(1/\log n)$.

Lemma 6.9 *Let v be a positive n-bit integer chosen uniformly at random. Given $y > 0$, let h be the maximum divisor of v with $P(h) \leq y$. Let $\epsilon > 0$, and $m > 0$ satisfying $\log m > (\log y)^{1+\epsilon}$. Then the probability $h > m$ is bounded by $\exp(-(\log m)^{\epsilon})$.*

Proof This follows immediately from Lemma 2.4 in [15]. □

We apply this with $y = \sqrt{k}$ and $m = 2^{d^2}$, so we can choose $\epsilon = 1$ here.

It remains to show that inputs that fail to be well behaved due to the third condition are rare. To do this, we require several preliminary lemmas. The first of these gives a rough estimate for $\Psi(x, y)$, which we will apply with $x = 2^n$ and $y = \sqrt{k} \approx n^{0.2}$.

Lemma 6.10 *Let x and y be positive integers with $(\log x)^{1/10} \leq y \leq (\log x)^{1/2}$. Then*

$$\log \Psi(x, y) = c(x, y) \cdot y \cdot \left(1 + O\left(\frac{\log\log y}{\log y}\right)\right) \tag{6.12}$$

where

$$c(x, y) = \frac{\log\log x}{\log y} - 1. \tag{6.13}$$

Note that $1 \leq c(x, y) \leq 9$.

Proof From Ennola [7] (see also [10]) we have that

$$\Psi(x, y) = \frac{(\log x)^{\pi(y)}}{\pi(y)!} \prod_{p \leq y} \frac{1}{\log p} \left(1 + O\left(\frac{y^2}{\log x \log y}\right)\right). \tag{6.14}$$

By the prime number theorem [9] we have $\pi(y) = (y/\log y)(1 + O(1/\log y))$. Using Stirling's formula for $n!$ (see [1, 6.1.38]) we deduce that

$$\log \pi(y)! = y \cdot \left(1 + O\left(\frac{\log\log y}{\log y}\right)\right). \tag{6.15}$$

We have

$$\log(\log x)^{\pi(y)} = \pi(y) \log\log x = \frac{y \log\log x}{\log y} \left(1 + O\left(\frac{1}{\log y}\right)\right). \tag{6.16}$$

Finally,

$$\log \prod_{p \le y} \frac{1}{\log p} = -\sum_{p \le y} \log\log p = O\left(\frac{y \log\log y}{\log y}\right). \tag{6.17}$$

Putting these three estimates together completes the proof. □

This next lemma shows that it is very unlikely for an integer to have a highly smooth, large divisor. Although similar to Lemma 6.9, the form of the bound is different and we will apply it in a different context: if j is the number of levels of recursion in `ModGenBin`, we will apply this to the value of $v_{f,j-1}/g_0$. Specifically, we will set $x_1 = 2^{n/\log n}$, $x_2 = 2^n$, and $y = \sqrt{k} \approx n^{0.2}$, and use Lemma 6.10 to evaluate Ψ.

Lemma 6.11 *Let $x_2 > x_1 > 0$ and $y > 0$ be integers. The proportion of integers $\le x_2$ that have a y-smooth divisor that exceeds x_1 is at most $O(\Psi(x_2, y)/x_1)$.*

Proof We have

$$\sum_{\substack{x_1 \le m \le x_2 \\ P(m) \le y}} \frac{1}{m} = \int_{x_1}^{x_2} \frac{1}{t} d\Psi(t,y) = \left.\frac{\Psi(t,y)}{t}\right|_{x_1}^{x_2} + \int_{x_1}^{x_2} \frac{\Psi(t,y)}{t^2} dt = O\left(\frac{\Psi(x_2,y)}{x_1}\right). \tag{6.18}$$

□

The last, and perhaps most difficult, bound for inputs not well behaved is next.

Lemma 6.12 *Let u, v be inputs not well behaved due to a violation of condition (3). The proportion of such n-bit inputs is $o(1/\log n)$.*

Proof A violation here means that the h defined for any one legal choice of j and the w_ℓ, and as a function of v_0, has at least $n/\log n$ bits. We must count the number of such v_0.

