Ancient Egyptian Numbers: A CS-Complete Example

Timothy S. Gegg-Harrison
Department of Computer Science
Winona State University
Winona, MN 55987
tsg@wind.winona.msus.edu

Abstract

A critical piece of any successful curriculum is a robust example that permeates the key concepts of the field. For computer science, we refer to such an example as CS-complete. A good CS-complete example is applicable in CS1, CS2, and Discrete Mathematics. Approximately 4000 years ago, the ancient Egyptians used a numbering system that serves as a prototype CS-complete example. In this paper, we outline the use of the Egyptian numbering system as an example that naturally extends through CS1, CS2, and Discrete Mathematics.

1 Introduction

Because of the dependency of computing on discrete mathematics, the Computing Curricula 2001 Task Force has proposed that *discrete structures* be added as a separate knowledge area [2]. Although computer scientists understand the importance of discrete mathematics to the foundations of their field, computer science students do not always see the relevance. Thus, it is important to find a way to show students the relevance via a unifying example. Some of the key topics included in the CS1/CS2 curriculum are representation, problem solving, algorithms, recursion, induction, and data structures. We refer to any example that includes each of these topics as CS-complete.

Approximately 4000 years ago, the ancient Egyptians used a numbering system [1] that makes an interesting example that is CS-complete. In this paper, we outline the use of this example in our CS1, CS2, and Discrete Mathematics classes.

2 CS1/CS2 Lesson

In our CS1 class, students are introduced to representation and problem solving using the Egyptian numbering system. Two aspects of the Egyptian numbering system are presented to our CS1 students. We begin with problem solving using Egyptian multiplication on integers followed by the representation of rational numbers using Egyptian fractions.

The beauty of Egyptian multiplication is that it provides a drastically different approach to multiplication while at the same time providing a natural bridge from the decimal numbering system to the binary numbering system. The basic idea behind Egyptian multiplication is to repeatedly double the multiplicand while at the same time halving the multiplier until it eventually reaches 1. Note that this algorithm is based on the fact that $m \times n = \frac{m}{2} \times 2n$. This works fine as long as m is even in which case $\frac{m}{2}$ is an integer. If m is odd, on the other hand, then $\frac{m}{2}$ is not an integer. In order to prevent introducing non-integers into the computation, it is necessary to use the expression $\left\lfloor \frac{m}{2} \right\rfloor \times 2n$ (where $\left\lfloor \frac{m}{2} \right\rfloor$ is the whole number of times that 2 will divide into m). When m is even, $\left\lfloor \frac{m}{2} \right\rfloor \times 2n = m \times n$. When m is odd, on the other hand, $\left\lfloor \frac{m}{2} \right\rfloor \times 2n \neq m \times n$. In fact, $\left\lfloor \frac{m}{2} \right\rfloor \times 2n$ is precisely n less than $m \times n$ whenever m is odd. Thus, the product of two integers is simply the sum of all the "doubled" multiplicands for which the corresponding "halved" multiplier is odd.

After giving a brief history of the Rhind Papyrus and the Rosetta Stone which unveiled the mathematical secrets of the ancient Egyptians and their numbering system, we provide an example of Egyptian multiplication on 33 × 26 using the following table.

"Halved" Multiplicand	"Doubled" Multiplier	Remainder
33	***26***	***26***
16	52	0
8	104	0
4	208	0
2	416	0
1	***832***	***832***

Students can compute the product by summing the values in the righthand column. Thus, the expression $33 \times 26 = (26 + 832) = 858$. The algorithm for multiplying the number m by the number n producing p is as follows.

