The Greatest Common Divisor

Jeffrey Shallit
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada
shallit@graceland.uwaterloo.ca
http://www.math.uwaterloo.ca/~shallit

Introduction

We are interested in algorithms for computing $\gcd(m, n)$, the greatest common divisor of two integers $m, n \geq 0$.

Routines for integer gcd form an integral part of computer algebra systems.

- For example, if we ask Maple for $\frac{5}{24} + \frac{15}{36}$, we don't get $\frac{45}{72}$:

```
    |\^/|     Maple V Release 4 (WMI Campus Wide License)
._|\|   |/|_. Copyright (c) 1981-1996 by Waterloo Maple Inc.
 \  MAPLE  /  All rights reserved. Maple and Maple V are
 <____ ____>  registered trademarks of Waterloo Maple Inc.
      |       Type ? for help.

> 5/24 + 15/36;
                                5/8
```

- A simple variant of most greatest common divisor algorithms allows us to solve the linear diophantine equation $au+bv = \gcd(u, v)$ and hence compute multiplicative inverses (mod u).

A Correct But Inefficient Method

- If the prime factorization of u is $p_1^{e_1} \cdots p_k^{e_k}$ and v is $p_1^{f_1} \cdots p_k^{f_k}$, then

$$\gcd(u, v) = p_1^{\min(e_1, f_1)} \cdots p_k^{\min(e_k, f_k)}.$$

- But using this as a method to compute $\gcd(u, v)$ is currently infeasible, since there is no known fast algorithm for integer factorization.

Euclid's Algorithm

- Euclid gave an algorithm for computing the greatest common divisor (c. 300 B. C. E.)
- "We might call it the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day." (Knuth)
- Presented in Euclid's *Elements*, Book VII.
- Based on the following observation:

$d \mid u$ and $d \mid v$ iff $d \mid v$ and $d \mid u \bmod v$.

- Thus $\gcd(u, v) = \gcd(v, u \bmod v)$.
- Here by $u \bmod v$ we mean the remainder upon division of u by v, e.g.,

$$u \bmod v = u - v \cdot \lfloor u/v \rfloor.$$

- For example, we find

$$\begin{aligned} \gcd(180, 146) &= \gcd(146, 34) = \gcd(34, 10) \\ &= \gcd(10, 4) = \gcd(4, 2) = \gcd(2, 0) \\ &= 2. \end{aligned}$$

Euclid's Algorithm

- This process can be turned into a recursive algorithm, as follows:

$\textsc{Euclid}(u, v)$ { inputs are integers ≥ 0}
(1) if $(v = 0)$ then
(2) return(u)
(3) else
(4) return$(\textsc{Euclid}(v, u \bmod v))$

- But how good is this algorithm? Is it efficient?
- Digression: complexity theory

Complexity

- Complexity of **algorithms**
  - What use does the algorithm make of scarce resources, such as time, space, and randomness?
  - Interested in upper bounds
- Complexity of **problems**
  - Interested in estimating the complexity of the **most efficient possible** algorithm for the problem
  - Upper bounds given by providing an algorithm; lower bounds much harder

Complexity Bounds

- Complexity bounds are phrased in terms of input size. We define

$$\lg n = \begin{cases} 1, & \text{if } n = 0; \\ 1 + \lfloor \log_2 |n| \rfloor, & \text{if } n \neq 0. \end{cases}$$

  Then $\lg n$ counts the number of bits in the binary representation of n, not counting the sign bit.

- For upper bounds, we usually use "big-O" notation:
  - $f = O(g)$ means that there exist constants c, N such that $f(n) \leq cg(n)$ for all $n \geq N$.

The Notion of "Step"

- "step" — the fundamental unit of computation
- no single unit is natural for all situations
  - in sorting algorithms, usually count the number of comparison steps
  - for matrix algorithms, usually count the number of arithmetic operations (addition, subtraction, multiplication, division) with unit cost

The Notion of "Step"

- problems with estimating computation time
  - different processors have different speeds (handle by suppressing constants with "big-O" notation)
  - unit cost assumption becomes unrealistic when the number of inputs or size of inputs becomes larger than the fixed word size of the computer
  - assuming we can perform arithmetic on integers of arbitrary size leads to results that contradict experience, e.g., we could efficiently factor integers
- experience shows that, in analyzing number-theoretic algorithms, it is useful to equate "step" with "bit operation".