So let us assume that h has at least $n/\log n$ bits. This means that $v_{f,j-1}$ is an integer below w_{j-1} that has a divisor of at least $n/\log n$ bits which is $\sqrt{k}$-smooth. By Lemmas 6.11 and 6.10 the proportion of integers up to w_{j-1} that satisfy this property is at most $\exp(O(n^{0.2})) \cdot 2^{-n/\log n}$.

Next observe that

$$v_{f,j-1} = (\cdots((v_0 \bmod w_0) \bmod w_1) \cdots \bmod w_{j-1}). \tag{6.19}$$

If v_0 is a uniform random n-bit integer, then for fixed j and g, the number of choices that lead to an integer $u_{f,j-1}$ that has a large smooth divisor will be at most the proportion of such integers below w_{j-1} times the number of choices for each w_ℓ, $0 \le \ell \le j$. But this is $\exp(O(n^{0.2} j)) \cdot 2^{-n/\log n}$ by Lemma 6.10. We are assuming w_{j-1} has at least $n/\log n$ bits, which implies j cannot exceed $O(\log\log n)$ because $w_\ell \le w_{\ell-1}^{3/4}$. Thus, the proportion of such v_0 is at most $\exp(O(n^{0.2}\log\log n)) \cdot 2^{-n/\log n} = o(1/\log n)$.

□

This completes the proof of our main result.

7 A Left-Shift Algorithm

As was shown in [19], there is a left-shift version of the k-ary GCD algorithm. Considering how successful Jebelean's algorithm is in practice, it makes sense to explore a Jebelean-style left-shift k-ary GCD algorithm, which we present here.

```
Algorithm LGenBin(g, u, v):    /** Returns g = gcd(u, v). **/
    d := 32 or 64; k := 2^d;
    u := |u|; v := |v|;
    u_orig := u; v_orig := v;
    while(uv ≠ 0) do:
        if u < v then (u, v) := (v, u); endif;
        e := ⌊(log_2 u − log_2 v)/d⌋;
        if e > 0 then:
            t := v · k^e;    /** Perform a left shift **/
        else
            t := v;
        endif;
        (a, b) :=Lfindab(u,t,k);
        u := |a · u + b · t|;
    enddo;
    w := u + v; /** note either u = 0 or v = 0 **/
    /** Strip off the spurious factors **/
    Euclid(g',w,v_orig);
    Euclid(g,u_orig,g');
end LGenBin;
```

If we modify this algorithm in the same way algorithm `GenBin` was modified to obtain algorithm `ModGenBin`, we can then show an average case running time of $O(n^2/\log n)$ following the same lines as in §6.

For an algorithm to compute `Lfindab`, see for example [6].

Finally, we note that this algorithm is quite amenable to performing extended GCD calculations, especially if one of the techniques mentioned in the next section is used to prevent the accumulation of spurious factors.

8 Preventing Spurious Factors

Although permitting the accumulation of spurious factors leads to faster algorithms in practice (as we'll see in the next section), there are applications where preventing them is desirable, such as in Jacobi Symbol computation (see [6]) or in extended GCD computation, which might be needed for computing inverses among other things. Here we briefly discuss three techniques for preventing spurious factors.

8.1 Trial Division. This was the technique used in [19]. The idea is to trial divide before the main loop to remove and save all common factors of u and v bounded by $\sqrt{k}$. Then after the main loop, a second trial division phase removes the spurious factors that were introduced, and the saved common factors are restored.

Although the asymptotic running time of this method is $O(n^2/\log n)$, the trial division phases favor a smaller value for k, while the main loop favors a larger value. This tradeoff leads to an optimal value for k that is usually much smaller than the machine word size, and the resulting algorithm is slow.

8.2 Computing $\gcd(v, a)$**.** During the main loop, after the (a, b) pair is found, compute $\ell := \gcd(v, a)$. Since a is small, this takes linear time. If $\ell > 1$, and ℓ divides u (and hence $\gcd(u, v)$). Set $u := u/\ell$ and save ℓ to be multiplied back into the GCD later. Then use the reduction $u := |(a/\ell)u + b(v/\ell)|$. Since spurious factors can only enter if they divide ℓ, this will prevent them.

