Egyptian Fractions

Kevin Gong

UC Berkeley
Math 196 Spring 1992
Faculty Advisor:
Professor Andrew Ogg

Acknowledgments

Thanks to my faculty advisor, Professor Andrew Ogg.

Thanks to Paul Campbell for a biography of sources on Egyptian fractions.

Contents

1 Introduction

Ancient Egyptian hieroglyphics tell us much about the people of ancient Egypt, including how they did mathematics. The Rhind Mathematical Papyrus, the oldest existing mathematical manuscript, tells us that their basic number system is very similar to ours except in one way – their concept of fractions.

The ancient Egyptians had a way of writing numbers to at least 1 million. However, their method of writing fractions was limited. To represent the fraction 1/5, they would simply use the symbol for 5, and place another symbol on top of it. In general, the reciprocal of an integer n was written in the same way. They had no other way of writing fractions, except for a special symbol for 2/3 and perhaps 3/4. [Gil72] This is not to say that the number 5/6 did not exist in ancient Egypt. They simply had no way of writing it as a single symbol. Instead, they would write 1/2 + 1/3.

Thus, *Egyptian fractions* is a term which now refers to any expression of a rational number as a sum of distinct unit fractions (a unit fraction is a reciprocal of a positive integer). The study of the properties of Egyptian fractions falls into the area of number theory, and provides many challenging unsolved problems.

In this paper we will examine some of the problems concerning Egyptian fractions which have inspired research from the days of Fibonacci to the present.

2 Introduction to Construction Algorithms

One basic problem concerning Egyptian fractions is the search for construction algorithms – ways to write any fraction as the sum of unit fractions. Over the years, many different algorithms have been formulated for varying purposes. They range from the purely theoretical to the practical and everywhere in between.

It is immediately evident that any rational has more than one distinct Egyptian fraction expansion. If $\frac{a}{b} = \frac{1}{x_1} + \ldots + \frac{1}{x_n}$, then the equation

$$\frac{1}{x} = \frac{1}{x+1} + \frac{1}{x(x+1)}$$

can be used to obtain $\frac{a}{b} = \frac{1}{x_1} + \ldots + \frac{1}{x_{n-1}} + \frac{1}{x_n+1} + \frac{1}{x_n(x_n+1)}$.

The Egyptians themselves may or may not have had a single algorithm to construct fraction expansions. They created a table of expansions of the numbers $2/n$ for all odd numbers $n < 100$ (see appendix B). Gill discusses some different criteria which the Egyptians may have used to create the table. [Gil72]

In this paper, we will only concern ourselves with rationals of the form $p/q < 1$. So, wherever it is not explicitly stated, assume that this is the case. It is possible to express improper rationals as the sum of unit fractions, but we will not discuss this (except briefly, in Appendix A).

Before we begin, some brief words on notation which will be useful.

Notation

We will describe and compare several different algorithms and evaluate their performance. In doing so, we will use the following notation:

Suppose $p/q = 1/n_1 + \ldots + 1/n_k$ with $n_1 < n_2 < \ldots < n_k$

Then define:

$D(p,q)$ = minimal possible value of n_k
$D(q)$ = $\max \{D(p,q) \mid 0 < p < q\}$
$L(p,q)$ = minimum possible value for k
$L(q)$ = $\max \{L(p,q) \mid 0 < p < q\}$

For convenience, we will also define:

P_i = *i*th prime number, where $P_1 = 2$ ($P_2 = 3$, $P_3 = 5$, etc.)
Π_k = $P_1 \cdot P_2 \cdots P_k$
S = $\{P^{2^k} \mid k \geq 0 \text{ and } P \text{ is a prime}\}$
s_i = *i*th smallest element of S

In many of the papers on Egyptian fractions, $\log_2 n$ is used as shorthand for log log n. However, we will not use that convention here. We will write out log log n. Thus, when we write $\log_2 n$, we mean the logarithm base 2.

Splitting Method

Probably the worst algorithm for creating a unit fraction expansion is the splitting method. It is based on repeated use of the equality

$$\frac{1}{x} = \frac{1}{x+1} + \frac{1}{x(x+1)}$$

which is known as the "splitting relation."

Given: rational $p/q < 1$ in lowest terms
Step 1: Write p/q as the sum of p unit fractions $1/q$
Step 2: If there are duplicated fractions $1/a$ in the expansion (for any integer a), keep one of them, but remove the other duplicated $(1/a)$'s by applying the splitting relation to them.
Step 3: Repeat Step 2 until an expansion is reached which has no denominator duplicated.

An example: 3/7

$$\begin{aligned}
\frac{3}{7} &= \frac{1}{7} + \frac{1}{7} + \frac{1}{7} \\
&= \frac{1}{7} + \left(\frac{1}{8} + \frac{1}{56}\right) + \left(\frac{1}{8} + \frac{1}{56}\right) \\
&= \frac{1}{7} + \frac{1}{8} + \frac{1}{8} + \frac{1}{56} + \frac{1}{56} \\
&= \frac{1}{7} + \frac{1}{8} + \left(\frac{1}{9} + \frac{1}{72}\right) + \frac{1}{56} + \left(\frac{1}{57} + \frac{1}{3192}\right) \\
&= \frac{1}{7} + \frac{1}{8} + \frac{1}{9} + \frac{1}{56} + \frac{1}{57} + \frac{1}{72} + \frac{1}{3192}
\end{aligned}$$

Campbell [Cam77] proves that this method always works. The difficulty is in proving that this method will eventually terminate. The techniques used to prove this are beyond the scope of this paper.

The algorithm itself produces expansions which are generally the worst (of those algorithms presented here). It is unclear if there are bounds for either L() or D() using this method.

Fibonacci-Sylvester Algorithm

A much more intuitive, useful algorithm is the Fibonacci-Sylvester algorithm. It was first discovered by Fibonacci‡ in 1202 [Dun66], and later by Sylvester [Syl880]. The algorithm is a straightforward, greedy algorithm. At each step, we simply take the largest unit fraction less than whatever is left. Fibonacci used it (he preferred working with unit fractions), but did not prove that it worked. It was not until 1880 that Sylvester proved its correctness.

‡ Actually, Fibonacci describes an algorithm with 7 different cases, the last of which is a default case which is exactly the greedy algorithm.

Given: rational $p/q < 1$ in lowest terms
Step 1: assign p' = p and q' = q
Step 2: If p' = 1, let p' / q' be part of the expansion, and we are done.
Otherwise, use the division algorithm to obtain q' = sp' + r, where r < p'
Step 3: Note that $\frac{p'}{q'} = \frac{1}{s+1} + \frac{p'-r}{q'(s+1)}$
So let $\frac{1}{s+1}$ be part of the expansion.
Step 4: Let p' = p' - r and q' = q'(s+1)
Step 5: Reduce p' / q' to lowest terms and go back to step 2

Example: $\frac{3}{7} = \frac{1}{3} + \frac{2}{21}$
$= \frac{1}{3} + \frac{1}{11} + \frac{1}{231}$

Theorem

The Fibonacci-Sylvester algorithm is guaranteed to produce an expansion with p *or fewer terms.*

proof

The algorithm produces:

$$\frac{p'}{q'} = \frac{1}{s+1} + \frac{p' - r}{q'(s+1)}$$

Intuitively, we know that the algorithm produces at most p terms because the numerators always get smaller. More formally:

Since p' / q' is in lowest terms, we know that r > 0.

At step 4, we have p' = p' - r, so the new p' ≤ old p' - 1

At step 2, we stop if p' = 1, so there can be at most p terms.

Thus, the worst case is where r = 1 each time, and the resulting fraction is always in lowest terms. Then the expansion clearly produces p terms. Δ

In practice, this worst case is seldom reached. On the other hand, the problem with this method is that the denominators can grow quite huge. For example, the Fibonacci-Sylvester algorithm expands 5 / 121 as:

$$\frac{5}{121} = \frac{1}{25} + \frac{1}{757} + \frac{1}{763309} + \frac{1}{873960180912} + \frac{1}{1527612795642093418846225}$$

Compare this with the optimal solution,

$$\frac{5}{121} = \frac{1}{33} + \frac{1}{121} + \frac{1}{363}$$

Fibonacci himself recognized this shortcoming, noting that

$$\frac{4}{49} = \frac{1}{13} + \frac{1}{319} + \frac{1}{319(637)}$$

but

$$\frac{4}{49} = \frac{1}{14} + \frac{1}{98}$$

and suggesting that one should try a smaller first fraction if the first attempt does not produce an "elegant" solution. He does not define what elegant is, and this then becomes less an algorithm and more trial-and-error.

Mays [May87] examines the worst case of the algorithm – cases in which the expansion requires *a* terms. The smallest fractions fitting this category are as follows:

terms	a/b
1	1/2
2	2/3
3	3/7
4	4/17
5	5/31
6	6/109
7	7/253
8	8/97
9	9/271
10	10/1621
11	11/199

Mays finds a set of congruences which the b's must satisfy for the expansion to be worst.

Golomb's Algorithm

Golomb [Gol62] describes a simple algorithm which can be used to represent a rational *p/q* as the sum of *p* or fewer unit fractions. The algorithm works as follows:

Given: rational $p/q < 1$ in lowest terms
Step 1: Let p' = p and q' = q
Step 2: If p' = 1, let p' / q' be part of the expansion, and we are done.
Step 3: Let p'' be such that p'p'' = q'r + 1, 0 < p'' < q'
(p'' is the multiplicative inverse of p' modulo q')
Step 4: Note $\frac{p'}{q'} = \frac{1}{p''q'} + \frac{r}{p''}$
So let $\frac{1}{p''q'}$ be part of the expansion.
Step 5: Let q' = p'' and p' = r and go back to step 2

Example: $\frac{3}{7} = \frac{1}{3} + \frac{2}{21}$

$= \frac{1}{3} + \frac{1}{15} + \frac{1}{35}$

This algorithm is better than the Fibonacci-Sylvester algorithm in the sense that the denominators are guaranteed to be at most q(q-1). The denominators may sometimes be better for the Fibonacci-Sylvester algorithm, but there is no such bound for the denominators and, as seen above, in fact can grow quite large.

Theorem

$D(q) < q(q-1)$ *for Golomb Algorithm*

proof

Note that at step 3, $p'' < q'$, so $p'' \leq q'-1$.
The denominator at step 4 is $p''q'$, so the denominator is $\leq (q'-1)q'$.
Note that in step 5, we let new $q' = p'' <$ old q', so the q' is always decreasing.
Thus, the denominators cannot be larger than $q(q-1)$. Δ

3 Practical Numbers

It is easily seen that if p can be written as the sum of divisors of q, then p/q can be expanded with no denominator greater than q itself. For example, if we want to expand 9/20, note that 4 and 5 are divisors of 20, so

$$\frac{9}{20} = \frac{4+5}{20} = \frac{1}{5} + \frac{1}{4}$$

In fact, Webb [Web75] proves a theorem by Rav (1966):

Theorem

$m/n = 1/x_1 + 1/x_2 + ... + 1/x_k$ if and only if there exist positive integers M and N and divisors $D_1, ..., D_k$ of N such that $M/N = m/n$ and $D_1 + D_2 + ... + D_k = 0 \pmod{M}$. Also, the last condition can be replaced by $D_1+D_2+...+D_k = M$; and the condition $(D_1, D_2, ..., D_k) = 1$ may be added without affecting the validity of the theorem.

In section 6, we will go through the proof of this theorem.

All of this brings us to what are called *practical* numbers. Srinivasan [Sri48] first defined practical numbers in 1948. They were also referred to as *panarithmic* numbers in [Rob79] and [Hey80].

Definition

A practical number is an integer N such that for all $n < N$, n can be written as the sum of distinct divisors of N.

For example, 4 is practical, since 1 = 1, 2 = 2, and 3 = 1 + 2. On the other hand, 10 is not, since 4 cannot be written as the sum of 1, 2, 5, and 10.

Relating to Egyptian fractions, the most important property of practical numbers, proved in [Rob79], is:

Theorem

If n is a practical number and q is any number relatively prime with n, and $q < 2n$, then qn is also practical.

So, if we want to expand 5/23, we can note that 12 is practical and thus:

$$\frac{5}{23} = \frac{5(12)}{23(12)}$$

Since 23 < 2(12) and 12 is practical, we know that 23(12) is also practical. So 5(12) can be written as the sum of distinct divisors of 23(12). In fact:

$$\frac{5(12)}{23(12)} = \frac{46 + 12 + 2}{23(12)} = \frac{1}{6} + \frac{1}{23} + \frac{1}{138}$$

Fibonacci almost strikes here again, as he suggested finding a number "which has in it many divisors like 12, 24, 36, 48, 60, ..." to multiply by. He fails to present an algorithm based upon this approach, however (or define which numbers have "many" divisors).

This type of computation is the basis for several different construction algorithms, sometimes known as multiplication algorithms – since the expansion is obtained by multiplying numerator and denominator by the same number. We will first create such an algorithm and prove things about it, then describe some of the most recent algorithms created.

Before we begin, some properties of practical numbers and references to their proofs:

If n has divisors $1 = d_1 < d_2 < ... < d_c = n$ then n is practical if and only if

$$\sum_{i=1}^{r} d_r \geq d_{r+1} - 1 \text{ for all } r < c-1 \qquad \text{[Rob79]}$$

The above fact is used in the computer programs to test for practicality.

If n has a subset of divisors $1 = d_1, d_2, ..., d_c = n$ in which each is at most twice the previous divisor, then n is practical. [Rob79]

If n is practical and m is a natural number $\leq$ n then mn is practical. $m^k n^l$ is also practical. [Hey80]

If n is practical and the sum of the divisors of n is at least n+k where k is a non-negative integer, then n(2n+k+1) is practical. [Hey80]

Binary Algorithm

We note that if $N = 2^n$ then any $m < N$ can be written as the sum of distinct divisors of N. We simply write the number in binary notation. In fact, m can be written as the sum of n or less divisors, since 2^n has exactly n divisors – $2^0, 2^1, 2^2, ..., 2^{n-1}$.

For example:

$$\frac{5}{16} = \frac{1+4}{16} = \frac{1}{16} + \frac{1}{4}$$

Given: rational $p/q < 1$ in lowest terms

Step 1: Find $N_{k-1} < q \leq N_k$ where $N_k = 2^k$

Step 2: If $q = N_k$ then simply write out p as the sum of k or less divisors of N_k: $p = \sum_{i=1}^{j} d_i$, and get the expansion

$$\frac{p}{q} = \sum_{i=1}^{j} \frac{d_i}{N_k} = \sum_{i=1}^{j} \frac{1}{N_k / d_i}$$

Otherwise, go to step 3.

Step 3: Note that for some integers s and r, where $0 < r < N_k$ we have:

$$\frac{p}{q} = \frac{pN_k}{qN_k} = \frac{qs+r}{qN_k} = \frac{s}{N_k} + \frac{r}{qN_k}$$

Step 4: Write $s = \sum d_i$ where d_i = distinct divisors of N_k

Write $r = \sum d_i'$ where d_i' = distinct divisors of N_k

Step 5: Thus get the expansion $\sum 1/(N_k/d_i) + \sum 1/(qN_k/d_i')$

For example:

5/21: $16 < 21 < 32$

$$\frac{5}{21} = \frac{5(32)}{21(32)}$$
$$= \frac{7(21) + 13}{21(32)}$$
$$= \frac{7}{32} + \frac{13}{21(32)}$$
$$= \frac{1+2+4}{32} + \frac{1+4+8}{21(32)}$$
$$= \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{84} + \frac{1}{168} + \frac{1}{672}$$

<u>Theorem</u>

The Binary Algorithm is guaranteed to produce an expansion with $D(n) < n^2$ *and* $L(n) = O(\log n)$.

<u>proof</u>

First, we prove the algorithm works in the first place.

In step 2, note that $p < q < N_k$ so $pN_k < qN_k$

$$qs + r = pN_k < qN_k$$

so $s < N_k$.

Thus, we can always find an expansion for both s and r. The resulting denominators of the expansion are distinct because q divides the second set of denominators (corresponding to r). It cannot divide the denominators corresponding to s unless q is a power of 2. But if it were, we never would have gotten past step 2. So the algorithm at least works.

In the case where $q = N_k$, the expansion clearly has at most k terms. In the case where $q < N_k$, the expansion has at most 2k terms. Since $k = \log_2 N_k$, it follows that there are at most 2log q terms in the expansion. Thus, $L(n) = O(\log n)$.