Counting Bit Operations

- Deal only with variables that can take the values 0 and 1
- More complex objects are built out of these logical variables
- The only permitted operations are the logical ones of `AND`, `OR`, and `NOT`.
- More complex operations built out of these
- The cost of computation is the total number of logical operations performed.
- In practice, we determine the bit complexity of basic operations such as addition, subtraction, etc., and base our analysis on these bounds.

Bit Complexity Bounds

$$\mu(m,n) = \begin{cases} m(\lg n)(\lg\lg n), & \text{if } m \geq n; \\ n(\lg m)(\lg\lg m), & \text{otherwise.} \end{cases}$$

Operation	Naive bit complexity	Best known
$a+b$	$\lg a + \lg b$	$O(\lg a + \lg b)$
$a-b$	$\lg a + \lg b$	$O(\lg a + \lg b)$
ab	$(\lg a)(\lg b)$	$O(\mu(\lg a, \lg b))$
$a = qb + r$	$(\lg q)(\lg b)$	$O(\mu(\lg q, \lg b))$

Euclid's Algorithm: First Steps Toward Analysis

Let's rewrite Euclid's algorithm as follows: let $u_0 = u$ and $u_1 = v$. We have

$$\begin{aligned} u_0 &= a_0 u_1 + u_2 \\ u_1 &= a_1 u_2 + u_3 \\ &\vdots \\ u_{n-2} &= a_{n-2} u_{n-1} + u_n \\ u_{n-1} &= a_{n-1} u_n, \end{aligned}$$

where $a_i = \lfloor u_i / u_{i+1} \rfloor$ for all i. Then $u_n = d = \gcd(u, v)$.

Euclid's Algorithm: 19th Century Analyses

Let us define $E(u, v) = n$, the number of division steps in the Euclidean algorithm.

Assume $u > v > 0$.

Theorem (Reynaud, 1811).
We have $E(u, v) \leq v$.

Theorem* (Reynaud, 1821).
We have $E(u, v) \leq v/2$.

Theorem (Finck, 1841).
We have $E(u, v) \leq (2 \lg v) + 1$.

Theorem (Lamé, 1844).
We have $E(u, v) \leq 5$ times the number of decimal digits in v.

Traité d'Arithmétique à l'Usage des Élèves qui se Destinent à l'École Polytechnique

par A.-A.-L. REYNAUD

Sixième édition

Paris

M$^{\text{me}}$ V$^{\text{e}}$ COURCIER, Imprimeur-Lib. pour les Mathématiques, quai des Augustins, n$^{\text{o}}$ 57

1811

p. 34, note #60:

60. *Le nombre de divisions à effectuer, pour obtenir le plus grand commun diviseur entre deux nombres, ne peut jamais excéder le plus petit des deux nombres proposés, car chaque reste étant un nombre entier moindre que le diviseur, les restes diminuent au moins d'une unité à chaque division; de sorte qu'on parviendra au reste zéro, après un nombre de divisions tout au plus égal au plus petit des deux nombres proposés.*

The number of divisions required in order to determine the greatest common divisor of two numbers, can never exceed the smaller of the two given numbers, because each remainder being an integer smaller than the divisor, the remainders decrease by at least one at each division; so that one arrives at a remainder of zero after a number of divisions at most equal to the smaller of the two given numbers.

Finck's 1841 Analysis of Euclid's Algorithm

Theorem.
Let $u > v > 0$ be integers. Then

$$E(u, v) \leq 2 \log_2 v + 1.$$

Proof.

- If $v \leq u/2$, then $u \bmod v < v \leq u/2$
- If $v > u/2$, then $u \bmod v = u - v < u/2$.
- In either case $u \bmod v < u/2$.
- Two division steps of the Euclidean algorithm replace

  $$(u, v) \to (v, r) \to (r, v \bmod r),$$

  where $r = u \bmod v$.
- After two division steps we have replaced u by a number smaller than $u/2$.
- On the other hand, after the first division step u is replaced by a number smaller than v.

- It follows that after at most $2\log_2 v + 1$ division steps, u is replaced by a number smaller than 2, and hence the algorithm terminates.