This technique was used for computing Jacobi symbols in [6]. It is still somewhat slower than Jebelean's algorithm, but this technique does not suffer from the drawbacks of trial division in that large values for k can be used.

8.3 Computing a complementary (c, d) **pair.** Another method for preventing spurious factors is to perform a double reduction: $(u, v) := (|au + bv|, |cu + dv|)$. Here any spurious factors that arise must divide $ad - bc$. Since the `findab` and `Lfindab` functions return (a, b) pairs that are relatively prime, an extended GCD calculation can quickly identify a (c, d) pair where $ad - bc = 1$ and $|c|, |d| \leq \sqrt{k}$. For the left-shift case, this technique causes the algorithm to become quite similar in effect to Lehmer's algorithm. For the right-shift case, one would want $ad - bc$ to be a power of 2; this can be accomplished by repeatedly doubling c and d and performing Bradley reductions [3], which keep c, d bounded by a, b in absolute value.

The author has tried this technique for the case of a table-driven fast single-precision GCD algorithm that seemed to perform very well. Details will be presented in a later paper.

9 Empirical Results

In this final section, we report the results of our experiments.

9.1 Spurious Bits. Here we show that indeed, in practice, the number of bits in the spurious factors is proportional to the number of main loop iterations. We tabulated the number of main loop iterations, the number of bits in the spurious factors produced by each algorithm, and the number of bits per iteration. We performed this experiment on pairs of random inputs ranging from 2^{10} to 2^{15} bits in length.

Input size	`GenBin`			`LGenBin`		
in bits	Iters.	Sp. bits	Bits/Iter	Iters.	Sp. bits	Bits/Iter
2^{10}	60.3	24.4	0.405	61.9	41.5	0.670
2^{11}	120.5	42.1	0.349	123.6	55.4	0.448
2^{12}	241.6	71.6	0.296	247.0	103.6	0.419
2^{13}	476.7	151.5	0.318	487.5	196.6	0.403
2^{14}	965.8	284.2	0.294	987.7	377.5	0.382
2^{15}	1933.0	558.1	0.289	1975.2	758.2	0.384

It is not surprising that the right-shift algorithm (`GenBin`) had fewer spurious bits, as that algorithm makes pains to remove powers of two at every opportunity, and the left-shift algorithm (`LGenBin`) does not. Also, the (a, b) values in `GenBin` are always odd, whereas in `LGenBin` one of a or b might be even. The reader might find it interesting to compare this data with the expected values given in Lemma 4.3: approximately 0.8223 bits for random numbers, and approximately 0.4889 bits for odd random numbers. Note that only one of u or v gets modified each iteration; this may partially explain the lower number of spurious bits per iteration than our

heuristics predict. Also, both algorithms did perform divisions when u and v differ by a factor of k or more, which should reduce the number of spurious bits.

9.2 Timing Results. The timing results presented below were obtained using the Gnu `g++` compiler with the Gnu-MP 4.1.2 multiprecision library. The timings were done on a 1.0GHz Pentium IV running Linux. The times given are averages in milliseconds. Again, we performed this experiment on pairs of random inputs ranging from 2^{10} to 2^{15} bits in length (roughly 300 to 10000 decimal digits in length).

Algorithm	Input Size:	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}
Euclid	Time	0.6260	1.6120	4.7100	14.5600	53.6800	202.7000
	Iters	598.0	1191.8	2423.1	4735.0	9593.1	19190.5
Binary	Time	0.4870	1.4460	4.7400	16.6400	63.4800	244.8000
	Iters	724.6	1447.0	2889.4	5734.1	11542.2	23126.2
RS k-ary	Time	1.0020	2.1730	5.0300	12.2800	34.7200	108.6000
$\gcd(v, a)$	Iters	60.5	121.0	241.9	477.3	967.3	1930.0
`ModGenBin`	Time	1.0940	2.2890	4.9900	11.3800	29.0400	82.7000
	Iters	60.1	120.5	241.0	476.0	964.6	1929.1
`GenBin`	Time	0.9310	1.9610	4.3700	10.0400	25.9600	74.1000
	Iters	60.3	120.5	241.6	476.7	965.8	1933.0
Lehmer	Time	0.2750	0.6360	1.6200	4.4000	14.3600	49.3000
	Iters	64.3	99.3	173.6	314.7	604.9	1184.3
`LGenBin`	Time	0.3830	0.8430	1.9900	5.0600	14.6400	47.2000
	Iters	61.9	123.6	247.0	487.5	987.7	1975.2
Gnu-MP	Time	0.1160	0.3210	0.9600	2.9800	10.6400	39.6000

We used an implementation of the Modified Lehmer algorithm from [20].