In the case where q = Nk, the largest denominator is clearly q. In the case where $q < N_k$, the largest denominator can be $q \cdot N_k$, so the largest denominator must be at most q(q-1). Thus, $D(n) = O(n^2)$. Δ

Bleicher/Erdös Algorithm

Note that while 2^n is a simple number, it is not the best choice for a practical number. Numbers of the form 2^n are the practical numbers with the fewest number of divisors. This causes the bound for the number of terms in an expansion to be log q. Clearly, if our practical number has more divisors, a numerator might be written as the sum of fewer divisors, thus lowering the bound for the number of terms. To increase the number of divisors, we can avoid duplicating factors in our practical number. Bleicher and Erdös take this approach in their algorithm of 1976 [Ble76a], where they define $N_k = \Pi_k$.

Given: rational $p/q < 1$ in lowest terms
Step 1: Find k such that $N_{k-1} < q \le N_k$
Step 2: If $q \mid N_k$ then $p/q = b/N_k$ and we can write $b = \sum d_i$ where all $d_i \mid N_k$
Step 3: If not, then $p/q = pN_k/qN_k = (sq + r)/qN_k = s/N_k + r/qN_k$ where we make the restriction

$$N_k(1-1/k) \le r \le N_k(2-1/k)$$

The term s/N_k can be done as with b/N_k
We find an expansion for r and multiply the denominators by q.

An example:
5/121: Thus, $k = 4$ and $N_k = 2 \cdot 3 \cdot 5 \cdot 7$

$$\frac{5}{121} = \frac{(2 \cdot 3 \cdot 5 \cdot 7) \cdot 5}{(2 \cdot 3 \cdot 5 \cdot 7) \cdot 121}$$

Note $N_k(1-1/k) = 315/2 = 157.5$
and $N_k(2-1/k) = 735/2 = 367.5$
Noting $5 \cdot (2 \cdot 3 \cdot 5 \cdot 7)/121$ = about 8.7, we let $q = 7$ and
$aN_k = 7 \cdot 121 + 203$

$$\text{Thus } \frac{5}{121} = \frac{7}{2 \cdot 3 \cdot 5 \cdot 7} + \frac{203}{(2 \cdot 3 \cdot 5 \cdot 7) \cdot 121}$$
$$= \frac{1}{30} + \frac{29}{(2 \cdot 3 \cdot 5) \cdot 121}$$
$$= \frac{1}{30} + \frac{3 + 5 + 6 + 15}{(2 \cdot 3 \cdot 5) \cdot 121}$$
$$= \frac{1}{30} + \frac{1}{1210} + \frac{1}{726} + \frac{1}{605} + \frac{1}{242}$$
$$= \frac{1}{30} + \frac{1}{242} + \frac{1}{605} + \frac{1}{726} + \frac{1}{1210}$$

Theorem

For the Bleicher/Erdös algorithm, $D(N) = O(N(\log N)^3)$

proof

We can easily prove by induction, using the first theorem on practical numbers, that the $\prod_k$ are practical. However, Bleicher and Erdös prove an even stronger statement:

Lemma 1

Any positive integer $n \le \sigma(\prod_k)$ can be written as the sum of distinct divisors of $\prod_k$. Here, $\sigma(n)$ denotes the sum of divisors of n, and it is obvious that $\sigma(n) > n$.

The proof is by induction on k.

i) The lemma is easily shown to be true for k = 0, 1, 2. For example, for k = 2, we have $\prod_k = 6$ and $\sigma(\prod_k) = 1 + 2 + 3 + 6 = 12$. So note that 4 = 1 + 3, 5 = 2 + 3, 7 = 6 + 1, 8 = 6 + 2, 9 = 6 + 3, 10 = 6 + 3 + 1, and 11 = 6 + 3 + 2.

ii) Suppose the lemma is true for 0, 1, 2, ..., k-1. If $n \le \sigma(\prod_{k-1})$ we are clearly done. So assume $\sigma(\prod_{k-1}) < n \le \sigma(\prod_k)$. Note that:

$$\sigma(\prod_k) = \sigma(\prod_{k-1}) \times (P_k + 1)$$
$$\sigma(\prod_k) - \sigma(\prod_{k-1}) = P_k \times \sigma(\prod_{k-1})$$

Therefore,

$$n - \sigma(\prod_{k-1}) \le \sigma(\prod_k) - \sigma(\prod_{k-1}) = P_k \times \sigma(\prod_{k-1})$$
$$n - P_k \times \sigma(\prod_{k-1}) \le \sigma(\prod_{k-1})$$

And for $k \ge 3$, we have

$$n > \sigma(\prod_{k-1}) \ge 2P_{k-1} > P_k$$

So we can find an integer s such that

$$0 < n\text{-}sP_k \le \sigma(\prod_{k-1})$$
and
$$0 < s \le \sigma(\prod_{k-1})$$

So, since the lemma is true for k-1, we can write

$$s = \sum d_i'$$
and
$$n\text{-}sP_k = \sum d_i$$

where d_i and d_i' are divisors of $\prod_{k-1}$ and the d_i' are distinct and the d_i are distinct. But then, since $P_k \nmid \prod_{k-1}$ we have that:

$$n = \sum(P_k d_i') + \sum d_i$$

is the desired representation of n. ◊

Lemma 2

Let P be a prime and k an integer with $0 \le k < P$. Given any k integers $\{x_1, x_2, ..., x_k\}$ none of which is divisible by P, then the 2^k sums of subsets of $\{x_1, x_2, ..., x_k\}$ lie in at least k+1 distinct congruence classes mod P.

Again, Bleicher and Erdös use induction on k.

i) If k = 0, then there is just one sum – 0. So it is true for k = 0.

ii) Suppose it is true for k-1. Then let n = the number of distinct congruence classes mod P resulting from the sums of subsets of $\{x_1, x_2, ..., x_{k-1}\}$. We know $n \ge (k-1)+1 = k$. If $n > k$, we are done. So assume n = k. Now calculate the sums resulting from adding x_k to all the sums. If a new congruence class is obtained, we are done. If not, then observe that if we let $x_{k+P} = ... = x_{k+2} = x_{k+1} = x_k$, and if we add them one at a time, we still have just k congruence classes. But this is impossible, since P is a prime and P does not divide x_k. ◊

Lemma 3

If r is any integer satisfying $N_k(1-1/k) \le r \le N_k(2-1/k)$ then there are distinct divisors d_i of N_k such that

1. $r = \sum d_i$
2. $d_i \ge cN_{k-3}$ for some constant c

The proof of this lemma is lengthy and complicated and will not be shown, but involves induction on k and the use of lemma 2. ◊

Lemma 4

If $N_{k-1} \le N \le N_k$ then

$$k \le \frac{\ln N}{\ln \ln N}\left(1 + \frac{\ln \ln \ln N}{\ln \ln N}\right)$$

Again, we omit the proof. ◊

Now, we prove the original theorem.

If $q \mid N_k$, then it is clear that no denominator is greater than N_k.

If not, then in step 3 the denominators associated with the s / N_k term are clearly also no greater than N_k, and we are clearly done.

For the r / qN_k term, note that lemma 3 allows us to write $r = \sum d_i$ with all the $d_i \ge cN_{k-3}$. So the denominators are at most $qN_k / cN_{k-3} = qP_kP_{k-1}P_{k-2} / c$. By lemma 4, this is $\le q(\ln q)^3 / c = O(q(\log q)^3)$. Δ

In 1986, Yokota [Yok86b] modified the algorithm slightly, letting r be such that:

$$(1-2/\sqrt{P_k})\prod_k \le r < 2\prod_k$$

Using this modified algorithm, Yokota proves that:

$$L(N) \le \frac{4\log N}{\log \log N}\left(1 + \frac{\log \log \log N}{\log \log N}\right)$$

and

$$D(N) \le \lambda N(\log N)^2$$

where $\lambda \to 1$ as $n \to \infty$.

The proof is long and is omitted.

To find out how many terms there are, we must find out how many terms we need to write $N < \prod_k$ as the sum of divisors of $\prod_k$.

To that end, before continuing with the Yokota algorithm, we will first discuss how Goldbach's conjecture might be used in Egyptian fractions.

Goldbach's Conjecture

Here we would like to suggest a possible use for Goldbach's conjecture in Egyptian fractions. It is not immediately obvious that it is helpful, but we will describe how it might be used.

Goldbach's conjecture‡ , formulated in 1742 is that:

Every even integer > 2 is the sum of two primes.

‡ Actually, Goldbach conjectured in a letter to Euler that every integer n > 5 is the sum of three primes. Euler noted the clear equivalence to the stated conjecture.

This has not been proven, but has been shown to be true for 2n up to 10^8 by Stein & Stein in 1965. [Rib89]

Also, Chen has proved that:

<u>Theorem</u>

Every sufficiently large even integer 2n may be written as 2n = p + m where p is a prime and m is the product of two (not necessarily distinct) primes.

Finally, Vinograd proved in 1937 [Rib89] that:

<u>Theorem</u>

There exists n_0 such that every odd $n \geq n_0$ is the sum of 3 primes. $n_0 = 3^{3^{15}}$ works.

So there is some strong evidence that Goldbach's conjecture is true, or at least that we can write a number as the sum of a fixed number of primes.

If we assume Goldbach's conjecture, then suppose we have a number $n < P_k$. If n is odd, we can write it as $n = P_a + P_b$ where a,b < k. If the primes are not distinct and we have $n = P_a + P_a$, then we can write $n = 2P_a = P_1P_a$. If n is even, then we can write n = m + 1 where m is odd and thus $n = P_a + P_b + 1$ or $n = P_1P_a + 1$. So n can be written as the sum of 3 or less divisors of $\prod_k$.

Note also that if we have $n < P_k^2$ then we can write n = sPk + r with s, r < Pk. This means that we can write n as the sum of 6 or less divisors of $\prod_k$.

So perhaps this can be used to create an upper bound on the number of terms required to write $n < \prod_k$ as the sum of distinct divisors of $\prod_k$.

Yokota [Yok86b] proves the very good result that:

<u>Theorem</u>

If $n < \prod_k$ then n can be written as the sum of 2k or fewer divisors of $\prod_k$.

The proof of this theorem is beyond the scope of this paper, involving a theorem about the Mobius function to prove that about half the numbers less than P_k are square-free, and the Cauchy-Davenport Theorem. The proof is done by induction on k.

But even this good result is not perfect.

It turns out that the number of terms required grows very slowly. A computer was used to calculate the number of terms required (see Appendix C),

From those calculations, it looks as if approximately k terms are required, rather than 2k, but it is hard to tell with such little data. It could possibly be asymptotically smaller than k.

Now back to the algorithms.

Yokota Algorithm

The Yokota Algorithm [Yok88a] is another algorithm like the binary algorithm. It defines N_k differently, however, to get very good asymptotic results.

Define: $N_k = \prod_{i=1}^{k} s_i$

Given: rational $a/N < 1$ in lowest terms

Step 1: we find k such that $N_{k-1} \le N < N_k$

Step 2: If $N \mid N_k$ then $a/N = b/N_k$ and we can write $b = \sum d_i$ where all $d_i \mid N_k$

Step 3: If not, then $a/N = aN_k/NN_k = (sN + r)/NN_k = s/N_k + r/NN_k$

where $(1-2/\sqrt{s_k})N_k \le r < 2N_k$ (and $1 \le s < N_k$)

The term s/N_k can be done as with b/N_k

We can find an expansion for r and multiply the denominators by N.

An example:

Note the set S = {2, 3, 4, 5, 7, 9, 11, 13, 16, ...}

So $N_1 = 2$, $N_2 = 6$, $N_3 = 24$, $N_4 = 120$, $N_5 = 840$, etc.

16/17: Thus, k = 3 and $N_k = 2 \cdot 3 \cdot 4 = 24$

$$\frac{16}{17} = \frac{16(24)}{17(24)} = \frac{[22(17)+10]}{17(24)}$$

note $(1-2/\sqrt{s_3})N_3 = 0$, so this is what we want. Continuing:

$$= \frac{22}{24} + \frac{10}{17(24)}$$

$$\frac{22}{24} = \frac{12+8+2}{24} = \frac{1}{2} + \frac{1}{3} + \frac{1}{12}$$

$$\frac{10}{17(24)} = \frac{8+2}{17(24)} = \frac{1}{17(3)} + \frac{1}{17(12)}$$

$$\frac{16}{17} = \frac{1}{2} + \frac{1}{3} + \frac{1}{12} + \frac{1}{51} + \frac{1}{204}$$

This algorithm ensures that

$$L(N) \le \frac{2\log N}{\log\log N}\left(1 + \frac{2\log\log\log\log N}{\log\log\log N}\right)$$

and

$$D(N) \le N(\log N)^{2+\varepsilon}$$

where $\varepsilon \to 0$ as $N \to \infty$.

The proof of the bounds is rather complicated, so we will simply prove that the algorithm works. To do so, it clearly suffices to show:

Theorem

If $N_k = \prod_{i=1}^{k} s_i$ and $r < 2N_k$, then r can be written as the sum of distinct divisors of N_k.

proof

The proof is almost identical to Lemma 1 for the Bleicher / Erdös algorithm.

The proof is by induction on k.

i) The theorem is easily shown to be true for k = 0, 1, or 2. For example, if k = 2, we have $N_k = 6$ and $2N_k = 12$. So note that 4 = 1 + 3, 5 = 2 + 3, 7 = 6 + 1, 8 = 6 + 2, 9 = 6 + 3, 10 = 6 + 3 + 1, and 11 = 6 + 3 + 2.

ii) Suppose the theorem is true for 0, 1, 2, ..., k-1. If $n < 2N_{k-1}$ we are clearly done. So assume $2N_{k-1} \leq n < 2N_k$. Note that:

$$2N_k = 2N_{k-1} \cdot s_k$$

So, find s, r such that

$$n = s \cdot s_k + r$$

with $s_k \leq r < 2s_k$

Clearly,

$$r < 2s_k < 2N_{k-1} \text{ for } k > 2$$

and

$$s \leq 2(N_{k-1} - 1) < 2N_{k-1}$$

So we can write

$$s = \sum d_i'$$

and

$$r = \sum d_i$$

where d_i and d_i' are divisors of N_{k-1} and the d_i' are distinct and the d_i are distinct.
But then, since $s_k \nmid N_{k-1}$ we have that:

$$n = \sum(s_k d_i') + \sum d_i$$

which is the desired representation of n. ◊

Now we will turn to the Tenenbaum / Yokota algorithm, which gives "optimal" asymptotic bounds.

Tenenbaum/Yokota Algorithm

The Tenenbaum / Yokota Algorithm [Ten90] is very similar to the Bleicher / Erdös algorithm. It uses the same definition for N_k and is identical for the s / N_k part. So it is asymptotically the same as the Bleicher / Erdös algorithm in the number of terms. However, its handling of the r / NN_k part is different, yielding a solution with asymptotically smaller denominators.

Define:	$N_k = \prod_k$
Given:	rational $a / N < 1$ in lowest terms

Step 1: we find k such that $N_{k-1} \leq N < N_k$
Step 2: If $N \mid N_k$ then $a/N = b/N_k$ and we can write $b = \sum d_i$ where all $d_i \mid N_k$
Step 3: If not, then $a/N = aN_k/NN_k = (sN + r)/NN_k = s/N_k + r/NN_k$
where $N_k \leq r < 2N_k$ (and $0 \leq s < N_k$)
The term s/N_k can be done as with b/N_k
We find an expansion for r/N_k as follows:

$$r/Nk = r^*/\prod_{j=1}^{n} s_j \qquad \text{where } s_n = p_k$$

We can thus find an expansion for this fraction, and multiply the denominators by N.

An example:
16/17: Thus, k = 3 and $N_k = 2 \cdot 3 \cdot 5 = 30$

$$\frac{16}{17} = \frac{16(30)}{17(30)} = \frac{[26(17)+38]}{17(30)} = \frac{26}{30} + \frac{38}{17(30)}$$

$$\frac{26}{30} = \frac{15+10+1}{30} = \frac{1}{2} + \frac{1}{3} + \frac{1}{30}$$

Note $s_4 = 5 = p_k$

$$\frac{38}{30} = \frac{152}{2\cdot3\cdot4\cdot5} = \frac{120+30+2}{120} = \frac{1}{1} + \frac{1}{4} + \frac{1}{60}$$

$$\frac{16}{17} = \frac{1}{2} + \frac{1}{3} + \frac{1}{30} + \frac{1}{17} + \frac{1}{68} + \frac{1}{1020}$$

This algorithm ensures that

$$D(N) \leq 4N(\log N)^2 \log\log N$$

and

$$P(N) \leq (1+\varepsilon)(\log N)/(\log\log N).$$

This is **best** in the sense that no algorithm can yield both

$$L(P) \leq c\log P \;/\; \log\log P$$

and

$$D(P) \leq P(\log P)^{1 + 1/c - e}$$

(see section 5 for the proof of this bound).