More Precise Analysis of Euclid's Algorithm

Lamé (1844) improved Finck's result by determining the inputs which elicit the worst-case behavior of the algorithm.

These are the famous Fibonacci numbers, defined by

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2.$$

Lemma. Let u, v be integers with $u > v > 0$, and suppose $E(u, v) = n$. Then $u \geq F_{n+2}$ and $v = F_{n+1}$.

Proof.

Let $u = u_0$ and $v = u_1$. Recall that we can write the Euclidean algorithm in the form

$$\begin{aligned} u_0 &= a_0 u_1 + u_2 \\ u_1 &= a_1 u_2 + u_3 \\ &\vdots \\ u_{n-2} &= a_{n-2} u_{n-1} + u_n \\ u_{n-1} &= a_{n-1} u_n. \end{aligned}$$

More Precise Analysis of Euclid's Algorithm

We prove by induction on n that $u_0 \geq F_{n+2}$ and $u_1 \geq F_{n+1}$.

The claim is true for $n = 1$. For then the entire Euclidean algorithm consists of one division, $u_0 = a_0 u_1$, and since $u_0 > u_1$, to find the least u_0, u_1, we must take $u_1 = 1$, $a_0 = 2$, and hence $u_0 = 2$.

Now assume the claim is true for all $i < n$; we wish to prove it for n. Then the first step of the Euclidean algorithm sets $u_0 = a_0 u_1 + u_2$, and we know that $E(u_1, u_2) = n - 1$. Thus $u_1 \geq F_{n+1}$ and $u_2 \geq F_n$ by induction. Then $u_0 \geq u_1 + u_2$; hence $u_0 \geq F_{n+2}$. ∎

More Precise Analysis of Euclid's Algorithm

Now, using the fact that

$$F_n = \frac{\alpha^n - \beta^n}{\sqrt{5}},$$

where

$$\alpha = (1 + \sqrt{5})/2,$$

and

$$\beta = (1 - \sqrt{5})/2,$$

we get:

Theorem. Let $u > v > 0$. Then

$$E(u, v) < c_1 \log v + 2,$$

where $c_1 = 1/\log \alpha \doteq 2.08$.

The worst case of Euclid's algorithm was also identified by

- Simon Jacob (1564)
- Thomas Fantet de Lagny (1733)
- Émile Léger (1837)

Bit Complexity of Euclid's Algorithm

We have seen that Euclid's algorithm performs $E(u, v) = O(\log u)$ division steps on inputs (u, v) with $u > v > 0$.

The naive bit complexity of division is $O((\log u)^2)$.

Thus we obtain

Theorem. The Euclidean algorithm uses $O((\log u)^3)$ bit operations.

But this can be significantly improved...

Bit Complexity of Euclid's Algorithm

Suppose $u = u_0$ and $v = u_1$ are the inputs. Recall that we have

$$\begin{aligned}
u_0 &= a_0 u_1 + u_2 \\
u_1 &= a_1 u_2 + u_3 \\
&\vdots \\
u_i &= a_i u_{i+1} + u_{i+2} \\
&\vdots \\
u_{n-2} &= a_{n-2} u_{n-1} + u_n \\
u_{n-1} &= a_{n-1} u_n .
\end{aligned}$$

Dividing u_i by u_{i+1} to get a quotient of a_i and a remainder of u_{i+2} can be done in

$$O((\lg a_i)(\lg u_{i+1}))$$

bit operations.

Bit Complexity of Euclid's Algorithm

The total bit complexity (disregarding constant factors) is

$$\sum_{0 \le i \le n-1} (\lg a_i)(\lg u_{i+1})$$

$$\le (\lg v) \sum_{0 \le i \le n-1} \lg a_i$$

$$\le (\lg v)(n + \log_2(\prod_{0 \le i \le n} a_i)).$$

We know $n = O(\log u)$ and it is not hard to see that

$$a_0 a_1 \cdots a_{n-1} \le u.$$

Hence we conclude

Theorem. Let u, v be integers. Euclid's algorithm computes the common divisor of u and v using $O((\lg u)(\lg v))$ bit operations.