Egyptian Multiplication Algorithm

The $\times$ operation for the expression $m \times n$ is p as defined in the following cases:

Case 1: $(m = 0)$
p is 0

Case 2: $(m = 1)$
p is n

Case 3: $(m > 1)$ and $((m \bmod 2) = 0)$
p is $(m \text{ div } 2) \times (n + n)$

Case 4: $(m > 1)$ and $((m \bmod 2) = 1)$
p is $((m \text{ div } 2) \times (n + n)) + n$

After studying the Egyptian Multiplication Algorithm and building on their understanding of recursion, our CS1 students are introduced to the idea of alternate representations with the binary numbering system. Students see that it is possible to convert decimal numbers into binary by repeatedly dividing the decimal number by 2 and maintaining the remainders. We show the conversion of the decimal number 75 to binary using the same basic table that we used for Egyptian multiplication.

Decimal Number	Binary Position	Binary Bit (Remainder)
75	1	1
37	2	1
18	4	0
9	8	1
4	16	0
2	32	0
1	64	1

This table is used to argue that the striking similarity between Egyptian multiplication and decimal to binary conversion is not a coincidence. Students see that, in fact, it is the same process that is taking place. This can be made even more clear by looking at decimal to binary conversion from a slightly different perspective. First of all, students are reminded that $2^0 = 1$ and therefore 75 can be rewritten as $75 \times 1 = 75 \times 2^0$. Following the same approach taken by the ancient Egyptians, 75×2^0 can be rewritten as $(37 \times 2^1) + 2^0$ since 75 is an odd number so (37×2^1) is 2^0 less than 75×2^0. Furthermore, 37×2^1 can be rewritten as $(18 \times 2^2) + 2^1$ since 37 is an odd number so (18×2^2) is 2^1 less than 37×2^1. This gives the following decomposition of 75:

$$
\begin{aligned}
75 = \quad 75 \times 1 \quad &= 75 \times 2^0 \\
&= (37 \times 2^1) + 2^0 \\
&= ((18 \times 2^2) + 2^1) + 2^0 \\
&= ((9 \times 2^3) + 2^1) + 2^0 \\
&= (((4 \times 2^4) + 2^3) + 2^1) + 2^0 \\
&= (((2 \times 2^5) + 2^3) + 2^1) + 2^0 \\
&= (((1 \times 2^6) + 2^3) + 2^1) + 2^0 \\
&= ((2^6 + 2^3) + 2^1) + 2^0
\end{aligned}
$$

Decimal to Binary Conversion Algorithm

The *binary equivalent* of the decimal number d is b as defined in the following cases:

Case 1: $(d = 0)$
b is 0

Case 2: $(d = 1)$
b is d

Case 3: $(d > 1)$ and $((d \bmod 2) = 0)$
b is (the *binary equivalent* of $(d \text{ div } 2)0$

Case 4: $(d > 1)$ and $((d \bmod 2) = 1)$
b is (the *binary equivalent* of $(d \text{ div } 2)1$

The binary numbering system is important for CS1 students because in addition to providing them with an alternate representation for number, it provides them with the practical knowledge of the actual numbering system that used in most computational devices. However, the binary numbering system merely provides a different base rather than a drastically different representation of number. Extending the binary numbering system to include integers by introducing complements and further to include rational numbers via floating point representation is important, but still very intuitive.

Egyptian fractions, on the other hand, provide a representation of rational numbers that is anything but intuitive to most first-year CS students. In addition to

providing another example of a recursive algorithm, the conversion of a standard fraction into the sum of distinct unit fractions (i.e., fractions with a numerator of 1) provides an example of a greedy algorithm that uses linked lists. Thus, Egyptian fraction conversion makes an ideal programming project for CS2 students.