The proof of the bounds for L(N) and D(N) involve Yokota's earlier results. Again, the proof for the bounds is fairly complicated. The theorem used to prove that Yokota's algorithm works is sufficient to prove that the Tenenbaum / Yokota algorithm also works.

"Optimal" Practical Number Algorithm

The proofs for the bounds of the last three algorithms are very complex. It would be nice to have a simple algorithm which gave good results with a simple proof. To this end, we can attempt to devise a somewhat "optimal" algorithm using practical numbers as follows:

> Given p / q in lowest terms
> Step 1: Set M = 1
> Step 2: If qM is not practical, let M = M+1 and repeat step 2; otherwise:
> Step 3: Note p / q = pM / qM and find an obvious expansion.

Note that in Step 2, we can instead test to see if pM can be written as the sum of distinct divisors of qM. However, in finding asymptotic results, we will have to take the worst case for p – thus, testing for practicality is more general.

Clearly, this algorithm will terminate because, if nothing else, we can increment M until we reach $2^k \geq q$ (the binary algorithm).

The obvious question is, what is the lowest value for M? If we knew asymptotically what the lowest value for M was, then we would have a pretty good asymptotic bound for the largest denominator in the best expansion. It would not necessarily be the best (in terms of D(N)) expansion, unless we describe some way of picking the best divisors.

If we let M(N) = smallest m such that mN is practical, then we can say:

$$D(N) \leq N \cdot M(N)$$

So if we can find a bound for M(N), we can also find an upper bound for D(N). Appendix C shows computer calculations for M(P) for values of P up to 80000. M(N) appears to grow somewhere between O(log N) and O(N). Perhaps there is some way of using the properties of practical numbers to prove some bound for M(N).

In calculating the value of M(P) with the computer, we make use of the following theorem:

Theorem

$M(P_i) \leq M(P_j)$ *for* $i < j$

proof

Suppose $M(P_j) = m$.

In the general case, take a number $n < mP_i < mP_j$

Find r,s such that $n = sP_i + r$ with $0 \leq r < P_i < P_j$

Since $r < P_j$, we can write r as the sum of distinct divisors of m.

$s < (n-r) / Pi < n / Pi < m$

We assume $m < p_j$ (this is clearly true for large enough j).

So we can write s as the sum of distinct divisors of m.

Thus, since m and P_i are relatively prime, we can write n as the sum of distinct divisors of mP_i.

Therefore, $M(P_i) \leq m = M(P_j)$
$M(P_i) \leq M(P_j)$ Δ

4 Other Algorithms

The following are some other algorithms which we will basically just describe. They are listed here to show the broader range of algorithms available.

Factorial Algorithm

The following produces a denominator-minimal expansion [Cam77], but is not the best algorithm in the world since it takes a very large amount of time to run. It also fails to give any useful asymptotic results.

Given: rational $p/q < 1$ in lowest terms
Step 0: Set n to 1
Step 1: Set M to n!
Step 2: Multiply p and q by M
Step 3: List all divisors of qM
Step 4: List all collections of distinct divisors whose some is pM
Step 5: If there is no such collection, increase n by 1 and go back to step 1
Step 6: Among the collections, select one with greatest minimum divisor
Step 7: Use the selected divisors as numerators of fractions with denominator qM
Step 8: Reduce the fractions to lowest terms to create an expansion
Step 9: If n = q(q-1) go to step 10, otherwise increase n by 1 and go to step 1
Step 10: Among the expansions saved at step 8, choose one with smallest denominator

Erdös [Erd50] proves that this produces an expansion with no more than 2n-2 terms, where (n-1)! < b ≤ n!

Farey Series Algorithm

The Farey Series Algorithm uses the Farey Series to produce an expansion of p/q with at most p terms, and no denominator greater than q(q-1). [Bec69]

The Farey Series of order n, F_n, consists of all the reduced fractions a/b with 0 ≤ a ≤ b ≤ n, arranged in increasing order. This series has the property that if a/b and c/d are adjacent fractions in Fn, and a/b < c/d, then c/d - a/b = 1/bd and b ≠ d.

Given: rational $p/q < 1$ in lowest terms
Step 1: assign p' = p and q' = q
Step 2: Find r/s, the fraction adjacent to p'/q' in the Farey series, with r/s < p'/q' If none exists, then add p'/q' to the expansion, and we are done.
Step 3: Note $\frac{p'}{q'} = \frac{1}{q's} + \frac{r}{s}$

So let $\frac{1}{q's}$ be part of the expansion
Step 4: Let p' = r and q' = s and go back to step 2

Example: $\frac{3}{7} = \frac{1}{3} + \frac{2}{21}$
$= \frac{1}{3} + \frac{1}{15} + \frac{1}{35}$

The Farey Series algorithm appears to give the same results as the Golomb Algorithm. Why? In the Farey Series, we have $r/s < p/q$; in fact, $p/q - r/s = 1/qs \Rightarrow ps - qr = 1 \Rightarrow ps = qr + 1$, which is precisely the Golomb algorithm.

Continued Fraction Algorithm

Bleicher [Ble72] describes an algorithm based on the continued fraction of

$$p/q = [0; a_1, a_2, ..., a_n].$$

Using it, he proves $D(N) < N(N-1)$, and $L(N) \leq \min\left(\frac{2(\ln q)^2}{\ln \ln q}, 1+a_2+a_4+...+a_n^*\right)$ where $a_n^* = 2[n/2]$.

Unfortunately, the algorithm is somewhat complicated, and the proof of the bounds is almost 40 pages.

Algorithm Comparison

There are three basic measures of an expansion:

1) the number of terms (length)
2) the maximum denominator
3) the number of characters required to write the expansion. For example, 1/2 + 1/3 can be written using 3 characters: 2,3 (since we know all numerators are 1) This is a combination of length and the size of the denominators.

A computer was used to compare four algorithms: Fibonacci-Sylvester, Golomb, Bleicher/Erdös, and Tenenbaum/Yokota.

The algorithms were compared using all three criteria, on prime denominators from 2 to 2002, and all corresponding numerators (all such proper rationals).

The results are listed in Appendix C.

The sample is probably too small to derive any real conclusions, but some interesting observations can be made.

In the length category, the Golomb algorithm was horrible, while the Fibonacci-Sylvester algorithm seemed asymptotically similar to the other two. On the other hand, the Fibonacci-Sylvester algorithm was better than the other two most of the time -- averaging rougly 35% fewer terms, and as good as or better than them about 95% of the time. The Bleicher/Erdös and Tenenbaum/Yokota algorithms were only the best about 10% of the time. So perhaps the Fibonacci-Sylvester algorithm can produce a better bound for L(N). Because of its erratic nature, not much has been proved about it.

In the denominator category, the Fibonacci-Sylvester algorithm, as expected, performed horribly. The Golomb algorithm, however, did very well compared with the other two. This is probably due to the relatively small denominators, however. The asymptotic bounds for the other two algorithms are known to be much better.

In the overall character category, the Bleicher / Erdös and Tenenbaum / Yokota algorithms are clearly superior. From appearances, it would appear that the Bleicher / Erdös algorithm is slightly better. This may be due to the small sample, or perhaps it simply is better. The *proven* bounds for the Tenenbaum / Yokota algorithm are better, but this does not mean that the actual bounds are.

5 Length and Denominator Bounds

Construction algorithms tells us how to find expansions and what asymptotic results we can achieve. If we want, however, to know what we *can't* achieve, then construction algorithms are of no use.

Denominator Bounds

Bleicher and Erdös [Ble76a] prove the remarkable result that $D(N) = \Omega(N \log N)$. We will try to provide greater detail than Bleicher and Erdös do in their original proof.

Theorem

$D(N) = \Omega(N \log N)$

proof

Let P be a prime.

Let $x_1 < x_2 < \ldots < x_t$ be distinct integers which occur in any unit fraction expansion of $a / P < 1$, where all the x_i are divisible by P.

Define x_i' by: $x_i'P = x_i$.

Clearly, if $x_i' \geq P$, then $x_i \geq P^2$, and we are done (we would have $D(P) \geq P^2$). So assume that $x_i' < P$.

Now, for a given a, write

$$\frac{a}{P} = \frac{1}{x_{i1}} + \frac{1}{x_{i2}} + \ldots + \frac{1}{x_{ij}} + \frac{1}{y_1} + \ldots + \frac{1}{y_k}$$

where only the $P \mid x_{in}$, but $P \nmid y_n$.

Thus:

$$\frac{a}{P} = \left(\frac{1}{P}\right)\left(\frac{1}{x_{i1}'} + \frac{1}{x_{i2}'} + \ldots + \frac{1}{x_{ij}'}\right) + \left(\frac{1}{y_1} + \ldots + \frac{1}{y_k}\right)$$

$$\frac{a}{P} = \left(\frac{1}{P}\right)\left(\frac{b}{c}\right) + \left(\frac{1}{y_1} + \ldots + \frac{1}{y_k}\right)$$

where $c = \prod_{n=1}^{j} x_{in}'$ and $b = \sum_{n=1}^{j} \prod_{m \neq n} x_{im}'$

$$\frac{a}{P} - \left(\frac{1}{P}\right)\left(\frac{b}{c}\right) = \left(\frac{1}{y_1} + \ldots + \frac{1}{y_k}\right)$$

$$\frac{ca - b}{cP} = \left(\frac{1}{y_1} + \ldots + \frac{1}{y_k}\right)$$

Since $P \nmid y_n$, we must have $P \mid (ca - b)$, thus $ca - b \equiv 0 \pmod{P}$.

Since $x_{in}' < P$, we know $c \not\equiv 0 \pmod{P}$, so for every different value of a, there must be different values for b and c to make that congruence true (since P is prime).

Different values for b and c correspond to a different set of $\{x_{i1}', x_{i2}', \ldots, x_{ij}'\}$.

Since these x_{in}' are taken from the set $\{x_1', x_2', ..., x_t'\}$, there are at most 2^t-1 possible values for (b,c) (since we must take at least one from the set).

There are P-1 possible values for a, and thus we need P-1 possible values for (b,c), which means we need

$$2^t - 1 \geq P-1$$
$$2^t \geq P$$
$$t \geq \log_2 P$$

Since the x_i are distinct, so are the x_i'. Therefore,

$$x_t' \geq \log_2 P$$
$$x_t \geq P\log_2 P$$

Thus, $D(P) \geq P\log_2 P$. So $D(N) = \Omega(N \log N)$. Δ

In 1976, Bleicher and Erdös [Ble76a] state:

"There is both theoretical and computational evidence to indicate that D(N) / N is maximum when N is a prime."

In 1986, Yokota [Yok86a] proves this by proving:

Theorem

For every N,

$$\frac{D(N)}{N} \leq \frac{D(P)}{P}$$

for some prime P that divides N.

Actually, Yokota first proves the more general result that $D(MN) \leq MD(N)$, and the theorem follows easily:

If $N = p_1p_2...p_n$ where $p_1 \leq p_2 \leq ... \leq p_n$ then

$$\frac{D(N)}{N} \leq \frac{p_1 D(N / p_1)}{N} = \frac{D(N / p_1)}{N / p_1} \leq ... \leq \frac{D(p_n)}{p_n}$$

Δ

This helps in proving upper bounds, as we only need to examine the case D(P). For example, if we can find the x_i for the above proof, then x_t is a bound for the denominators.

In 1988, Yokota [Yok88b] proves:

Theorem

$D(N) \leq N(\log N)^{1 + \delta(N)}$

where $\delta(N) \to 0$ as $N \to \infty$

Yokota proves this by using $\prod_{i=1}^{t} s_i$ and proving that a certain subset of divisors of that number contains all residues modulo s_t. The proof itself is very detailed and will not be shown. An algorithm, per se, can't really be extracted from the proof because the proof only deals explicitly with D(P). So an algorithm based on the proof would only apply to rationals with prime denominators. That doesn't, however, lessen the result.

In [Ble76b], Bleicher and Erdös show:

Theorem

For a prime P with $\log_{2r} P \geq 1$

$$D(P) \geq (P\log P \log\log P) / \left(\log_{r+1} P \prod_{j=4}^{r+1} \log_j P\right)$$

Only for this result, $\log_x P$ means the xth log of P. Thus, $\log_3 P = \log\log\log P$.

Of course this can be generalized to D(N).

proof

To prove the theorem, they first define:

Definition

S(N) = the number of distinct possible values of $\sum_{k=1}^{n} \varepsilon_k / k$ where $\varepsilon_k = 0$ or 1.

Basically, this means that given the fractions $1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots, \frac{1}{N}$, S(N) is the number of different sums we can get by adding some of those fractions together.

Bleicher and Erdös then prove the following lemma:

Lemma

For $r \geq 1$ and $\log_{2r} N \geq 1$,

$$S(N) \leq \exp\left(\frac{(N\log_r N)}{(\log^2 N \log_2 N)} \prod_{j=1}^{r} \log_j N\right)$$

With this lemma, it isn't too hard to modify the proof that $D(N) = \Omega(N \log N)$ to use this bound for S(N) to prove this theorem. Δ

Length Bounds

Vose [Vos85] proves:

Theorem

$L(N) = O(\sqrt{\log N})$

proof

Lemma

There exists an increasing sequence N_k of positive integers such that any integer $1 < m < N_k$ is the sum of not more than $O(\sqrt{\log N_{k-1}})$ distinct divisors of N_k.

The proof of the lemma is long and complicated. The N_k aren't too complicated – they are defined as:

$$N_k = 4^{\alpha k^2} \prod_{l=2}^{k} p_l^2$$

where $p_2 < p_3 < \ldots$ are odd primes and α is any "sufficiently large integer."
The p_i's are not *i*th primes, however, and must be chosen in a special manner. ◊

Once we have the lemma, we basically use the same type of reasoning we have used so many times before:

Given $0 < a/b < 1$ choose integers k and l such that

$$N_{k-1} < b \le N_k \text{ and } l/N_k \le a/b < (l+1)/N_k.$$

$$aN_k - bl < b \le N_k \text{ and } l < N_k.$$

By the lemma, we can say:

$$aN_k - bl = d_1 + d_2 + \ldots + d_r \text{ and } l = d'_1 + \ldots + d'_s,$$

where d_i and d'_j are divisors of N_k. (and r and s are $O(\sqrt{\log N_{k-1}})$) Now define integers u_i and v'_j:

$$u_i = N_k b / d_i \qquad v_j = N_k / d'_j$$

Since $d_r \le aN_k - bl < b$, it follows that $v_1 \le N_k < N_k b / d_r = u_r$. This proves that $v_s < \ldots < v_1 < u_r < \ldots < u_1$ provided $d_1 < \ldots < d_r$ and $d'_1 < \ldots < d'_s$. Then

$$1/v_1 + \ldots + 1/v_s + 1/u_1 + \ldots + 1/u_r = 1/N_k(1 + (aN_k\text{-}bl)/b) = a/b$$

And thus $n \le r+s = 2\,O(\sqrt{\log N_{k-1}}) = O(\sqrt{\log b})$. Δ

It is perhaps intuitive, but not obvious, that $\sqrt{\log n} < \log n / \log \log n$, so we will prove it.

Theorem

$O(\sqrt{\log n}) < O(\log n / \log \log n)$

proof

Clearly, $O(b^a) > O(a^2)$		for sufficiently large a.
$O(a^2)$	$<$	$O(b^a)$
$O(\log^2 x)$	$<$	$O(x)$
$O(x)$	$<$	$O(x^2 / \log^2 x)$
$O(\sqrt{x})$	$<$	$O(x / \log x)$
$O(\sqrt{\log n})$	$<$	$O(\log n / \log \log n)$ Δ

We have not seen any references to any lower bound on the number of terms. This would be a bound on the inverse of the function E(t) described earlier.