The Binary gcd Algorithm

J. Stein discovered a "non-Euclidean" algorithm for computing the gcd (1967). It uses the following easy observations:

(a) If u and v are both even, then

$$\gcd(u, v) = 2\gcd\left(\frac{u}{2}, \frac{v}{2}\right). \qquad (1)$$

(b) If u is even and v is odd, then

$$\gcd(u, v) = \gcd\left(\frac{u}{2}, v\right). \qquad (2)$$

(c) If u and v are both odd, then

$$\gcd(u, v) = \gcd(|u - v|/2, v). \qquad (3)$$

The Binary gcd Algorithm

BINARY GCD(u, v)
{ inputs are $u, v > 0$ }
(1) $g \leftarrow 1$ { g holds powers of 2 in $\gcd(u, v)$ }

(2) while $(u \bmod 2 = 0)$ and $(v \bmod 2 = 0)$ do
(3) $\quad u \leftarrow u/2$
(4) $\quad v \leftarrow v/2$
(5) $\quad g \leftarrow 2g$ { remove powers of 2 }
{ now at least one of u, v is odd }

(6) while $(u \neq 0)$ do
(7) $\quad$ if $(u \bmod 2 = 0)$ then $u \leftarrow u/2$
(8) $\quad$ else if $(v \bmod 2 = 0)$ then $v \leftarrow v/2$
(9) $\quad$ else { both odd }
(10) $\quad\quad t \leftarrow |u - v|/2$
(11) $\quad\quad$ if $u \geq v$ then $u \leftarrow t$ else $v \leftarrow t$
$\quad\quad\quad$ { replace larger of u, v with $\frac{|u-v|}{2}$ }
(12) return $(g \cdot v)$

The Binary gcd Algorithm

Theorem. Let $u, v > 0$ be integers. The function BINARY GCD(u, v) computes the greatest common divisor of u and v in $O((\lg uv)^2)$ bit operations.

Proof.

- The algorithm terminates
  - Each pass through the loops on lines (2)–(5) and (6)–(11) decreases uv by a factor of at least 2.
  - Clearly true for each iteration of the first loop, which divides both u and v by 2.
  - The second loop divides u by 2 or v by 2, whichever is even; if both are odd, then it replaces the larger of u and v with $|u - v|/2$.
  - Hence in at most $1+\log_2 uv$ passes through the loops, we find $uv = 0$.
  - Hence either $u = 0$ or $v = 0$.

The Binary gcd Algorithm

  - But v can never be set to 0; if it were, it would have to occur in lines (10)–(11); this implies $u = v$, and so the algorithm would have set u to 0, not v.
  - Hence u must eventually be set to 0, and the second loop terminates.

- The bit complexity bound holds:
  - The algorithm terminates in at most $1 + \log_2 uv$ passes through the loop.
  - Each of the loop steps (a division by 2; a subtraction) can be done in $O(\lg uv)$ bit operations.

Constructing a gcd-free Basis

Consider the following problem of H. W. Lenstra, Jr.:

Given six positive integers, a, b, c, i, j, k, efficiently determine whether

$$a^i b^j = c^k.$$

- Cannot simply multiply a^i and b^j, and see if the result is c^k, for a^i, b^j, and c^k cannot be computed in polynomial time (in $\lg abcijk$).
- If we knew the prime factorizations of a, b, and c, we could solve the problem efficiently by comparing the exponents for each prime divisor of abc.
- However, there is no known efficient algorithm for factorization.

Constructing a gcd-free Basis

One way to solve the problem is to express $a, b,$ and c not as the product of primes, but as the product of relatively prime pieces. The advantage to this method is that such a "factorization" can be completed in polynomial time.

Let $A = (a_1, a_2, \ldots, a_m)$ be a nonempty list of m positive integers, not necessarily distinct. Let $B = \{b_1, b_2, \ldots, b_n\}$ be a set of integers, each ≥ 2.

We say B is a *gcd-free basis* for A if

(a) $b_i \perp b_j$ for all $i \neq j$; and

(b) there exist mn non-negative integers e_{ij} such that $a_i = \prod_{1 \leq j \leq n} b_j^{e_{ij}}$ for all i, $1 \leq i \leq m$.

For example, $\{2, 7, 15\}$ is a gcd-free basis for the list $(4, 30, 14, 49)$.

Constructing a gcd-free Basis

One reason for the importance of the concept of the gcd-free basis is the following:

Theorem. Let the set B be a gcd-free basis for $A = (a_1, a_2, \ldots, a_m)$. Then each element of A can be expressed uniquely as the product of non-negative powers of elements of B (up to the order of the factors).