Egyptian Fraction Conversion Algorithm

The *Egyptian fraction equivalent* of the fraction $\frac{p}{q}$ is f as defined in the following cases:

Case 1: $(p = 0)$

f is 0

Case 2: $(p \neq 0)$

f is $\frac{1}{k}$ + (the *Egyptian fraction equivalent* of $\left(\frac{p}{q} - \frac{1}{k}\right)$) where k is the smallest positive integer such that $\frac{p}{q} \geq \frac{1}{k}$

As an example, students are asked to consider the application of the Egyptian Fraction Conversion Algorithm to $\frac{6}{7}$. The largest unit fraction that is less than $\frac{6}{7}$ is $\frac{1}{2}$. Thus, we reduce the problem to the conversion of the fraction $\frac{6}{7} - \frac{1}{2} = \frac{5}{14}$. The largest unit fraction that is less than $\frac{5}{14}$ is $\frac{1}{3}$ leaving the fraction $\frac{5}{14} - \frac{1}{3} = \frac{1}{42}$. Since $\frac{1}{42}$ is a unit fraction, the algorithm terminates with $\frac{6}{7} = \frac{1}{2} + \frac{1}{3} + \frac{1}{42}$.

The Egyptian Fraction Conversion Algorithm can be used to convert both proper and improper fractions, but it will produce duplicate unit fractions on improper fractions. Students are shown that it is possible to remove duplicates by applying the following identity:

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

They are shown that an alternative way to remove duplicates is not allow them to be created in the first place by modifying the algorithm to force k to be unique in Case 2. From a programming perspective, Egyptian fractions are very robust. Students create an instantiable class for rational numbers that contains an inner class to define recursive nodes that make up a linked list (or a more sophisticated data structure like a deterministic skip list or AVL tree) of unit fractions.

Egyptian Fraction Class

```
import java.math.BigInteger;
public class EgyptianFraction extends Object
             implements Cloneable, Comparable {

 private class UnitFraction {
  private BigInteger denom;
  private UnitFraction next;
  private UnitFraction(BigInteger d, UnitFraction n) {
    denom = d;
    next = n;
  }
 }

 private UnitFraction value = null;

 public EgyptianFraction(BigInteger p, BigInteger q) {
   BigInteger k = new BigInteger("2");
   while (p.compareTo(BigInteger.ZERO) != 0) {
     while (((p.multiply(k)).compareTo(q)) < 0)
       k = k.add(BigInteger.ONE);
     value = new UnitFraction(k, value);
     p = (p.multiply(k)).subtract(q);
     q = q.multiply(k);
     k = k.add(BigInteger.ONE);
   }
 }
 public boolean equals(Object o) {
   ⋮
 }
 public int compareTo(Object o) {
   ⋮
 }
 public Object clone() {
   ⋮
 }
 public EgyptianFraction add(EgyptianFraction e) {
   ⋮
 }
 public EgyptianFraction multiply(EgyptianFraction e) {
   ⋮
 }
 ⋮
}
```

The Java implementation of the `EgyptianFraction` class highlights many key programming concepts. In addition to giving students practice with inheritance by extending the `Object` class and implementing the `Comparable` and `Cloneable` interfaces, students also encounter computational limits head-on. To avoid overflowing the program stack, students need to replace the tail-recursive algorithm with an implementation using nested `while` loops. Furthermore, because unit fractions quickly become too large for the integer primitive types `int` and `long`, students are forced to use Java's `BigInteger` class.

3 Discrete Mathematics Lesson

In our Discrete Mathematics class, students are introduced to the notion of proof using strong induction. One of the classic examples in discrete mathematics is to show the completeness of the binary numbering system. Although we present this proof, students find it trivial and unnecessary. Egyptian fractions, on the other hand, are not as intuitive to most first-year CS students. Some of them are not at all convinced that every fraction (both proper and improper) can be expressed as the sum of distinct unit fractions.

A proof of the completeness of Egyptian fractions (i.e., that every rational number can be represented in Egyptian form) can be given using strong induction. It provides an excellent example of the need for proof, especially if it is followed by an attempt to prove that "powers of two" Egyptian fractions (i.e., Egyptian fractions whose unit fractions all have denominators that are powers of two) are complete. Showing that "powers of two" Egyptian fractions are simply an alternative way of representing repeating binaries (i.e., the binary equivalent of repeating decimal numbers) provides further insight into the representation of number. A proof that all proper fractions can be represented in Egyptian form is presented below. The proof that all fractions (both proper and improper) can be represented in Egyptian form is also given in class.