Length/Denominator Bounds

There are also bounds involving both length and denominator. In 1986, Yokota proves:

Theorem

Suppose P is a large prime. Then there is no algorithm which yields both

$$L(P) \leq c\log P/\log\log P$$

and

$$D(P) \leq P(\log P)^{1+1/(c+\varepsilon)} \text{ for } \varepsilon > 0$$

proof

Yokota uses the following lemma:

Lemma

Let M be a large number. If $t \leq (\log M)^{1+1/(c+\varepsilon)}$ for $\varepsilon > 0$, then

$$\binom{t}{0} + \binom{t}{1} + \dots + \binom{t}{[c\log M/\log\log M]} < M$$

The proof of the lemma involves algebra and Stirling's formula. ◊

Yokota uses a technique similar to the one used to prove the lower bound for D(N). We will deviate slightly from Yokota's proof to remain consistent.

Suppose that we can write a/P as the sum of k unit fractions, with $k \leq c\log P/\log\log P$ and the largest denominator $\leq P(\log P)^{1+1/(c+\varepsilon)}$ for $\varepsilon > 0$.

Then, using the notation of the proof for D(N), we note that we must have $x_t \leq P(\log P)^{1+1/(c+\varepsilon)}$.

We still need (P-1) different (b,c) pair, but they correspond to subsets of size $\leq c\log P/\log\log P$ (rather than any size subset). The total set has size t. So the number of such subsets is clearly

$$\binom{t}{1} + \binom{t}{2} + \dots + \binom{t}{[c\log P/\log\log P]}$$

Since we need P-1 (b,c) pairs, we need

$$\binom{t}{1} + \binom{t}{2} + \dots + \binom{t}{[c\log P/\log\log P]} \geq P-1$$

But since the x_i are distinct, it is clear that

$$tP \leq x_t$$

But then

$$tP \leq P(\log P)^{1+1/(c+\varepsilon)}$$
$$t \leq (\log P)^{1+1/(c+\varepsilon)}$$

and thus, by the lemma, we have

$$\binom{t}{0} + \binom{t}{1} + \dots + \binom{t}{[c\log P/\log\log P]} < P$$

$$\binom{t}{1} + \binom{t}{2} + \dots + \binom{t}{[c\log P/\log\log P]} < P-1$$

This is an obvious contradiction. Δ

The following is a table of all the upper and lower asymptotic bounds.

	Lower Bound	Upper Bound
D(N)	N log N Bleicher/Erdös 1976	$N(\log N)^{1+\delta(N)}$ Yokota 1988
	$(N\log N \log\log N) / (\log_{r+1} N \prod_{j=4}^{r+1} \log_j N$ Bleicher/Erdös 1976	
L(N)	1	$\sqrt{\log N}$ Vose 1985
Both D(N) L(N)	 $N(\log N)^{1+1/(c+\varepsilon)}$ for $\varepsilon > 0$ $c\log P/\log\log P$ Yokota 1986	 $N(\log N)^2 \log\log N$ $(1+\varepsilon)(\log N)/(\log\log N)$ Tenenbaum/Yokota 1990

6 Problems Involving a Fixed Number of Terms

We now move from construction algorithms to Diophantine equations. If we fix the number of terms allowed in an Egyptian fraction expansion, we discover some very interesting problems.

If we fix the number of terms to a constant number, then we simply have a specific case of Rav's theorem stated earlier (for some value of k). The interesting questions involve fixing both the numerator and the number of terms. But first, we will repeat the theorem and go over the proof:

Theorem

$m/n = 1/x_1 + 1/x_2 + \ldots + 1/x_k$ if and only if there exist positive integers M and N and divisors $D_1, \ldots, D_k$ of N such that $M/N = m/n$ and $D_1 + D_2 + \ldots + D_k = 0 \pmod M$. Also, the last condition can be replaced by $D_1+D_2+\ldots+D_k = M$; and the condition $(D_1, D_2, \ldots, D_k) = 1$ may be added without affecting the validity of the theorem.

proof

First, suppose M and N exist which satisfy the given conditions. Then we simply have

$$\frac{m}{n} = \frac{M}{N} = \frac{D_1 + D_2 + \ldots + D_k}{cN} = \frac{1}{cN/D_1} + \frac{1}{cN/D_2} + \ldots + \frac{1}{cN/D_k}$$

On the other hand, suppose $m/n = 1/x_1 + 1/x_2 + \ldots + 1/x_k$ is solvable. Then

$$\frac{m}{n} = \sum_{i=1}^{k}\frac{1}{x_i} = \frac{\sum_{i=1}^{k} x_1 \cdots x_{i-1}x_{i+1} \cdots x_k}{x_1x_2 \cdots x_k} = \frac{M}{N}$$

Clearly, then, $M = D_1 + D_2 + \ldots + D_k$, where the D_i all divide N. And we are done. If $(D_1, D_2, \ldots, D_k) = d \neq 1$, then we simply take M/d and N/d instead. Δ

The 4/n Problem

The outstanding unsolved question of Egyptian fractions concerns the case $4/n$ Can a proper fraction $4/n$ always be expressed with 3 or fewer terms? In other words, can the Diophantine equation

$$\frac{4}{n} = \frac{1}{a} + \frac{1}{b} + \frac{1}{c}$$

always be solved in positive integers for any integral value of *n* greater than 4?

Erdös and Straus believe it can always be solved. It has been verified for very large values of *n*, but never proved. Nicola Franceschine has verified the conjecture for $n \leq 10^8$. Mordell [Mor69] has shown it is true, except possibly in cases where *n* is prime and congruent to $1^2, 11^2, 13^2, 17^2, 19^2$, or $23^2 \pmod{840}$.

Vaughan [Vau70] has shown that if $E_a(N)$ is the number of natural numbers n not exceeding N for which more than 3 terms are needed to express a/n, then $E_a(N)$ «

$N \exp\{-(\log N)^{2/3} / C(a)\}$ Most of the asymptotic results in this area use sieve methods.

To provide a flavor of the problem, we will go through Mordell's result in great detail.

Theorem

$4/n = 1/a \ \ 1/b + 1/c$ (EQ 1)

is solvable in positive integers for any integer $n > 4$ where $n \not\equiv 1^2, 11^2, 13^2, 17^2, 19^2$, or $23^2 \pmod{840}$

proof

It will be useful to use the following two equations:

Note that if $na + b + c = 4abcd$ (EQ 2)

then $1/bcd + 1/acdn + 1/abdn = 4/n$

Lemma 1

If the (EQ 1) is solvable for n, then it is also solvable for all multiples of n.

Suppose that $4/n = 1/a + 1/b + 1/c$. Then $4/mn = 1/ma + 1/mb + 1/mc$. ◊

Lemma 2

(EQ 1) is solvable for all $n \not\equiv 1 \pmod 4$

Clearly, if $n = 4a$, then $4/n = 1/a$, so $4/n$ is expressible as a single unit fraction (and, trivially, also as the sum of three unit fractions by the splitting relation). Thus, (EQ 1) is solvable for $n \equiv 0 \pmod 4$.

In (EQ 2), if we let $a = 2$, $b = 1$, and $c = 1$, then we have

$2n + 1 + 1 = 8d$

so

$n = 4d-1$

Thus, if we allow d to range over the integers, we find that $4/n$ is always expressible as the sum of 3 unit fractions. In other words, (EQ 1) is solvable for $n \equiv 3 \pmod 4$.

Similarly, note that if we let $a = 1$, $b = 1$, and $c = 1$, then we have

$n + 1 + 1 = 4d$

so

$n = 4d - 2$

Thus, (EQ 1) is solvable for $n \equiv 2 \pmod 4$.

Thus, (EQ 1) is solvable for $n \not\equiv 1 \pmod 4$. ◊

Lemma 3

(EQ 1) is solvable for all $n \not\equiv 1 \pmod 8$

In (EQ 2), if we let $a = 1$, $b = 1$, and $c = 2$, then we have

$n + 1 + 2 = 8d$

so

$n = 8d - 3$

Thus, (EQ 1) is solvable for all $n \equiv 5 \pmod 8$.

By Lemma 2, (EQ1) is solvable for all $n \not\equiv 1 \pmod 4$, which means $n \not\equiv 1$ or $5 \pmod 8$. Thus, (EQ 1) is solvable for all $n \not\equiv 1 \pmod 8$. ◊

Lemma 4

(EQ 1) is solvable for all $n \not\equiv 1 \pmod 3$

From (EQ 2), we have

$$na + b + c = 4abcd$$
$$na + b = 4abcd - c = c(4abd - 1)$$
$$na + b = (4abd - 1)c \qquad \text{(EQ 3)}$$

If we let $a = b = d = 1$, then we have

$$n + 1 = 3c$$
$$n = 3c - 1$$

Thus, (EQ 1) is solvable for $n \equiv 2 \pmod 3$.
Note that $4/3 = 1/1 + 1/4 + 1/12$.
Thus, by Lemma 1, (EQ 1) is solvable for $n \equiv 0 \pmod 3$.
Thus, (EQ 1) is solvable for all $n \not\equiv 1 \pmod 3$. ◊

Lemma 5

(EQ 1) is solvable for all $n \not\equiv 1, 2,$ or $4 \pmod 7$

From (EQ 3), if we take $a = 1$, $b = 2$, $d = 1$, then

$$n + 2 = 7c \quad \Rightarrow \quad n = 7c - 2 \quad \Rightarrow \quad n \equiv 5 \pmod 7$$

If $a = 2$, $b = 1$, $d = 1$, then

$$2n + 1 = 7c \quad \Rightarrow \quad 2n = 7c - 1 \quad \Rightarrow \quad 2n = 6 \pmod 7 \quad \Rightarrow \quad n \equiv 3 \pmod 7$$

If $a = 1$, $b = 1$, $d = 2$, then

$$n + 1 = 7c \quad \Rightarrow \quad n = 7c - 1 \quad \Rightarrow \quad n \equiv 6 \pmod 7$$

Thus, (EQ 1) is solvable for $n \equiv 3, 5,$ or $6 \pmod 7$.
Noting that $4/7 = 1/2 + 1/15 + 1/210$, lemma 1 tells us that (EQ 1) is also solvable for $n \equiv 0 \pmod 7$.
Thus, (EQ 1) is solvable for all $n \not\equiv 1, 2,$ or $4 \pmod 7$. ◊

Lemma 6

(EQ 1) is solvable for all $n \not\equiv 1$ or $4 \pmod 5$

Lemma 4 tells us that (EQ 1) is solvable for all $n \not\equiv 1 \pmod 3$. Thus, (EQ 1) is solvable for all $n \not\equiv 1, 4, 7, 10,$ or $13 \pmod{15}$.
Again, taking (EQ 3), if we let $a = 1$, $b = 2$, $d = 2$, then

$$n + 2 = 15c \quad \Rightarrow \quad n \equiv 13 \pmod{15}$$

If $a = 2$, $b = 1$, $d = 2$, then

$$2n + 1 = 15c \quad \Rightarrow \quad 2n \equiv 14 \pmod{15} \quad \Rightarrow \quad n \equiv 7 \pmod{15}$$

Thus, (EQ 1) is solvable for $n \equiv 7$ or $13 \pmod{15}$.
So (EQ 1) is solvable for all $n \not\equiv 1, 4,$ or $10 \pmod{15}$.
Thus, (EQ 1) is solvable for all $n \not\equiv 0, 1$ or $4 \pmod 5$.
Nothing that $4/5 = 1/2 + 1/5 + 1/10$, lemma 1 tells us that (EQ 1) is solvable for $n \equiv 0 \pmod 5$.
Thus, (EQ 1) is solvable for all $n \not\equiv 1$ or $4 \pmod 5$. ◊

Now for the proof of the theorem.
Lemmas 3 and 4 combine to tell us that (EQ 1) is solvable for all $n \not\equiv 1 \pmod{24}$.
So (EQ 1) is solvable for all $n \not\equiv 1, 25, 49, 73, \text{or } 97 \pmod{120}$.
But then Lemma 6 tells us (EQ 1) is solvable for all $n \not\equiv 1 \text{ or } 49 \pmod{120}$.
Combining this will Lemma 5, we see that (EQ 1) is solvable for all n $\not\equiv 1, 121, 361, 169, 289, \text{or } 569 \pmod{840}$.
Thus, (EQ 1) is solvable for all $n \not\equiv 1^2, 11^2, 13^2, 17^2, 19^2, \text{or } 23^2 \pmod{840}$. Δ

The 5/n Problem

Sierpinski has conjectured that 5 / n can also always be expressed as the sum of 3 or fewer unit fractions. Stewart [Ste64] has confirmed this for all $n \le 1057438801$ and for all n not of the form 278460k + 1. Stewart takes a slightly different approach to proving this, showing how to pick a first fraction which leaves a result which can be expressed with 2 terms.

The 6/n Problem

Webb [Web74] proves that 6 / n is solvable for all n not of the form $n \equiv 1, 61$, or 541 (mod 660).

Webb also states that 10 / n is solvable except for $n \equiv 1 \pmod{10}$, 3 (mod 140), 43 (mod 140), or 7 (mod 60).

The k/n Problem

Kiss makes the larger conjecture that for $4 \le a \le 7$, a / b has an expansion of length 3 or less, and for $8 \le a \le 12$, a / b has an expansion of length of 4 or less. [Kis60]

Sierpinski makes an even more general conjecture, that for a given k, there exists N such that all k / n with n > N are expressible as the sum of 3 or fewer unit fractions. [Gar92] It then seems logical to extend this to the following conjecture:

Conjecture

Given $t \ge 3$ and $k > t$, there exists N such that:

For all n > N, k / n is expressible as the sum of t or fewer unit fractions.

A computer was used to obtain a list of some rationals not expressible as the sum of t or fewer unit fractions, where we set t = 3, 4, 5, or 6 for various values of k. The results are listed in Appendix C.

It seems apparent from the computer results that the following are the smallest (in the sense of smallest denominator) rationals not expressible in a fixed number of terms:

t	Smallest Rational Not Expressible in t Terms
2	2/3
3	8/11
4	16/17
5	77/79
6	728/739

Thus, an interesting question might be, what is the asymptotic growth of E(t), where E(t) is the smallest denominator q such that there exists p such that $p / q < 1$ is not expressible in t terms. Thus, from above, we have E(6) = 739.

7 Conclusions

We have explored many of the intricacies of algorithms for Egyptian fraction expansions. We have also looked at some Diophantine equation problems resulting from Egyptian fractions.

This just scratches the surface of the wealth of number-theoretic problems arising from Egyptian fractions. We list some of them in Appendix A.

Some of the findings of this paper include: values for M(P), E(t), and a comparison of some of the various algorithms available. And we have also raised a few suggestions and questions which prompt further research, including the performance of the Fibonacci-Sylvester algorithm, and the “optimal” practical number algorithm. We would also like to point out that, apparently, no research has been done on finding a lower bound for the number of terms.

The fundamental trouble in solving problems concerning Egyptian fractions, and many number theory problems, is the apparent random distribution of prime numbers. This reduces most attempts to searches for only asymptotic results, while dooming most efforts at the k/n problem to failure.

Still, there are many problems where further progress can be made, and the asymptotic bounds for L(N) and D(N) continue to move.

Little could the Egyptians know that their simple table of fractions could thousands of years later be the subject of so much research. It is bound to be the subject of research for many years to come.

A Misc. Egyptian Fraction Problems

There are many other problems concerning Egyptian fractions, some of which we will list here.

Improper Rationals

Stewart [Ste64] proves that any improper rational can be written as the sum of unit fractions. We can simply write the harmonic series 1/2 + 1/3 + 1/4 + ... until we have a proper fraction remaining.

Znam's Problem

Znam's problem is: Does there exist an integer x_i for every integer $s > 1$ such that x_i is a proper factor of $x_1 \cdots x_{i-1}x_{i+1} \cdots x_s + 1$ for $i = 1,\ldots,s$?
The equation

$$\sum_{j=1}^{s} \frac{1}{x_j} + \frac{1}{x_1 \cdots x_s} = 1 \text{ where } 1 < x_1 < x_2 < \ldots < x_s$$

is related to Znam's problem. [Zhe87]

Representing 1 with Egyptian Fractions

Define U_n = smallest number of different unit fractions totalling 1 where the largest unit fraction is $\leq 1/n$. For example, $U_3 = 5$ because

$$1 = \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{20}$$

is the shortest expansion of 1 without using 1/2.
Erdös and Straus [Erd78] prove that there are constants c_1 and c_2 such that

$$(e-1)n - c_2 < U_n < (e-1)n + c_1 n/\log n$$

Representing 1 with Relatively Prime Denominators

A somewhat interesting question is whether 1 can be represented as $1/x_1 + 1/x_2 + \ldots + 1/x_k$, with $x_i \nmid x_j$ for all $i \neq j$. This was solved by Burshtein [Bur73], providing one possible solution. The reciprocals of the following integers sum to 1, and they are all relatively prime:

6 10 14 15 21 22 33 35
55 77

26	39	65	91	
34	51	119	187	
38	57	95	133	
58	145	319		
62	93	155		
82	123	287		
106	159	265	583	
118	295	413		
122	355	497		
309	515	1133		
226	791	1243		
835	1169	1837		
1329	2215	3101	4873	
1438	3595	5033	7909	
5854	8781	14635	20489	32197

141	188	235	
332	415	581	
267	356	979	
1167	1556	1945	2723

The numbers are listed in this form to facilitate showing that the conditions hold.