The Right-Shift k-ary method uses the $\gcd(v, a)$ technique to prevent spurious factors. We did not bother to time the trial division method. We used $k = 2^{62}$ for the right-shift k-ary style algorithms, and we used $k = 2^{32}$ for the left-shift one.

The Gnu-MP built-in GCD algorithm uses an optimized implementation of Jebelean's algorithm [8].

Remarks 9.1 Which algorithm should we use in practice?

The results given here only apply to a particular implementation on a Pentium running Linux. The main tradeoff seems to be the relative cost of computing various (a, b) (and sometimes (c, d)) coefficients versus the cost of the arithmetic in the main loop. But clearly one should either use Lehmer's algorithm, expecially for extended GCD computation or computing inverses, or use a version of Jebelean's algorithm.

Acknowledgements

The author would like to thank the referees for their hard work and their very helpful comments. This paper is much better because of them.

References

[1] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions*. Dover, 1970.

[2] Eric Bach and Jeffrey O. Shallit. *Algorithmic Number Theory*, volume 1. MIT Press, 1996.
[3] Gordon H. Bradley. Algorithm and bound for the greatest common divisor of n integers. *Communications of the ACM*, 13(7):433–436, 1970.
[4] G. E. Collins. The computing time of the Euclidean algorithm. *SIAM Journal on Computing*, 3(1):1–10, 1974.
[5] J. Dixon. The number of steps in the Euclidean algorithm. *Journal of Number Theory*, 2:414–422, 1970.
[6] Shawna M. Meyer Eikenberry and Jonathan P. Sorenson. Efficient algorithms for computing the Jacobi symbol. *Journal of Symbolic Computation*, 26(4):509–523, 1998.
[7] V. Ennola. On numbers with small prime divisors. *Ann. Acad. Sci. Fenn. Ser. A I*, 440, 1969. 16pp.
[8] Gnu MP version 4.1.2 online reference manual. http://swox.com/gmp/manual/index.html, 2002.
[9] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, 5th edition, 1979.
[10] A. Hildebrand and G. Tenenbaum. Integers without large prime factors. *Journal de Théorie des Nombres de Bordeaux*, 5:411–484, 1993.
[11] Tudor Jebelean. A generalization of the binary GCD algorithm. In M. Bronstein, editor, *1993 ACM International Symposium on Symbolic and Algebraic Computation*, pages 111–116, Kiev, Ukraine, 1993. ACM Press.
[12] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, Reading, Mass., 3rd edition, 1998.
[13] D. H. Lehmer. Euclid's algorithm for large numbers. *American Mathematical Monthly*, 45:227–233, 1938.
[14] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, 1997.
[15] Carl Pomerance and Jonathan P. Sorenson. Counting the integers factorable via cyclotomic methods. *Journal of Algorithms*, 19:250–265, 1995.
[16] A. Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. *Acta Informatica*, 1:139–144, 1971.
[17] Robert Sedgewick and Phillipe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, 1996.
[18] Mohamed S. Sedjelmaci and Christian Lavault. Improvements on the accelerated integer GCD algorithm. *Information Processing Letters*, 61:31–36, 1997.
[19] Jonathan P. Sorenson. Two fast GCD algorithms. *Journal of Algorithms*, 16:110–144, 1994.
[20] Jonathan P. Sorenson. An analysis of Lehmer's Euclidean GCD algorithm. In A. H. M. Levelt, editor, *1995 ACM International Symposium on Symbolic and Algebraic Computation*, pages 254–258, Montreal, Canada, July 1995. ACM Press.
[21] J. Stein. Computational problems associated with Racah algebra. *Journal of Computational Physics*, 1:397–405, 1967.
[22] Ken Weber. The accelerated integer GCD algorithm. *ACM Transactions on Mathematical Software*, 21(1):111–122, 1995.