Completeness of Egyptian Fractions

Every fraction $\frac{p}{q}$ such that $0 < \frac{p}{q} < 1$ can be represented in Egyptian form $\frac{p}{q} = \frac{1}{n_1} + \frac{1}{n_2} + \cdots + \frac{1}{n_m}$ where $n_1, n_2, \ldots, n_m$ are positive integers satisfying $n_1 < n_2 < \cdots < n_m$.

Proof (by induction on p):

Basis ($p = 1$):

If p is 1 then $\frac{p}{q}$ is already in Egyptian form.

Inductive Hypothesis: Every fraction $\frac{p}{q}$ such that $0 < \frac{p}{q} < 1$ and $1 \le p < k$ can be represented in Egyptian form $\frac{p}{q} = \frac{1}{n_1} + \frac{1}{n_2} + \cdots + \frac{1}{n_m}$ where $n_1, n_2, \ldots, n_m$ are positive integers satisfying $n_1 < n_2 < \cdots < n_m$.

Inductive Step: We must show that every fraction $\frac{k}{q}$ such that $0 < \frac{k}{q} < 1$ can be represented in Egyptian form: $\frac{k}{q} = \frac{1}{n_1} + \frac{1}{n_2} + \cdots + \frac{1}{n_m}$ where $n_1, n_2, \ldots, n_m$ are positive integers satisfying $n_1 < n_2 < \cdots < n_m$. Choose the smallest positive integer n such that $\frac{1}{n} \le \frac{k}{q}$. Clearly, $n > 1$ since we know that $\frac{k}{q} < 1$. Furthermore, since n is the smallest positive integer satisfying $\frac{1}{n} \le \frac{k}{q}$ and $n-1$ is a positive integer less than n, it follows that $\frac{k}{q} < \frac{1}{n-1}$. There are two cases to consider: $\frac{1}{n} = \frac{k}{q}$ and $\frac{1}{n} < \frac{k}{q}$.

Case 1 ($\frac{1}{n} = \frac{k}{q}$): The Egyptian form for $\frac{k}{q}$ is $\frac{1}{n}$.

Case 2 ($\frac{1}{n} < \frac{k}{q}$): Consider the value $\frac{k}{q} - \frac{1}{n}$. Clearly $\frac{k}{q} - \frac{1}{n} > 0$ since $\frac{1}{n} < \frac{k}{q}$. Furthermore, since, $\frac{k}{q} < \frac{1}{n-1}$ it follows that $k(n-1) < q$ so $nk - q < k$. Note that we can rewrite $\frac{k}{q} - \frac{1}{n}$ as $\frac{nk-q}{nq}$. Since $nk - q < k$, we know by the inductive hypothesis that $\frac{nk-q}{nq}$ can be written in Egyptian form $\frac{nk-q}{nq} = \frac{1}{n_1} + \frac{1}{n_2} + \cdots + \frac{1}{n_m}$ where $n_1, n_2, \ldots, n_m$ are positive integers satisfying $n_1 < n_2 < \cdots < n_m$. So $\frac{k}{q}$ can be written in Egyptian form $\frac{k}{q} = \frac{1}{n} + \frac{1}{n_1} + \frac{1}{n_2} + \cdots + \frac{1}{n_m}$ where $n_1, n_2, \ldots, n_m$ are positive integers satisfying $n_1 < n_2 < \cdots < n_m$.