Odd Egyptian Fractions

Breusch proves in [Bre54] that every positive rational with odd denominator can be written as the sum of a finite number of unit fractions with odd denominators. The proof involves proving that a recursive procedure always terminates.

Representing 1 with Odd Denominators

$$1 = \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \frac{1}{9} + \frac{1}{15} + \frac{1}{21} + \frac{1}{27} + \frac{1}{35} + \frac{1}{63} + \frac{1}{105} + \frac{1}{135}$$

[Bur73]

The Odd Greedy Algorithm

It is unknown whether the odd greedy "algorithm" always terminates. [Guy80] It is just like the normal greedy algorithm, except we always take the largest unit fraction with an odd denominator less than the remainder.

For example, with the normal greedy algorithm we get the expansion

$$\frac{2}{7} = \frac{1}{4} + \frac{1}{28}$$

On the other hand, with the odd greedy algorithm, we get the expansion

$$\frac{2}{7} = \frac{1}{5} + \frac{1}{13} + \frac{1}{115} + \frac{1}{10465}$$

Schinzel's Conjecture

Straus and Subbarao [Str78] prove

$a / n = 1 / x \pm 1 / y \pm 1 / z$ (A. Schinzel's conjecture (1956))

has integral solutions for sufficiently large n for all $a < 40$.

They prove this first by showing that $a / n = 1 / x \pm 1 / y$ is solvable for a = 1, 2, 3, 4, or 6. They then look at the equation $a / n = 1 / m \pm r / mn$ and look at various cases involving a, r, and $\phi(r)$.

B The Rhind Mathematical Papyrus

Here are the expansions given in the Rhind Mathematical Papyrus taken from [Gil72].

Fraction 2/	Divisors of Expansion				Fraction 2/	Divisors of Expansion			
					53	30	318	795	
5	3	15			55	30	330		
7	4	28			57	38	114		
9	6	18			59	36	236	531	
11	6	66			61	40	244	488	610
13	8	52	104		63	42	126		
15	10	30			65	39	195		
17	12	51	68		67	40	335	536	
19	12	76	114		69	46	138		
21	14	42			71	40	568	710	
23	12	276			73	60	219	292	365
25	15	75			75	50	150		
27	18	54			77	44	308		
29	24	58	174	232	79	60	237	316	790
31	20	124	155		81	54	162		
33	22	66			83	60	332	415	498
35	30	42			85	51	255		
37	24	111	296		87	58	174		
39	26	78			89	60	356	534	890
41	24	246	328		91	70	130		
43	42	86	129	301	93	62	186		
45	30	90			95	60	380	570	
47	30	141	470		97	56	679	776	
49	28	196			99	66	198		
51	34	102			101	101	202	303	606

Thus, for example, $\frac{2}{39} = \frac{1}{26} + \frac{1}{78}$

C Computer Results

Practical Numbers

The following is a table of the minimum number of terms required to write a number as the sum of divisors of various practical numbers. These were calculated by computer except where noted.

It is interesting to note that the number of terms required for $s_1 \cdots s_n$ seem to be better than the number for $\prod_k$. For example, 9699690 requires 9 terms, but 16216200, a larger number, requires just 8 terms.

Practical Number			Terms	Worst Number
$\prod_2$	=	6	2	5
$\prod_3$	=	30	4	29
$\prod_4$	=	210	5	209
$\prod_5$	=	2310	7	2252
$\prod_6$	=	30030	8	29990
$\prod_7$	=	510510	8	510509
$\prod_8$	=	9699690	9	9699631
$\prod_9$	=	223092870	$\geq$ 10	
$s_1 s_2$	=	6	2	5
$s_1 s_2 s_3$	=	24	3	23
$s_1 s_2 s_3 s_4$	=	120	4	119
$s_1 \cdots s_5$	=	840	5	839
$s_1 \cdots s_6$	=	7560	6	7559
$s_1 \cdots s_7$	=	83160	7	83016
$s_1 \cdots s_8$	=	1081080	7	1081053
$s_1 \cdots s_9$	=	16216200	8	16215773
$s_1 \cdots s_{10}$	=	259459200	$\geq$ 8	
2^{10}	=	1024	10‡	1023
2^{23}	=	8388608	23‡	8388607

Algorithm Comparison

The following is a comparison of 4 different algorithms: Fibonacci-Sylvester, Bleicher / Erdös, Tenenbaum / Yokota, and Golomb.

In all the following comparisons, only prime n in the ranges given are used. All fractions a / n where a < n are expanded using the four algorithms.

The numbers under the Average column represent averages over all expansions in that range, and the numbers under the Worst column represent the worst taken from all expansions in that range.

‡ done by hand

Length Comparison

The numbers in the table are the number of terms in the expansions.

		Average					Worst			
n		Fib	Ble	Ten	Gol		Fib	Ble	Ten	Gol
2...102	3.7	6.0	5.6	8.2		8	10	9	100	
...202		4.3	6.2	6.3	11.6		11	10	10	198
...302		4.5	8.1	7.0	13.5		9	12	11	292
...402		4.6	8.1	7.1	14.9		10	12	11	400
...502		4.8	8.2	7.2	16.0		12	13	12	498
...602		4.9	8.2	7.3	16.9		10	13	12	600
...702		4.9	8.1	7.4	17.6		11	13	12	700
...802		5.0	8.0	7.5	18.3		11	13	12	796
...902		5.1	7.9	7.5	18.9		11	13	12	886
...1002	5.1	7.9	7.6	19.5		11	13	12	996	
...1102	5.1	7.8	7.7	20.0		13	12	12	1096	
...1202	5.2	7.8	7.7	20.4		11	12	12	1200	
...1302	5.2	7.9	7.7	20.9		12	12	13	1300	
...1402	5.3	7.9	7.8	21.2		12	13	12	1398	
...1502	5.3	7.9	7.8	21.6		11	12	13	1498	
...1602	5.3	7.9	7.8	22.0		12	13	13	1600	
...1702	5.3	7.9	7.9	22.3		11	13	13	1698	
...1802	5.4	8.0	7.9	22.6		12	12	13	1800	
...1902	5.4	8.0	8.0	22.9		12	13	13	1900	
...2002	5.4	8.0	8.1	23.2		12	13	13	1998	

Denominator Comparison

The numbers in the table are the largest denominators in the expansions.

		Average					Worst			
n		Fib	Ble	Ten	Gol		Fib	Ble	Ten	Gol
2...102	5.8	4.0	4.1	3.6		150	5	5	5	
...202		9.0	4.5	4.7	4.5		1348	5	6	5
...302		10.3	5.6	6.2	4.8		396	6	8	5
...402		11.6	5.7	6.4	5.1		537	6	8	6
...502		12.9	6.0	6.5	5.5		2847	7	8	6
...602		13.6	6.1	6.6	5.6		259	7	8	6
...702		14.6	6.1	6.6	5.7		759	7	8	6
...802		15.0	6.1	6.7	5.8		304	7	8	6
...902		15.9	6.2	6.8	5.8		289	7	8	6
...1002	16.5	6.3	6.9	5.9		862	7	8	6	
...1102	17.2	6.4	6.9	6.0		3455	7	8	7	
...1202	17.6	6.4	7.0	6.2		592	7	8	7	
...1302	18.2	6.4	7.0	6.3		877	7	9	7	
...1402	18.5	6.5	7.0	6.4		997	7	9	7	
...1502	19.0	6.5	7.1	6.5		959	7	9	7	
...1602	19.7	6.5	7.1	6.5		916	7	9	7	
...1702	19.9	6.5	7.1	6.6		1160	7	9	7	
...1802	20.4	6.5	7.1	6.6		647	7	9	7	
...1902	20.5	6.5	7.1	6.7		775	7	9	7	
...2002	21.0	6.5	7.1	6.7		1236	7	9	7	

Character Comparison

The numbers in the table represent how many characters would be used to print out the expansions – the number of digits + the number of terms - 1.

		Average					Worst			
n		Fib	Ble	Ten	Gol		Fib	Ble	Ten	Gol
2...102	14.0	20.4	18.5	29.6		309	36	34	458	
...202		20.9	22.5	23.0	47.9		2709	38	39	1046
...302		23.7	34.1	29.1	59.4		802	53	50	1610
...402		26.5	34.4	30.1	68.5		1086	53	52	2343
...502		29.2	35.0	30.9	76.2		5708	56	58	3029
...602		30.6	36.6	31.6	82.8		527	59	58	3743
...702		32.7	35.9	32.1	87.8		1530	59	58	4443
...802		33.6	35.4	32.8	92.6		621	61	57	5115
...902		35.4	35.0	33.1	96.6		585	62	58	5745
...1002	36.6	35.0	33.7	100.6		1736	62	59	6515	
...1102	38.1	35.6	35.1	104.3		6924	61	59	7312	
...1202	38.9	35.5	35.5	108.1		1196	60	60	8144	
...1302	40.2	35.7	35.7	111.8		1767	60	63	8944	
...1402	40.8	36.1	36.0	114.6		2006	63	63	9728	
...1502	41.9	36.2	36.3	118.0		1927	61	66	10528	
...1602	43.3	36.4	36.6	120.8		1843	61	65	11344	
...1702	43.7	36.6	36.9	123.1		2330	64	66	12128	
...1802	44.7	36.7	37.2	125.8		1310	61	67	12944	
...1902	44.9	36.8	37.6	128.3		1561	61	64	13744	
...2002	46.1	36.9	37.9	130.4		2485	63	66	14528	

Best Percentages

The following percentages refer to the percentage of the time that an algorithm is the best (or tied for the best) in any certain category. For example, in the range of n from 402 to 502, the Fibonacci-Sylvester algorithm has the fewest terms 96% of the time.

n: 402 to 502

	Fib	Ble	Ten	Gol
Length	96%	2%	9%	15%
Denominator	12%	19%	12%	70%
Characters	57%	10%	21%	19%

n: 1002 to 1102

	Fib	Ble	Ten	Gol
Length	95%	8%	9%	11%
Denominator	8%	34%	17%	57%
Characters	51%	24%	23%	16%

n: 1902 to 2002

	Fib	Ble	Ten	Gol
Length	95%	9%	9%	9%
Denominator	6%	48%	21%	44%
Characters	46%	34%	24%	14%

Practical Numbers Revisited

The following is a table of M(P) (see the “Optimal” Practical Number Algorithm, section 3). The table lists only the entries where M(P) is different, listing the smallest P for which M(P) is a certain value. For example, M(7) = 4, but M(5) = 4, so we only list M(5).

P	M(P)
2	1
3	2
5	4
11	6
17	12
31	16
37	18
41	20
47	24
67	30
79	36
97	42
101	48
127	60
173	72
197	84
227	90
239	96
257	108
283	120
367	144
409	168
487	180
557	210
587	216
607	240
751	288
821	300
877	336
997	360
1181	420
1361	480
1523	504
1567	540
1693	600
1867	630
1877	660
2027	720
2423	840
2887	960
3061	1008
3229	1080
3607	1200
3847	1260
4373	1440
4919	1560
5051	1620
5087	1680
5981	1800
6047	1920
6131	1980
6563	2100
6947	2160
7451	2340
7649	2400
7817	2520
9371	2880

9923	3120
10427	3240
10903	3360
12101	3600
12497	3780
13451	3960
14051	4200
14887	4320
15131	4620
16139	4680
16411	5040
19373	5760
19913	5880
20011	5880
20533	6120
21067	6240
21179	6300
22571	6720
24391	7200
25391	7560
28807	7920
29021	8400
30757	8820
31139	9240
34583	10080
39317	10920
40009	10920
40343	11340
40693	11760
42433	11880
43207	12240
43541	12600
48371	13440
48973	13860
52433	15120
59539	16380
61169	16800
62501	17640
65003	17640
66697	18480
71429	19800
72547	20160
72689	21000
74887	21420
75083	21600
75989	22176
76091	22320
77383	22440
77641	24000
77743	24300
78539	25344
79031	26136
79811	27132

Fixed Number of Terms

The following are values of k / n which aren't expressible as the sum of a certain number of unit fractions. Testing was done only for the following values of n:

k	largest value of n tested
8-9	2222
10-11	5000
12-24	1000
25-49	2000
77	1000
78-99	300
100-129	400

So, for xample, all 9/n for n ≤ 2222 are expressible as the sum of 3 unit fractions except 9/11 and 9/19.

Not Expressible with 3 Terms

k	n									
8	11	17	131	241						
9	11	19								
10	11	43	61	67	181					
11	37									
12	13	25	29	31	37	73	97	193	433	577
13	14	53	61	67	79	211	281			
14	17	19	29	59	257	353	841			
15	16	17	19	23	31	34	47	53	61	
	79	113	122	137	151	197	226	233	271	
	541									

Not Expressible with 4 Terms

k	n			
16	17			
17-20				
21	23			
22	23			
23-26				
27	29			
28	29	59		
29	59			
30	31	41		
31-32				
33	34	67		
34	41	43		
35	47			
36	37			
37	38			
38	39	41	47	61
39	43	47		
40	41	43		
41	43	83		
42				
43	47	97		
44	47	53	137	

45	47	61		
46	47	49		
47	53	57	59	71
48	97			
49	50	59	71	

Not Expressible with 5 Terms

k	n	
77	79	
78-100		
101	107	
102	103	
104	107	
106	107	
108	109	
112	113	
115	118	
117	118	
119	127	
123	127	
129	131	137

Not Expressible with 6 Terms

$$\frac{728}{739}$$

D References

For a comprehensive bibliography of Egyptian fractions, write to
Paul J. Campbell
Beloit College
Beloit, Wisconsin 53511
For references in Mathematical Reviews, see section 11D68.
Also, [Guy80] lists 4 pages of references.

[Bec69] Beck, Bleicher, and Crowe. "Egyptian Fractions," in Excursions Into Mathematics. pp. 421-434. Worth Publishers, Inc., 1969.

[Ble72] Bleicher, M. N. "A New Algorithm for the Expansion of Egyptian Fractions," *Journal of Number Theory* 4 (1972) 342-382.

[Ble76a] Bleicher, M. N. and Erdös, P. "Denominators of Egyptian Fractions," *Journal of Number Theory* 8 (1976). 157-168

[Ble76b] Bleicher, M. N. and Erdös, P. "Denominators of Egyptian Fractions II," *Illinois Journal of Mathematics* 20 (1976). 598-613

[Bre54] Breusch, R. "A Special Case of Egyptian Fractions," *American Mathematical Monthly* 61 (1954) 200-201.

[Bur73] Burshtein, Nechemia. "On Distinct Unit Fractions Whose Sum Equal 1," *Discrete Mathematics* 5 (1973) 201-206

[Cam77] Campbell, Paul. "A 'Practical' Approach to Egyptian Fractions," *Journal of Recreational Mathematics* 10 (1977-78) 81-86.

[Dun66] Dunton, M. and Grimm, R. E. "Fibonacci on Egyptian Fractions," *Fibonacci Quarterly* 4 (1966) 339-354. Translation of Fibonacci's original work, Liber Abacci (1202).

[Erd50] Erdös, Paul. "The Solution in Whole Numbers of the Equation: $1/x_1 + 1/x_2 + ... + 1/x_n = a/b$," *Mat. Lapok* 1 (1950) 192-210.

[Erd78] Erdös, P. and Straus, E. "Representation of 1 by Egyptian Fractions," *American Mathematical Monthly* 78 (1971) 302.

[Gar92] Gardner, Martin. "Egyptian Fractions," in Fractal Music, Hypercards And More..., pp. 100-109. W.H. Freeman and Company, 1992.

[Gil72] Gillings, Richard. Mathematics In The Time Of The Pharaohs. MIT Press, 1972.