Proof.

- Suppose there were two distinct factorizations of a_i, as follows:

$$a_i = b_1^{e_{i1}} \cdots b_n^{e_{in}} = b_1^{f_1} \cdots b_n^{f_n}.$$

- Then

$$b_1^{e_{i1}-f_1} \ \cdots \ b_n^{e_{in}-f_n} = 1.$$

- If the two factorizations were truly different, this product would contain at least one term with a negative exponent, and at least one term with a positive exponent.

- Then, by rearranging, we would have

$$\prod_{i \in S} b_i^{g_i} = \prod_{j \in S'} b_j^{g_j}.$$

- Choose any term $b_i^{g_i}$ on the left-hand side with $g_i \neq 0$, and let p be a prime dividing b_i.
- Then p must divide some $b_j^{g_j}$ on the right-hand side; hence $\gcd(b_i, b_j) \geq p$, a contradiction.

Constructing a gcd-free Basis

Not only is there "unique factorization" as a product of elements of B — we can even find this factorization efficiently:

Theorem. Suppose $A = (a_1, \ldots, a_m)$ is a list of integers, each ≥ 2, and $B = \{b_1, \ldots, b_n\}$ is a gcd-free basis for A. Then we can compute the mn non-negative integers e_{ij} such that

$$a_i = \prod_{1 \leq j \leq n} b_j^{e_{ij}}$$

using $O((\lg c)^2 + (\lg c)(\lg d))$ bit operations, where $c = a_1 a_2 \cdots a_m$, and $d = b_1 b_2 \cdots b_n$.

Proof. We compute the e_{ij} by trial division: for each a_i, we attempt to remove a factor of b_j as many times as possible. We use the following algorithm:

Constructing a gcd-free Basis

BASIS-FACTOR(A, B)
(1) for $i \leftarrow 1$ to m do
(2) $\quad t \leftarrow a_i$
(3) $\quad$ for $j \leftarrow 1$ to n do
(4) $\quad\quad e_{ij} \leftarrow 0$
(5) $\quad\quad$ while $b_j \mid t$ do
(6) $\quad\quad\quad t \leftarrow t/b_j$
(7) $\quad\quad\quad e_{ij} \leftarrow e_{ij} + 1$
(8) return(e)

To determine the running-time of BASIS-FACTOR, we compute the cost for all the divisions in lines (5) and (6), which majorize the cost for the other lines.

The cost of lines (5) and (6) for each pair (a_i, b_j) is, from our definition of naive bit complexity,

$$O((e_{ij} + 1)(\lg a_i)(\lg b_j)).$$

Hence the total cost associated with lines (5) and (6) is, ignoring the constant implicit in the $O()$,

$$\begin{aligned}
&\sum_{i,j} \quad (e_{ij}+1)(\lg a_i)(\lg b_j) \\
&\quad \leq \sum_{i,j}(e_{ij}+1)(1+\log_2 a_i)(1+\log_2 b_j) \\
&\quad \leq 4\sum_{i,j}(e_{ij}+1)(\log_2 a_i)(\log_2 b_j) \\
&\quad = 4\sum_{i,j} e_{ij}(\log_2 a_i)(\log_2 b_j) + 4\sum_{i,j}(\log_2 a_i)(\log_2 b_j) \\
&\quad = 4\sum_{i,j}(\log_2 a_i)(\log_2 b_j^{e_{ij}}) + 4\sum_{i,j}(\log_2 a_i)(\log_2 b_j) \\
&\quad = 4\sum_{i}(\log_2 a_i)\sum_{j}\log_2 b_j^{e_{ij}} + 4\sum_{j}(\log_2 b_j)\sum_{i}\log_2 a_i \\
&\quad = 4\sum_{i}(\log_2 a_i)^2 + 4(\log_2 m)\sum_{j}\log_2 b_j \\
&\quad \leq 4(\lg c)^2 + 4(\lg c)(\lg d).
\end{aligned}$$

This completes the analysis of BASIS-FACTOR.

■

Constructing a gcd-free Basis

Our next task is to show how to efficiently compute a gcd-free basis for a list of integers $A = (a_1, a_2, \ldots, a_m)$. First, we discuss the case where $m = 2$.