Hence, regardless of the choice of n, $\frac{k}{q}$ can be represented in Egyptian form. Since we have proved the basis step and the inductive step of the strong mathematical induction, every fraction $\frac{p}{q}$ such that $0 < \frac{p}{q} < 1$ can be represented in Egyptian form $\frac{p}{q} = \frac{1}{n_1} + \frac{1}{n_2} + \cdots + \frac{1}{n_m}$ where $n_1, n_2, \ldots, n_m$ are positive integers satisfying $n_1 < n_2 < \cdots < n_m$. □

Another important concept for first-year CS students to grasp is the notion of program correctness. The Egyptian multiplication algorithm can be used as an example of the use of strong induction to prove the correctness of a Java program. In addition to proving the correctness of various Java programs (including one that computes n^2 recursively using the formula $(n-1)^2+2n-1$), we prove the correctness of the following Java implementation of a `times` method that finds the product of two positive integers using Egyptian multiplication.

Egyptian Multiplication Method

```
int times(int m, int n) {
  if (m == 0)
    return 0;
  else if (m == 1)
    return n;
  else if ((m%2) == 0)
    return times(m/2,n+n);
  else
    return times(m/2,n+n)+n;
}
```

Correctness of Egyptian Multiplication

The function $\mathsf{times}(m,n)$ returns the value $m \cdot n$ where $m, n \geq 0$.

Proof (by induction on m):

Basis ($m = 0$ and $m = 1$):

If $m = 0$ then the function $\mathsf{times}(0,n)$ returns $0 = 0 \cdot n$ as defined by the *then* part of the first conditional.

If $m = 1$ then the function $\mathsf{times}(1,n)$ returns $n = 1 \cdot n$ as defined by the *then* part of the second conditional.

Inductive Hypothesis: For all i with $1 \leq i < k$, the function $\mathsf{times}(i,n)$ returns $i \cdot n$.

Inductive Step: We must show that the function $\mathsf{times}(k,n)$ returns $k \cdot n$. There are two cases to consider: k is even and k is odd.

Case 1 (k is even): By the *then* part of the third (or innermost) conditional, it follows that:

$$\begin{aligned}\mathsf{times}(k,n) &= \mathsf{times}(k \text{ div } 2, n+n)\\ &= \mathsf{times}(\left\lfloor \frac{k}{2} \right\rfloor, 2n)\\ &= \mathsf{times}(\frac{k}{2}, 2n)\\ &= (\frac{k}{2}) \cdot (2n)\\ &= k \cdot n\end{aligned}$$

Case 2 (k is odd): By the *else* part of the third (or innermost) conditional, it follows that:

$$\begin{aligned}\mathsf{times}(k,n) &= \mathsf{times}(k \text{ div } 2, n+n)+n\\ &= \mathsf{times}(\left\lfloor \frac{k}{2} \right\rfloor, 2n)+n\\ &= \mathsf{times}(\frac{k-1}{2}, 2n)+n\\ &= \left((\frac{k-1}{2}) \cdot (2n)\right)+n\\ &= \big((k-1) \cdot n\big)+n\\ &= k \cdot n\end{aligned}$$

Hence, regardless of whether k is even or odd, the program is correct. Since we have proved the basis step and the inductive step of the strong mathematical induction, the function $\mathsf{times}(m,n)$ returns the value $m \cdot n$ where $m, n \geq 0$. □

4 Conclusion

Although our students were taking discrete mathematics during their first year, they were apparently not retaining it. In order to address this problem, we restructured our CS curriculum last year. Now our students simultaneously take two semesters of discrete mathematics, a Discrete Mathematics course that is taught by the Mathematics Department followed by a Discrete Structures course taught by the Computer Science Department, while they are taking CS1 and CS2.

The additional semester of discrete mathematics has helped, however, we believe that the current success of our CS students in CS1 and CS2 has significantly benefited from a very active attempt to make the discrete mathematics more relevant. We have attempted to show our students this relevance by using a CS-complete example, a unifying example that is applicable in CS1, CS2, and Discrete Mathematics.

References

[1] Burton, D.M. *The History of Mathematics: An Introduction*. Boston: Allyn and Bacon, Inc., 1985.

[2] Computing Curricula 2001. Chapter 6 – Defining a Curriculum. Online. Internet. [March 6, 2000 draft]. Available WWW: http://www.computer.org/education/cc2001/report/curriculum.html