[Gol62] Golomb, Solomon. "An Algebraic Algorithm For The Representation Problems of the Ahmes Papyrus," *American Mathematical Monthly*. 69 (1962) 785-786.

[Guy80] Guy, Richard. "Egyptian Fractions," in Unsolved Problems in Number Theory, Section D11. Springer-Verlag, 1980.

[Hey80] Heyworth, Malcolm R. "More on Panarithmic Numbers," *New Zealand Mathematics Magazine* 17 (1980) 28-34.

[Kis60] Kiss, E. "Remarks on the representation of fractions between 0 and 1 as the sum of unit fractions," *Acad. R. P. Romine Fil. Cluj Stud. Cerc. Mat.* 11 (1960) 319-323.

[May87] Mays, Michael. "A Worst Case of the Fibonacci-Sylvester Expansion," *The Journal of Combinatorial Mathematics and Combinatorial Computing* 1 (1987) 141-148.

[Mor69] Mordell, L. J. Diophantine Equations, pp. 287-290. Academic Press, 1968.

[Ren62] Rényi, A. "A New Approach to the Theory of Engel's Series," *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös nominatoae* 5 (1962) 25-32.

[Rib89] Ribenboim, Paulo. "Goldbach's Famous Conjecture," in The Book of Prime Number Records, Second Edition. pp. 229-235. Springer-Verlag, 1989. Also, "The Growth of $\pi(x)$," pp. 164-165.

[Rob79] Robinson, D. F. "Egyptian Fractions via Greek Number Theory," *New Zealand Mathematics Magazine* 16 (1979) 47-52.

[Sri48] Srinivasan, A. K. "Practical Numbers," *Current Science* 17 (1948) 179-180

[Ste64] Stewart, B. M. "Egyptian Fractions," in Theory Of Numbers. pp. 198-207. The Macmillan Company, 1964.

[Str78] Straus E. G. and Subbarao M. V. "On the Representation of Fractions as Sum and Difference of Three Simple Fractions," in Proceedings of the Seventh Manitoba Conference on Numerical Mathematics and Computing. pp. 561-579. Utilitas Mathematica Publishing Inc., 1978.

[Syl880] Sylvester, J. J. "On a Point in the Theory of Vulgar Fractions," *American Journal of Mathematics* 3 (1880) 332-335

[Ten90] Tenenbaum, G. and Yokota, H. "Length and Denominators of Egyptian Fractions, III," *Journal of Number Theory* 35 (1990). 150-156

[Vau70] Vaughan, R. C. "On A Problem of Erdös, Straus and Schinzel," *Mathematika* 17 (1970) 193-198.

[Vos85] Vose, M. D. "Egyptian Fractions," *The Bulletin of the London Mathematical Society* 17 (1985). 21-24.

[Web74] Webb, William A. "Rationals Not Expressible As a Sum of Three Unit Fractions," Elemente der Mathematik 29 (1974) 1-6.

[Web75] Webb, William A. "On A Theorem Of Rav Concerning Egyptian Fractions," *Canadian Mathematical Bulletin* 18 (1975) 155-156.

[Yok86a] Yokota, H. "On a Conjecture of M. N. Bleicher and P. Erdös," *Journal of Number Theory* 24 (1986) 89-94.

[Yok86b] Yokota, H. "Length and Denominators of Egyptian Fractions," *Journal of Number Theory* 24 (1986) 249-258.

[Yok88a] Yokota, H. "Length and Denominators of Egyptian Fractions II," *Journal of Number Theory* 28 (1988) 272-282.

[Yok88b] Yokota, H. "On a Problem of Bleicher and Erdös," *Journal of Number Theory* 30 (1988) 198-207.

[Zhe87] Zhenfu, C., Rui, L., and Liangrui, Z. "On the Equation $\sum_{j=1}^{s}(1/x_j) + (1/(x_1 \cdots x_s)) = 1$ and Znam's Problem," *Journal of Number Theory* 27 (1987) 206-211.

E Computer Program Listings

All programs were written in ANSI C.

Algorithm Package

The following is the algorithm package used to compare different algorithms in terms of length, denominators, and a combination of both.

FRACTIONS.H

```
#ifndef FRACTIONS_INCLUDED
#define FRACTIONS_INCLUDED

#include "vlint.h"


struct ExpansionStruct
{
    int numTerms;
    VLInt denoms[20];
    VLInt *maxDenom;
    int measure;
};

typedef struct ExpansionStruct Expansion;


#endif FRACTIONS_INCLUDED
```

FIB.C

```
/* fib.c
 *
 *  Fibonacci-Sylvester Algorithm
 *
 * Kevin Gong
 * Spring 1992
 *
 */

#include <assert.h>
#include <stdio.h>
#include "headers/fib.h"
#include "headers/fractions.h"
#include "headers/vlint.h"


void FibonacciConstruction(unsigned long goalNum, unsigned long goalDen,
                           Expansion *fibExp)
{
    static VLInt num, den;
    static VLInt s, r;
    static VLInt temp;
    static boolean init = FALSE;

    if ( ! init )
    {
        num.digits = NULL;      den.digits = NULL;
```

```
        temp.digits = NULL;     s.digits = NULL;    r.digits = NULL;
        init = TRUE;
    }

    VLIntCreate(&num, goalNum);
    VLIntCreate(&den, goalDen);

    InitExpansion(fibExp);

    while ( num.numDigits != 0 )
    {
        VLIntDivide(&den, &num, &s);
        VLIntMod(&den, &num, &r);

        if ( r.numDigits == 0 )
        {
            AddExpansionTerm(fibExp, &s);
            return;
        }

        VLIntSubtract(&num, &r, &temp);
        VLIntCopy(&temp, &num);
        VLIntAddDigit(&s, (char) 1, &temp);
        VLIntCopy(&temp, &s);

        VLIntMultiply(&den, &s, &temp);
        VLIntCopy(&temp, &den);
        AddExpansionTerm(fibExp, &s);
    }
}
```

GOLOMB.c

```
/* golomb.c
 *
 *  Golomb Algorithm
 *
 * Kevin Gong
 * Spring 1992
 *
 */

#include <stdio.h>
#include "headers/golomb.h"
#include "headers/general.h"
#include "headers/fractions.h"


static unsigned int MultiplicativeInverse(unsigned int p, unsigned int q,
                                          unsigned int *r);


void GolombConstruction(unsigned int goalNum, unsigned int goalDen,
                        Expansion *golExp)
{
    unsigned int inverse, r;
    static VLInt num, den;
    static VLInt temp;
    static boolean init = FALSE;

    InitExpansion(golExp);

    if ( ! init )
```

```
    {
        num.digits = NULL;      den.digits = NULL;      temp.digits = NULL;
        init = TRUE;
    }

    VLIntCreate(&num, goalNum);
    VLIntCreate(&den, goalDen);

    while ( TRUE )  /* infinite loop */
    {
        MakeLowestTerms(&goalNum, &goalDen);
        if ( goalNum == 1 )
        {
            VLIntCreate(&temp, (unsigned long) goalDen);
            AddExpansionTerm(golExp, &temp);
            return;
        }

        inverse = MultiplicativeInverse(goalNum, goalDen, &r);

        VLIntCreate(&temp, (unsigned long) inverse*goalDen);
        AddExpansionTerm(golExp, &temp);

        goalDen = inverse;
        goalNum = r;
    }
}


static unsigned int MultiplicativeInverse(unsigned int p, unsigned int q,
                                          unsigned int *r)
{
    register int index;

    for ( index = 0; index < q; index++ )
    {
        if ( (index*p) % q == 1 )
        {
            *r = ((index*p)-1)/q;
            return index;
        }
    }

    fprintf(stderr, "croak in Multiplicative inverse\n");
    fprintf(stderr, "p = %u, q = %u\n", p, q );
    exit(-1);
}
```

BLEICHER.C

```
/* bleicher.c
 *
 *  Bleicher/Erdös Algorithm
 *
 * Kevin Gong
 * Spring 1992
 *
 */

#include <stdio.h>
#include "headers/bleicher.h"
#include "headers/general.h"
#include "headers/fractions.h"
```

```
boolean IsPrime(int number);
int Primes(int nth);
int FindPI(int k);
void Sort(int data[], int number);
void FindDivisorSum(int b, int d[], int *numD, int k);


int divisors[10][1024];
int numDivisors[10];
int primes[25];
int PI[10];
int numPI = 0;
int numPrimes = 0;


void BleicherConstruction(unsigned int a, unsigned int N,
                          Expansion *bleExp)
{
    int k;
    int b;
    int d[1024];
    int numD;
    register int index;
    int q, r;
    static VLInt temp;
    static boolean init = FALSE;

    if ( ! init )
    {
        temp.digits = NULL;
        init = TRUE;
    }

    InitExpansion(bleExp);

    if ( a == 1 )
    {
        VLIntCreate(&temp, (unsigned long) N);
        AddExpansionTerm(bleExp, &temp);
        return;
    }

    k = 0;
    while ( PI[k] < N )
        k++;

    if ( PI[k] % N == 0 )
    {
        b = a*PI[k]/N;

        FindDivisorSum(b, d, &numD, k);

        for ( index = 0; index < numD-1; index++ )
        {
            VLIntCreate(&temp, PI[k]/d[index]);
            AddExpansionTerm(bleExp, &temp);
        }

        VLIntCreate(&temp, PI[k]/d[numD-1]);
        AddExpansionTerm(bleExp, &temp);
    }
    else
    {
```

```
        q = a*PI[k]/N;
        while ( (r = a*PI[k]-q*N) < PI[k]-PI[k]/k )
            q--;

/* handle q/PI[k] */
        FindDivisorSum(q, d, &numD, k);

        for ( index = 0; index < numD; index++ )
        {
            VLIntCreate(&temp, PI[k]/d[index]);
            AddExpansionTerm(bleExp, &temp);
        }

/* now handle r/N*PI[k] */
        FindDivisorSum(r, d, &numD, k);

        for ( index = 0; index < numD-1; index++ )
        {
            VLIntCreate(&temp, N*PI[k]/d[index]);
            AddExpansionTerm(bleExp, &temp);
        }

        VLIntCreate(&temp, N*PI[k]/d[numD-1]);
        AddExpansionTerm(bleExp, &temp);
    }

    return;
}

void FindDivisorSum(int b, int d[], int *numD, int k)
{
    register int index = numDivisors[k]-1;
    register int number = 0;

    while ( b != 0 )
    {
        while ( divisors[k][index] > b )
            index--;

        d[number] = divisors[k][index];
        b -= d[number];
        number++;
    }

    *numD = number;
}

(some functions not listed)
```

TENENBAUM.c

```
/* tenenbaum.c
 *
 *  Tenenbaum/Yokota Algorithm
 *
 * Kevin Gong
 * Spring 1992
 *
 */

#include <stdio.h>
#include "headers/tenenbaum.h"
#include "headers/general.h"
```

```
#include "headers/fractions.h"

extern  int divisors[10][1024];
static int sigDivisors[30][1024];
extern int numDivisors[10];
static int numSigDivisors[30];
extern int primes[25];
static int sigma[99];
extern int PI[10];
extern int numPI;
extern int numPrimes;
static int numSigma = 0;

void TenenbaumConstruction(unsigned int a, unsigned int N,
                          Expansion *tenExp)
{
    int k;
    int b;
    int d[1024];
    int numD;
    register int index;
    int s, r;
    int n;
    int sigProd;
    int rStar;
    static VLInt temp;
    static boolean init = FALSE;

    InitExpansion(tenExp);

    if ( ! init )
    {
        temp.digits = NULL;
        init = TRUE;
    }

    if ( a == 1 )
    {
        VLIntCreate(&temp, N);
        AddExpansionTerm(tenExp, &temp);
        return;
    }

    k = 0;
    while ( PI[k] < N )
        k++;

    if ( PI[k] % N == 0 )
    {
        b = a*PI[k]/N;

        FindDivisorSum(b, d, &numD, k);

        for ( index = 0; index < numD-1; index++ )
        {
            VLIntCreate(&temp, PI[k]/d[index]);
            AddExpansionTerm(tenExp, &temp);
        }

        VLIntCreate(&temp, PI[k]/d[numD-1]);
        AddExpansionTerm(tenExp, &temp);
    }
    else
    {
```

```
        s = a*PI[k]/N;
        while ( (r = a*PI[k]-s*N) < PI[k] )
            s--;

/* handle s/PI[k] */
        FindDivisorSum(s, d, &numD, k);

        for ( index = 0; index < numD; index++ )
        {
            VLIntCreate(&temp, PI[k]/d[index]);
            AddExpansionTerm(tenExp, &temp);
        }

/* now handle r/N*PI[k] */
        sigProd = 1;
        for ( n = 0; n < numSigma; n++ )
        {
            sigProd *= sigma[n];
            if ( primes[k] == sigma[n] )
                break;
        }

        rStar = r*(sigProd/PI[k]);

        FindSigmaSum(rStar, d, &numD, n);

        for ( index = 0; index < numD-1; index++ )
        {
            VLIntCreate(&temp, N*sigProd/d[index]);
            AddExpansionTerm(tenExp, &temp);
        }

        VLIntCreate(&temp, N*sigProd/d[numD-1]);
        AddExpansionTerm(tenExp, &temp);
    }

    return;
}

void FindSigmaSum(int b, int d[], int *numD, int n)
{
    register int index = numSigDivisors[n]-1;
    register int number = 0;

    while ( b != 0 )
    {
        while ( sigDivisors[n][index] > b )
            index--;

        d[number] = sigDivisors[n][index];
        b -= d[number];
        number++;
    }

    *numD = number;
}

(some functions not listed)
```

Practical Number Package

The following is the practical number package used to test numbers for practicality, and also to test certain attributes of practical numbers. Terms.c finds k such that k*denominator is a practical number (it tests only denominators which are prime). Length.c find the number of terms required to express all the numbers less than a given practical number.

TERMS.C

```
/* terms.c
 *
 *    Find k such that k*denominator is a practical number
 *
 * Kevin Gong
 * Spring 1992
 *
 */


#include <stdio.h>
#include <math.h>
#include "headers/general.h"
#include "headers/vlint.h"
#include "headers/fractions.h"
#include "headers/readPrimes.h"
#include "headers/divisors.h"


#define TRUE   1


void main(unsigned long argc, char **argv)
{
    unsigned long goalDen;
    unsigned long test;
    unsigned long start, finish;
    register int index;
    unsigned long co;
    unsigned long oldCo = 0;

    ReadPrimes();

    if ( argc != 3 )
    {
       fprintf(stderr, "Error -- usage:  terms <start> <finish>\n");
       exit(-1);
    }

    start = atoi(argv[1]);
    finish = atoi(argv[2]);
    index = 0;
    while ( primes[index] < start )
       index++;

    co = 1;
    while ( primes[index] <= finish )
    {
       goalDen = primes[index];
       test = co*goalDen;

       while ( ! PracticalNumber(test) )
       {
```

```
            test += goalDen;
            co++;
        }

        if ( co != oldCo )
        {
            fprintf(stdout, "%lu\t%lu\n", goalDen, test/goalDen);
            oldCo = co;
         }
        index++;
    }
}
```

LENGTH.C

```
/* length.c
 *
 *  Finds the number of divisors of a practical number needed to express
 *  all integers less than that practical number
 *
 */


#include <assert.h>
#include <stdio.h>
#include <math.h>
#include "headers/general.h"
#include "headers/vlint.h"
#include "headers/readPrimes.h"
#include "headers/divisors.h"


#define TRUE    1

int Expand(unsigned long number, unsigned long divisors[], int numDivisors);


void main(unsigned long argc, char **argv)
{
    unsigned long test;
    int numDivisors;
    unsigned long divisors[999];
    register long toTest;
    int numTerms;
    int worst;

    if ( argc != 2 )
    {
        fprintf(stderr, "Error -- usage:  terms <practical>\n");
        exit(-1);
    }

    ReadPrimes();

    test = (unsigned long) atoi(argv[1]);
    numDivisors = FastFindDivisors(test, divisors);
    worst = 0;
    for ( toTest = test-1; toTest > 0; toTest-- )
    {
        numTerms = Expand(toTest, divisors, numDivisors);
        if ( numTerms > worst )
        {
            worst = numTerms;
```

```
            fprintf(stdout, "Number %lu\tWorst Num Terms = %d (from %lu)\n",
                    test, worst, toTest );
        }
    }
}


/*
 * expands 'number' as the sum of items from 'divisors' and returns the
 * number of items used
 *
 */
int Expand(unsigned long number, unsigned long divisors[], int numDivisors)
{
    register int index;
    int numTerms = 0;

    index = numDivisors-1;

    while ( number != 0 )
    {
        assert(index>=0);
        while ( divisors[index] > number )
        {
            index--;
            assert(index>=0);
        }
        number -= divisors[index];
        index--;

        numTerms++;
    }

    return numTerms;
}
```