Consider the following algorithm, which "refines" a partial factorization of $n = ab$ into relatively prime pieces:

$\textsc{Refine}(a, b)$
(1) $g \leftarrow \gcd(a, b)$
(2) if $g = 1$ then return$(a, \emptyset, b)$
(3) else $(\ell_1, S_1, r_1) \leftarrow \textsc{Refine}(a/g, g)$
(4) $(\ell_2, S_2, r_2) \leftarrow \textsc{Refine}(r_1, b/g)$
(5) return$(\ell_1, S_1 \cup S_2 \cup \{\ell_2\}, r_2)$

We first prove that this algorithm always terminates:

Theorem. The algorithm REFINE always terminates, and on input $(a, b) \neq (1, 1)$, it makes at most $2 \log_2 ab - 2$ additional recursive calls to itself.

Proof. By induction on ab. Clearly the algorithm halts on input $(1, 1)$. For the rest of the proof, we assume $ab \geq 2$. If $ab = 2$, then the input is $(1, 2)$ or $(2, 1)$, and no recursive calls are made.

Now assume the result is true for all a, b with $1 < ab < c$; we will prove the result for $ab = c$.

If the input is $(c, 1)$ or $(1, c)$, the result clearly holds. Otherwise, $a, b > 1$. Let $R(c)$ denote the maximum number of recursive calls made on input (a, b), as a, b range over all positive integers such that $c = ab$; a priori $R(c)$ could be infinite. Then we have

$$R(c) \leq 2 + \max_{\substack{ab=c \\ a,b \neq 1}} (R(a) + R(b)).$$

By induction, all quantities appearing in the sum are finite, so $R(c)$ is finite. Also, we have

$$\begin{aligned} R(c) &\leq 2 + \max_{\substack{ab=c \\ a,b\neq 1}} (R(a) + R(b)) \\ &\leq 2 + \max_{\substack{ab=c \\ a,b\neq 1}} (2\log_2 a + 2\log_2 b - 4) \\ &\leq 2\log_2 c - 2. \end{aligned}$$

This completes the proof. ∎

Theorem. If, on input (a, b), the algorithm REFINE returns an output (ℓ, S, r), then

(i) $\ell \mid a$ and $r \mid b$;

(ii) each element of S divides $\gcd(a, b)$;

(iii) the set $(S \cup \{\ell, r\}) - \{1\}$ is a gcd-free basis for (a, b); and

(iv) $\gcd(\ell, b) = \gcd(r, a) = 1$.

Proof.

(i): We show that $\ell \mid a$. If the algorithm terminates on line 2 without any recursive calls, this is clear. Otherwise by induction, $\ell \mid a/g$ in line 4, and hence $\ell \mid a$. In a similar fashion, $r \mid b$.

(ii): We show that each element of S divides $\gcd(a, b)$. If the algorithm terminates on line 2, this is trivially true. Otherwise, by induction, each element of S_1 divides $g = \gcd(a, b)$. From the result in the previous paragraph, $r_1 \mid g$, and ℓ_2 and each element of S_2 divides r_1. Therefore each element of $S = S_1 \cup S_2 \cup \{\ell_2\}$ divides g.

(iii): In order to show that

$$S' = (S \cup \{\ell, r\}) - \{1\}$$

is a gcd-free basis for (a, b), we first show that $\ell \perp r$. This is clear if the algorithm terminates on line 2. Otherwise, by a result above, $\ell_1 \mid a/g$, $r_2 \mid b/g$, and therefore

$$\gcd(\ell, r) = \gcd(\ell_1, r_2) \mid \gcd(a/g, b/g) = 1.$$

We now introduce a useful notational convention: if q is an integer and S is a set, by $q \perp S$ we mean q is relatively prime to each member of S. Using this convention, we now show $\ell \perp S$. This is clearly true if the algorithm terminates on line

2, for then $S = \emptyset$. Otherwise, by induction we have $\ell_1 \perp S_1$. Also by induction, all elements of S_2 must divide b/g, but $\ell_1 \,|\, a/g$, so $\ell_1 \perp S_2$. By the result above, $\ell_1 \perp r_1$, and $\ell_2 \,|\, r_1$, so $\ell_1 \perp \ell_2$. In a similar fashion, we can show $r \perp S$.