PRACTICAL.C

```
/* practical.c
 *
 *  Practical Numbers
 *
 * PracticalNumber returns TRUE if and only if 'number' is a practical number
 *
 * Kevin Gong
 * Spring 1992
 *
 */


#include "headers/practical.h"

boolean PracticalNumber(unsigned long number)
{
    unsigned long divisors[999];
    int numDivisors;
    register int index;
    register unsigned long sum = 0;

    numDivisors = FastFindDivisors(number, divisors);
    for ( index = 0; index < numDivisors-1; index++ )
    {
```

```
        sum += divisors[index];
        if ( sum < divisors[index+1]-1 )
            return FALSE;
    }
    return TRUE;
}
```

DIVISORS.C

```
/* divisors.c
 *
 * FastFindDivisors finds all the divisors of a number
 *      first finds prime factors, then computes divisors
 *      returns the number of divisors
 *
 * Kevin Gong
 * Spring 1992
 *
 */


#include <math.h>
#include "headers/divisors.h"
#include "headers/readPrimes.h"

typedef struct FactorStruct
{
    unsigned long   number;
    int             exponent;
} Factor;


int FastFindDivisors(unsigned long number, unsigned long divisors[])
{
    register unsigned long end;
    register int index;
    Factor  factors[500];
    int numFactors = 0;
    unsigned long lastFactor = 0;
    int numDivisors = 0;
    register int loop;
    int temp;
    unsigned long thisDivisor;
    register int inner;
    int i, j;

    end = (unsigned long) sqrt((double)number) + 1;
    index = 0;
    while ( primes[index] <= end )
    {
        if ( number % primes[index] == 0 )
        {
            number /= primes[index];
            if ( primes[index] == lastFactor )
                factors[numFactors-1].exponent++;
            else
            {
                factors[numFactors].number = primes[index];
                factors[numFactors].exponent = 1;
                lastFactor = primes[index];
                numFactors++;
            }
        }
```

```
        else
            index++;
    }

    if ( number != 1 )
    {
        factors[numFactors].number = number;
        factors[numFactors].exponent = 1;
        numFactors++;
    }

    divisors[numDivisors] = 1;
    numDivisors++;
    for ( index = 0; index < numFactors; index++ )
    {
        temp = numDivisors;
        thisDivisor = 1;
        for ( loop = 0; loop < factors[index].exponent; loop++ )
        {
            thisDivisor *= factors[index].number;
            for ( inner = 0; inner < temp; inner++ )
            {
                divisors[numDivisors] = thisDivisor*divisors[inner];
                numDivisors++;
            }
        }
    }

    /* sort divisors */
    for ( i = 0; i < numDivisors-1; i++ )
        for ( j = i+1; j < numDivisors; j++ )
            if ( divisors[i] > divisors[j] )
            {
                temp = divisors[i];
                divisors[i] = divisors[j];
                divisors[j] = temp;
            }

    return numDivisors;
}
```

Fixed Number of Terms Package

The following was used to check whether a number could be expanded using k terms, where k ranged from 2 to 6. This was used to calculated E(t).

```
/* fixed.c
 *
 *  minimize number of terms in an Egyptian fraction expansion
 *
 * Kevin Gong
 * Spring 1992
 *
 */


#include <stdio.h>
#include <math.h>
#include "headers/general.h"
#include "headers/vlint.h"
#include "headers/fractions.h"
```

```
#define TRUE   1
#define MAX_UNSIGNED_INT    (unsigned long) (~1)


void FindDivisors(unsigned long number, unsigned long divisors[],
                int *numDivisors);
boolean RelativelyPrime(unsigned long x, unsigned long y);
void ShortestConstruction(unsigned long goalNum, unsigned long goalDen,
                       Expansion *shortExp);
boolean TwoExpandable(unsigned long goalNum, unsigned long goalDen,
                    VLInt *x, VLInt *y);
boolean NExpandable(unsigned long goalNum, unsigned long goalDen, int n, VLInt x[]);

void MakeLowestTerms(unsigned long *numer, unsigned long *denom);
unsigned long ExpandFraction(unsigned long goalNum, unsigned long goalDen);
void PrintExpansion(Expansion *exp);
unsigned long GenerateDenoms(unsigned long k, unsigned long index);
void FindCounterExamples(void);


unsigned long notExpress[10][599];
int numNot[10];

void main(unsigned long argc, char **argv)
{
    unsigned long goalNum;
    unsigned long goalDen;
    register unsigned long index;
    unsigned long maxDenom;
    unsigned long minDenom;
    unsigned long temp1, temp2;
    unsigned long best;
    unsigned long mask;
    unsigned long total;
    int numTerms;
    int termCount[5];
    int count;

    for ( index = 0; index < 10; index++ )
    {
       numNot[index] = 0;
    }

    if ( argc == 1 )
    {
       FindCounterExamples();
    }
    else if ( argc == 3 )
    {
       goalNum = atoi(argv[1]);
       goalDen = atoi(argv[2]);
       MakeLowestTerms(&goalNum, &goalDen);
       ExpandFraction(goalNum, goalDen);
    }
    else
    {
       if ( argc == 2 )
       {
           fprintf(stderr, "Minimum denominator:  ");
           fscanf(stdin, "%u", &minDenom);
           fprintf(stderr, "Maximum denominator:  ");
           fscanf(stdin, "%u", &maxDenom);
       }
```

```
        else if ( argc == 4 )
        {
            minDenom = atoi(argv[2]);
            maxDenom = atoi(argv[3]);
        }
        goalNum = atoi(argv[1]);
        total = 0;

        termCount[2] = 0;     termCount[3] = 0;
        for ( goalDen = minDenom; goalDen <= maxDenom; goalDen++ )
        {
            temp1 = goalNum;
            temp2 = GenerateDenoms(goalNum, goalDen);
            MakeLowestTerms(&temp1, &temp2);
            if ( temp1 == goalNum )
            {
               numTerms = ExpandFraction(goalNum, temp2);
               termCount[numTerms]++;
            }
        }

        for ( index = 5; index < 10; index++ )
        {
            if ( numNot[index] == 0 )
               continue;
            fprintf(stderr, "NOT EXPRESSIBLE in %d TERMS:  %ld/...\n", index,
                   goalNum);
            for ( count = 0; count < numNot[index]; count++ )
               fprintf(stderr, "%ld\t", notExpress[index][count]);
            fprintf(stderr, "\n");
        }
    }
}

unsigned long ExpandFraction(unsigned long goalNum, unsigned long goalDen)
{
    unsigned long fib, gol, ble;
    unsigned long best;
    unsigned long mask = 0;
    Expansion fibExp, golExp, bleExp, shortExp;
    int numTerms;

    fprintf(stdout, "-> Expansion of %u/%u\n", goalNum, goalDen);

    ShortestConstruction(goalNum, goalDen, &shortExp);
    PrintExpansion(&shortExp);

    numTerms = shortExp.numTerms;
    return numTerms;
}


void MakeLowestTerms(unsigned long *numer, unsigned long *denom)
{
    register unsigned long index;

    index = 2;
    while ( index <= *numer )
    {
        if ( (*numer % index == 0) && (*denom % index == 0) )
        {
            *numer /= index;
            *denom /= index;
        }
```

```
        else
            index++;
    }
}


void InitExpansion(Expansion *exp)
{
    exp->numTerms = 0;
    exp->maxDenom = 0;
}


void AddExpansionTerm(Expansion *exp, VLInt *term)
{
    register unsigned long i, j;
    register unsigned long temp;

    bcopy(term, &exp->denoms[exp->numTerms], sizeof(VLInt));
    exp->numTerms++;
}


void PrintExpansion(Expansion *exp)
{
    register unsigned long index;
    char temp[100];

    for ( index = 0; index < exp->numTerms-1; index++ )
        fprintf(stdout, "1/%s + ", VLIntPrint(&exp->denoms[index], temp));
    fprintf(stdout, "1/%s (%d)\n",
            VLIntPrint(&exp->denoms[exp->numTerms-1], temp),
            exp->numTerms);
}


void ShortestConstruction(unsigned long goalNum, unsigned long goalDen,
                        Expansion *shortExp)
{
    VLInt w, x, y, z;
    VLInt temp;
    VLInt terms[10];
    register int n, index;

    InitExpansion(shortExp);
    if ( goalNum == 1 )
    {
        AddExpansionTerm(shortExp, VLIntCreate(&temp, goalDen));
        return;
    }

    for ( n = 2; n < 10; n++ )
    {
        fprintf(stdout, "Testing for size %d expansion\n", n);
        if ( NExpandable(goalNum, goalDen, n, terms) )
        {
            for ( index = 0; index < n; index++ )
                AddExpansionTerm(shortExp, &terms[index]);
            return;
        }
        else
        {
            notExpress[n][numNot[n]] = goalDen;
            numNot[n]++;
```

```
        }
    }

    exit(0);
}


boolean NExpandable(unsigned long goalNum, unsigned long goalDen, int n, VLInt x[])
{
    if ( n == 2 )
        return TwoExpandable(goalNum, goalDen, &x[0], &x[1]);
    else
    {
        unsigned long first;
        unsigned long restNum, restDen;
        unsigned long nthRec;

        nthRec = n*goalDen/goalNum;
        first = 1+(goalDen/goalNum);
        while ( first < nthRec )
        {
            restDen = goalDen*first;
            restNum = goalNum*first - goalDen;
            MakeLowestTerms(&restNum, &restDen);

            if ( NExpandable(restNum, restDen, n-1, x) )
            {
                VLIntCreate(&x[n-1], first);
                return TRUE;
            }

            first++;
        }

        return FALSE;
    }
}


void FindDivisors(unsigned long number, unsigned long divisors[],
                int *numDivisors)
{
    register unsigned long index;

    *numDivisors = 0;

    for ( index = 1; index <= (unsigned long)sqrt((double)number); index++ )
    {
        if ( number % index == 0 )
        {
            divisors[*numDivisors] = index;
            (*numDivisors)++;
            divisors[*numDivisors] = number/index;
            (*numDivisors)++;
        }
    }
}


boolean TwoExpandable(unsigned long goalNum, unsigned long goalDen,
                    VLInt *x, VLInt *y)
{
    register unsigned long P, Q;
    register unsigned long mult;
```

```
    register unsigned long total;
    unsigned long divisors[999];
    int numDivisors;
    int ptr1, ptr2;
    VLInt temp1, temp2;

    FindDivisors(goalDen, divisors, &numDivisors);

    for ( ptr1 = 0; ptr1 < numDivisors-1; ptr1++ )
    {
       for ( ptr2 = ptr1+1; ptr2 < numDivisors; ptr2++ )
       {
           P = divisors[ptr1];      Q = divisors[ptr2];
           if ( RelativelyPrime(P, Q) &&
                (goalDen % P == 0) && (goalDen % Q == 0) &&
                ((P + Q) % goalNum == 0) )
           {
              mult = (P+Q)/goalNum;
              VLIntCreate(&temp1, mult);
              VLIntCreate(&temp2, goalDen/P);
              VLIntMultiply(&temp1, &temp2, x);
              VLIntCreate(&temp2, goalDen/Q);
              VLIntMultiply(&temp1, &temp2, y);
              return TRUE;
           }
       }
    }

    return FALSE;
}


boolean RelativelyPrime(unsigned long x, unsigned long y)
{
    register unsigned long index = 2;
    register unsigned long end;

    if ( (x == 1) || (y == 1) )
       return TRUE;

    if ( x == y )
       return FALSE;
    else if ( (x < y) && (y % x == 0) )
       return FALSE;
    else if ( (y < x) && (x % y == 0) )
       return FALSE;

    end = (x < y) ? (unsigned long)sqrt((double)x) :
                  (unsigned long)sqrt((double)y);

    while ( index <= end )
    {
       if ( (x % index == 0) && (y % index == 0) )
           return FALSE;

       index++;
    }

    return TRUE;
}


unsigned long GenerateDenoms(unsigned long k, unsigned long index)
{
```

```
    unsigned long mult;
    unsigned long denom;

    if ( k == 4 )
    {
       mult = index/6;
       denom = 840*(mult+1);
       switch(index%6)
       {
           case 0: return denom+1*1;
           case 1: return denom+11*11;
           case 2: return denom+13*13;
           case 3: return denom+17*17;
           case 4: return denom+19*19;
           case 5: return denom+23*23;
       }
    }
    else
       return index;
}


void FindCounterExamples()
{
    register int numTerms;
    unsigned long goalNum, goalDen;
    VLInt x[10];
    unsigned long temp1, temp2;

    goalNum = 7;
    goalDen = 8;
    for ( numTerms = 3; numTerms < 10; numTerms++ )
    {
       while ( NExpandable(goalNum, goalDen, numTerms, x) )
       {
           goalDen++;
           temp2 = goalDen;    temp1 = goalNum;
           MakeLowestTerms(&temp1, &temp2);
           while ( temp1 != goalNum )
           {
              goalDen++;  temp2 = goalDen; temp1 = goalNum;
              MakeLowestTerms(&temp1, &temp2);
           }

           if ( goalDen > goalNum+40 )
           {
              goalNum++;
              fprintf(stdout, "Testing %ld/...\n", goalNum);
              goalDen = goalNum+1;
           }
       }
       fprintf(stderr, "Not expressible in %d terms:  %ld/%ld\n",
              numTerms, goalNum, goalDen);
    }
}
```