Next, we show that the elements of S are pairwise relatively prime. By induction, the elements of S_1 are pairwise relatively prime, and so are the elements of S_2. Let $s_1 \in S_1$, and $s_2 \in S_2$; then $s_1 \,|\, a/g$ and $s_2 \,|\, b/g$. Therefore $s_1 \perp s_2$, since $\gcd(a/g, b/g) = 1$. Next, $\ell_2 \perp S_1$, since $\ell_2 \,|\, r_1$, and $r_1 \perp S_1$ by induction. Finally, $\ell_2 \perp S_2$, by induction.

Now we show that a and b can be expressed as the product of (powers of) elements of $S \cup \{\ell, r\}$. If the algorithm terminates at line 2, this is clear. Otherwise, $a = (a/g)g$, and by induction, each of these can be expressed as the product of elements of $S_1 \cup \{\ell_1, r_1\}$; also, r_1 can be expressed as the product of elements of $S_2 \cup \{\ell_2, r_2\}$. A

similar proof applies to b. It follows that $(S \cup \{\ell, r\}) - \{1\}$ is a gcd-free basis for $A = (a, b)$.

Our next task is to show that $\ell \perp b$. If the algorithm terminates on line 2, this is clearly true. Otherwise, by a result above, $\ell_1 \mid a/g$. Now $\gcd(\ell_1, b) \mid \gcd(a/g, b)$, and $\gcd(a/g, b) \mid g$. Therefore $\gcd(\ell_1, b) \mid g$. But in line 3, $\ell_1 \perp g$ by induction. Hence $\ell_1 \perp b$.

Similarly, we can show $r \perp a$. Again, this is trivially true if the algorithm terminates on line 2. Otherwise, $r_2 \perp r_1$ by induction, and $r_2 \perp \ell_1$ and $r_2 \perp S_1$ by the results above. Since $(\ell_1, S_1, r_1) = \textsc{Refine}(a/g, g)$, we know that a/g and g can be expressed as the product of elements of $S_1 \cup \{\ell_1, r_1\}$. Therefore $r_2 \perp a/g$ and $r_2 \perp g$. It follows that $r_2 \perp a$.

This completes the proof of the properties of the output of the algorithm. ∎

Constructing a gcd-free Basis

We now show that the algorithm REFINE is surprisingly efficient:

Theorem. On input (a, b), the algorithm REFINE uses $O((\lg ab)^2)$ bit operations.

Proof. We have seen the cost for computing $\gcd(a, b)$, a/g, and b/g, measured in bit operations, is $O((\lg a)(\lg b))$. Provided $a \neq 1$ and $b \neq 1$, this cost is bounded by a constant times $2(\log a)(\log b)$, and using this figure greatly simplifies the analysis. It does not, however, correctly handle the case where $a = 1$ or $b = 1$. Thus, we need to add the cost of computing gcd's when one or both of the arguments to REFINE is 1. However, as we have seen, no more than $2 \log_2 ab - 2$ recursive calls are made to REFINE on input $(a, b) \neq (1, 1)$, and it is easy to see that no recursive calls are made when the input is $(1, 1)$. Hence the total cost of handling inputs where one factor is 1 is $O((\log ab)^2)$.

To obtain the desired time bound, then, we are free to charge $2(\log a)(\log b)$ for the cost of computing $\gcd(a, b)$, a/g, and b/g. Let $T(c)$ be the cost for computing a gcd-free basis (not including the cost of dealing with inputs containing 1) for (a, b), where $c = ab$. We claim that $T(c) \leq (\log c)^2$, and we prove this inequality by induction on c. It is clearly true when $c = 1$. For $c > 1$, we have

$$\begin{aligned} T(c) &\leq T(a/g \cdot g) + T(r_1 \cdot b/g) + 2(\log a)(\log b) \\ &\leq T(a) + T(b) + 2(\log a)(\log b) \\ &\leq (\log a)^2 + (\log b)^2 + 2(\log a)(\log b) \\ &= (\log a + \log b)^2 \\ &= (\log c)^2, \end{aligned}$$

and the result is true by induction. ∎

Open Problem

Find an efficient parallel algorithm to compute $\gcd(u, v)$. It should use $O((\lg \lg u)^c)$ time employing $O((\lg u)^d)$ processors. Or prove that no such algorithm exists.