VLInt Package

The following is the VLInt (Very Large Integer) package used by the other programs to manipulate positive integers with many digits.

```
/* vlint.h
 *
 *  Very Large Integers
 *
 * Handles positive integers of any length
 *
 * This package provides routines to manipulate positive integers of any length
 * Functions include add, subtract, multiply, divide, modulo, and comparison
 *
 * Kevin Gong
 * Spring 1992
 *
 */

#ifndef VLINT_INCLUDED
#define VLINT_INCLUDED

#include "general.h"


#define Min(a,b)    ((a < b) ? a : b)

typedef struct VLIntStruct
{
    char *digits;       /* digits[0] = lsb */
    int numDigits;
    int maxDigits;
} VLInt;

VLInt *VLIntCreate(VLInt *vlint, unsigned long number);
char *VLIntPrint(VLInt *vlint, char *string);
boolean VLIntEquality(VLInt *src1, VLInt *src2);
boolean VLIntLessThan(VLInt *src1, VLInt *src2);
VLInt *VLIntAdd(VLInt *src1, VLInt *src2, VLInt *dest);
VLInt *VLIntSubtract(VLInt *src1, VLInt *src2, VLInt *dest);
VLInt *VLIntMultiply(VLInt *src1, VLInt *src2, VLInt *dest);
VLInt *VLIntDivide(VLInt *src1, VLInt *src2, VLInt *dest);
VLInt *VLIntMod(VLInt *src1, VLInt *src2, VLInt *dest);
VLInt *VLIntMultiplyDigit(VLInt *src1, char digit, VLInt *dest);
void VLIntCopy(VLInt *src, VLInt *dest);
VLInt *VLIntShiftLeft(VLInt *src, int exponent, VLInt *dest);
VLInt *VLIntAddDigit(VLInt *src, char digit, VLInt *dest);

#endif VLINT_INCLUDED
```

```
/* vlint.c
 *
 *  Very Large Integers
 *
 * Handles positive integers of any length
 *
 * Kevin Gong
 * Spring 1992
 *
 */


/*--------------*
 * HEADER FILES *
```

```
 *--------------*/
#include <math.h>
#include <stdio.h>
#include <assert.h>
#include "headers/general.h"
#include "headers/vlint.h"

char VLIntDivideResultDigit(VLInt *src1, VLInt *src2, int place, VLInt table[],
                           char left, char right);


/*----------------*
 * Create a VLInt *
 *----------------*/
VLInt *VLIntCreate(VLInt *vlint, unsigned long number)
{
    register int pointer = 0;
    register char *digits;
    int size;

    if ( number < 10 )
        size = 1;
    else
        size = (int) log10((double)number) + 1;

    if ( (vlint->digits == NULL) || (size > vlint->maxDigits) )
    {
        if ( vlint->digits != NULL )
            free(vlint->digits);
        size *= 2;
        vlint->digits = (char *) malloc(size*sizeof(char));
        vlint->maxDigits = size;
    }

    digits = vlint->digits;

    while ( number != 0 )
    {
        digits[pointer] = number % 10;
        number /= 10;
        pointer++;
    }

    vlint->numDigits = pointer;

    assert(vlint->numDigits <= vlint->maxDigits);

    return vlint;
}


/*---------------*
 * Print a VLInt *
 *---------------*/
char *VLIntPrint(VLInt *vlint, char *string)
{
    register int pointer;
    register char *digits = vlint->digits;
    register int numDigits = vlint->numDigits;

    for ( pointer = 0; pointer < numDigits; pointer++ )
        string[pointer] = digits[numDigits-pointer-1] + '0';

    string[pointer] = '\0';
```

```
    return string;
}


/*--------------*
 * src1 == src2 *
 *--------------*/
/*---------------------------------------*
 * Simple equality test.  Takes O(n) time *
 *---------------------------------------*/
boolean VLIntEquality(VLInt *src1, VLInt *src2)
{
    register int index;
    char *digits1 = src1->digits;
    char *digits2 = src2->digits;

    if ( src1->numDigits != src2->numDigits )
        return FALSE;

    for ( index = src1->numDigits-1; index >= 0; index-- )
        if ( digits1[index] != digits2[index] )
            return FALSE;

    return TRUE;
}


/*-------------*
 * src1 < src2 *
 *-------------*/
/*-----------------------------------------*
 * Simple inequality test.  Takes O(n) time *
 *-----------------------------------------*/
boolean VLIntLessThan(VLInt *src1, VLInt *src2)
{
    register int index;
    char *digits1 = src1->digits;
    char *digits2 = src2->digits;

    if ( src1->numDigits < src2->numDigits )
        return TRUE;
    else if ( src1->numDigits > src2->numDigits )
        return FALSE;

    for ( index = src1->numDigits-1; index >= 0; index-- )
        if ( digits1[index] < digits2[index] )
            return TRUE;
        else if ( digits1[index] > digits2[index] )
            return FALSE;

    return FALSE;
}


/*--------------------*
 * dest = src1 - src2 *
 *--------------------*/
/*------------------------------*
 * Subtraction.  Takes O(n) time *
 *------------------------------*/
VLInt *VLIntSubtract(VLInt *src1, VLInt *src2, VLInt *dest)
{
    char *digits1, *digits2, *destDigits, *restDigits;
    register int pointer;
```

```
    register int firstEnd, secondEnd;
    register char borrow = 0;
    char temp;
    int size;

    digits1 = src1->digits;
    digits2 = src2->digits;

    if ( src1->numDigits < src2->numDigits )
    {
        fprintf(stdout, "Negative output in subtract\n");
        exit(-1);
    }
    else
    {
        firstEnd = src2->numDigits;
        secondEnd = src1->numDigits;
        restDigits = digits1;

        size = src1->numDigits;
        if ( (dest->digits == NULL) || (size > dest->maxDigits) )
        {
            if ( dest->digits != NULL )
                free(dest->digits);
            size *= 2;
            dest->digits = (char *) malloc(size*sizeof(char));
            dest->maxDigits = size;
        }
        destDigits = dest->digits;
    }

    for ( pointer = 0; pointer < firstEnd; pointer++ )
    {
        temp = digits1[pointer] - digits2[pointer] - borrow;
        if ( temp < 0 )
        {
            destDigits[pointer] = temp+10;
            borrow = 1;
        }
        else
        {
            destDigits[pointer] = temp;
            borrow = 0;
        }
    }

    for ( ; pointer < secondEnd; pointer++ )
    {
        temp = restDigits[pointer] - borrow;
        if ( temp < 0 )
        {
            destDigits[pointer] = temp+10;
            borrow = 1;
        }
        else
        {
            destDigits[pointer] = temp;
            borrow = 0;
        }
    }

    if ( borrow == 1 )
    {
        fprintf(stdout, "Negative output in subtract\n");
```

```
        exit(-1);
    }

    pointer = secondEnd-1;
    while ( (pointer >= 0) && (destDigits[pointer] == 0) )
        pointer--;

    dest->numDigits = pointer+1;

    return dest;
}


/*--------------------*
 * dest = src1 + src2 *
 *--------------------*/
/*----------------------------*
 * Addition.  Takes O(n) time *
 *----------------------------*/
VLInt *VLIntAdd(VLInt *src1, VLInt *src2, VLInt *dest)
{
    char *digits1, *digits2, *destDigits, *restDigits;
    register int pointer;
    register int firstEnd, secondEnd;
    register char carry = 0;
    int size;

    digits1 = src1->digits;
    digits2 = src2->digits;
    if ( src1->numDigits < src2->numDigits )
    {
        firstEnd = src1->numDigits;
        secondEnd = src2->numDigits;
        restDigits = digits2;
        size = src2->numDigits+1;
    }
    else
    {
        firstEnd = src2->numDigits;
        secondEnd = src1->numDigits;
        restDigits = digits1;
        size = src1->numDigits+1;
    }

    if ( (dest->digits == NULL) || (size > dest->maxDigits) )
    {
        if ( dest->digits != NULL )
            free(dest->digits);
        size *= 2;
        dest->digits = (char *) malloc(size*sizeof(char));
        dest->maxDigits = size;
    }
    destDigits = dest->digits;

    for ( pointer = 0; pointer < firstEnd; pointer++ )
    {
        destDigits[pointer] = digits1[pointer] + digits2[pointer] + carry;
        if ( destDigits[pointer] > 9 )
        {
            destDigits[pointer] -= 10;
            carry = 1;
        }
        else
            carry = 0;
```

```
    }

    for ( ; pointer < secondEnd; pointer++ )
    {
        destDigits[pointer] = restDigits[pointer] + carry;
        if ( destDigits[pointer] > 9 )
        {
            destDigits[pointer] -= 10;
            carry = 1;
        }
        else
            carry = 0;
    }

    if ( carry == 1 )
    {
        destDigits[pointer] = carry;
        dest->numDigits = secondEnd+1;
    }
    else
        dest->numDigits = secondEnd;

    assert(dest->numDigits <= dest->maxDigits);

    return dest;
}


/*--------------------*
 * dest = src1 * src2 *
 *--------------------*/
/*--------------------------------------------------------------------*
 * Multiplication.  Create a table of 0*multiplicand, 1*multiplicand, ... *
 * 9*multiplicand.  Takes O(n*n) due to n additions.                  *
 *--------------------------------------------------------------------*/
VLInt *VLIntMultiply(VLInt *src1, VLInt *src2, VLInt *dest)
{
    VLInt *multiplier, *multiplicand;
    char *multiplierDigits;
    boolean computed[10];
    static VLInt temp, temp2;
    static VLInt table[10];
    static boolean init = FALSE;
    register int pointer;
    register int index;

    for ( index = 0; index < 10; index++ )
    {
        computed[index] = FALSE;
    }

    if ( ! init )
    {
        temp.digits = NULL;    temp2.digits = NULL;
        for ( index = 0; index < 10; index++ )
            table[index].digits = NULL;
        init = TRUE;
    }

    if ( src1->numDigits < src2->numDigits )
    {
        multiplier = src1;
        multiplicand = src2;
        multiplierDigits = src1->digits;
```

```
    }
    else
    {
        multiplier = src2;
        multiplicand = src1;
        multiplierDigits = src2->digits;
    }

    VLIntCreate(dest, (unsigned long) 0);
    for ( pointer = 0; pointer < multiplier->numDigits; pointer++ )
    {
        if ( multiplierDigits[pointer] == 0 )
            continue;

        if ( ! computed[multiplierDigits[pointer]] )
        {
            if ( multiplierDigits[pointer] == 1 )
                VLIntCopy(multiplicand, &table[multiplierDigits[pointer]]);
            else
                    VLIntMultiplyDigit(multiplicand, multiplierDigits[pointer],
                                        &table[multiplierDigits[pointer]]);
        }

        VLIntShiftLeft(&table[multiplierDigits[pointer]], pointer, &temp);
        VLIntAdd(dest, &temp, &temp2);
        VLIntCopy(&temp2, dest);
    }

    assert(dest->numDigits <= dest->maxDigits);

    return dest;
}

/*-------------------*
 * dest = src1*digit *
 *-------------------*/
VLInt *VLIntMultiplyDigit(VLInt *src1, char digit, VLInt *dest)
{
    char *destDigits;
    char *digits1 = src1->digits;
    int pointer;
    int carry = 0;
    int temp;
    int size;

    size = src1->numDigits+1;

    if ( (dest->digits == NULL) || (size > dest->maxDigits) )
    {
        if ( dest->digits != NULL )
            free(dest->digits);
        size *= 2;
        dest->digits = (char *) malloc(size*sizeof(char));
        dest->maxDigits = size;
    }
    destDigits = dest->digits;

    for ( pointer = 0; pointer < src1->numDigits; pointer++ )
    {
        temp = digit*digits1[pointer] + carry;
        if ( temp > 9 )
        {
            destDigits[pointer] = temp % 10;
            carry = temp/10;
```

```
        }
        else
        {
            destDigits[pointer] = temp;
            carry = 0;
        }
    }


    if ( carry != 0 )
    {
        destDigits[pointer] = carry;
        pointer++;
    }

    dest->numDigits = pointer;

    assert(dest->numDigits <= dest->maxDigits);

    return dest;
}


/*--------------------*
 * dest = src1 / src2 *
 *--------------------*/
/*-------------------------------------------------------------------*
 * Division.  Create a table of 0*divisor, 1*divisor, ..., 9*divisor. *
 * Takes O(n*n), but uses binary search to find digits of answer.     *
 *-------------------------------------------------------------------*/
VLInt *VLIntDivide(VLInt *src1, VLInt *src2, VLInt *dest)
{
    static VLInt rest;
    static VLInt table[10];
    static VLInt temp, temp2;
    static boolean init = FALSE;
    register char index;
    register int place;
    register char digit;
    int maxPlace = 0;
    int size;

    if ( ! init )
    {
        temp.digits = NULL;     temp2.digits = NULL;    rest.digits = NULL;
        for ( index = 0; index < 10; index++ )
            table[index].digits = NULL;

        init = TRUE;
    }

    VLIntCopy(src1, &rest);
    for ( index = 0; index < 10; index++ )
        VLIntMultiplyDigit(src2, index, &table[index]);

    place = rest.numDigits - src2->numDigits;
    size = place+1;
    if ( (dest->digits == NULL) || (size > dest->maxDigits) )
    {
        if ( dest->digits != NULL )
            free(dest->digits);
        size *= 2;
        dest->digits = (char *) malloc(size*sizeof(char));
        dest->maxDigits = size;
```

```
    }

    while ( VLIntLessThan(src2, &rest) )
    {
        digit = VLIntDivideResultDigit(&rest, src2, place, table, 0, 9);
        dest->digits[place] = digit;
        if ( digit != 0 )
        {
            VLIntShiftLeft(&table[digit], place, &temp);
            VLIntSubtract(&rest, &temp, &temp2);
            VLIntCopy(&temp2, &rest);
            maxPlace = (place > maxPlace) ? place : maxPlace;
        }
        place--;
    }

    while ( place >= 0 )
    {
        dest->digits[place] = 0;
        place--;
    }

    dest->numDigits = maxPlace+1;

    return dest;
}


char VLIntDivideResultDigit(VLInt *src1, VLInt *src2, int place, VLInt table[],
                            char left, char right)
{
    VLInt *temp;
    char middle;
    int numDigits;
    char *digits1, *digits2;
    register int index;

    if ( left == right )
        return left;

    middle = (left+right+1)/2;
    temp = &table[middle];

    numDigits = src1->numDigits-place;

    if ( numDigits < temp->numDigits )
        return VLIntDivideResultDigit(src1, src2, place, table, left, middle-1);
    else if ( numDigits > temp->numDigits )
        return VLIntDivideResultDigit(src1, src2, place, table, middle, right);

    digits1 = src1->digits;
    digits2 = temp->digits;
    for ( index = src1->numDigits-1; index >= place; index-- )
        if ( digits1[index] < digits2[index-place] )
            return VLIntDivideResultDigit(src1, src2, place, table,
                                          left, middle-1);
        else if ( digits1[index] > digits2[index-place] )
            return VLIntDivideResultDigit(src1, src2, place, table,
                                          middle, right);

    return middle;
}
```

```
/*--------------------*
 * dest = src1 % src2 *
 *--------------------*/
/*-----------------------------------------*
 * Modulo.  Basically the same as division. *
 *-----------------------------------------*/
VLInt *VLIntMod(VLInt *src1, VLInt *src2, VLInt *dest)
{
    static VLInt table[10];
    register char index;
    register int place;
    register char digit;
    static VLInt temp, temp2;
    int maxPlace = 0;
    static boolean init = FALSE;

    if ( ! init )
    {
        temp.digits = NULL;    temp2.digits = NULL;
        for ( index = 0; index < 10; index++ )
            table[index].digits = NULL;
        init = TRUE;
    }

    VLIntCopy(src1, dest);
    for ( index = 0; index < 10; index++ )
        VLIntMultiplyDigit(src2, index, &table[index]);

    place = dest->numDigits - src2->numDigits;
    while ( VLIntLessThan(src2, dest) )
    {
        digit = VLIntDivideResultDigit(dest, src2, place, table, 0, 9);
        if ( digit != 0 )
        {
            VLIntShiftLeft(&table[digit], place, &temp);
            VLIntSubtract(dest, &temp, &temp2);
            VLIntCopy(&temp2, dest);
            maxPlace = (place > maxPlace) ? place : maxPlace;
        }
        place--;
    }

    return dest;
}


void VLIntCopy(VLInt *src, VLInt *dest)
{
    if ( (dest->digits == NULL) || (src->numDigits > dest->maxDigits) )
    {
        if ( dest->digits != NULL )
            free(dest->digits);
        dest->digits = (char *) malloc(2*src->numDigits*sizeof(char));
        dest->maxDigits = 2*src->numDigits;
    }
    bcopy(src->digits, dest->digits, src->numDigits*sizeof(char));
    dest->numDigits = src->numDigits;
}


/*------------------------*
 * dest = src*10^exponent *
 *------------------------*/
VLInt *VLIntShiftLeft(VLInt *src, int exponent, VLInt *dest)
```

```
{
    char *destDigits;
    int size;

    size = exponent + src->numDigits;

    if ( (dest->digits == NULL) || (size > dest->maxDigits) )
    {
        if ( dest->digits != NULL )
            free(dest->digits);
        dest->digits = (char *) malloc(size*2*sizeof(char));
        dest->maxDigits = size*2;
    }
    destDigits = dest->digits;

    bcopy(src->digits, &destDigits[exponent], src->numDigits*sizeof(char));
    bzero(destDigits, exponent*sizeof(char));

    dest->numDigits = size;

    assert(dest->numDigits <= dest->maxDigits);

    return dest;
}


/*--------------------*
 * dest = src + digit *
 *--------------------*/
VLInt *VLIntAddDigit(VLInt *src, char digit, VLInt *dest)
{
    char *digits, *destDigits;
    register int pointer;
    register char carry;
    int size;

    if ( src->numDigits == 0 )
    {
        VLIntCreate(dest, (unsigned long) digit);

        assert(dest->numDigits <= dest->maxDigits);
        return dest;
    }

    digits = src->digits;
    size = src->numDigits+1;

    if ( (dest->digits == NULL) || (size > dest->maxDigits) )
    {
        if ( dest->digits != NULL )
            free(dest->digits);
        size *= 2;
        dest->digits = (char *) malloc(size*sizeof(char));
        dest->maxDigits = size;
    }
    destDigits = dest->digits;

    carry = digit;
    for ( pointer = 0; pointer < src->numDigits; pointer++ )
    {
        destDigits[pointer] = digits[pointer] + carry;
        if ( destDigits[pointer] > 9 )
        {
            destDigits[pointer] -= 10;
```

```
            carry = 1;
        }
        else
        {
            carry = 0;
            bcopy(&digits[pointer+1], &destDigits[pointer+1],
                  (src->numDigits-(pointer+1))*sizeof(char));
            break;
        }
    }

    if ( carry == 1 )
    {
        destDigits[pointer] = carry;
        dest->numDigits = src->numDigits+1;
    }
    else
        dest->numDigits = src->numDigits;

    assert(dest->numDigits <= dest->maxDigits);

    return dest;
}
```