A Gentle Introduction to Mutual Recursion

Manuel Rubio-Sánchez
Departamento de Lenguajes y Sistemas Informáticos 1
Universidad Rey Juan Carlos
c/ Tulipán s/n, 28933 Móstoles
Madrid, Spain
manuel.rubio@urjc.es

Jaime Urquiza-Fuentes
Departamento de Lenguajes y Sistemas Informáticos 1
Universidad Rey Juan Carlos
c/ Tulipán s/n, 28933 Móstoles
Madrid, Spain
jaime.urquiza@urjc.es

Cristóbal Pareja-Flores
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Avda. Puerta de Hierro s/n
28040 Madrid, Spain
cpareja@sip.ucm.es

ABSTRACT

Recursion is an important topic in computer science curricula. It is related to the acquisition of competences regarding problem decomposition, functional abstraction and the concept of induction. In comparison with direct recursion, mutual recursion is considered to be more complex. Consequently, it is generally addressed superficially in CS1/2 programming courses and textbooks. We show that, when a problem is approached appropriately, not only can mutual recursion be a powerful tool, but it can also be easy to understand and fun. This paper provides several intuitive and attractive algorithms that rely on mutual recursion, and which have been designed to help strengthen students’ ability to decompose problems and apply induction. Furthermore, we show that a solution based on mutual recursion may be easier to design, prove and comprehend than other solutions based on direct recursion. We have evaluated the use of these algorithms while teaching recursion concepts. Results suggest that mutual recursion, in comparison with other types of recursion, is not as hard as it seems when: (1) determining the result of a (mathematical) function call, and, most importantly, (2) designing algorithms for solving simple problems.

Categories and Subject Descriptors
D.1.1 [Applicative (Functional) Programming]: Recursion; G.2.1 [Combinatorics]: Counting problems; K.3.2 [Computer and Information Science Education]: Computer science education

General Terms
Algorithms

Keywords
Mutual recursion, recursion problems, combinatorics, counting problems, Fibonacci numbers.

1. TEACHING MUTUAL RECURSION

Recursion is a fundamental programming concept, and is therefore necessary in a computer science curriculum. It plays an important role in the acquisition of competences regarding functional abstraction and problem decomposition through the concept of induction. Nevertheless, computer science students generally find recursion hard to master. Numerous authors have tried to identify the factors (conceptual models, cognitive learning styles, functional abstraction reasoning) responsible for these difficulties [14, 12]. Others have proposed methods designed to overcome these problems by using visualizations of the recursion or activation tree [4, 9], or through three-dimensional animations [2]. Additional studies propose teaching methodologies, such as presenting concepts in gradual steps (firstly grammars, then functional languages, and finally recursion in imperative languages) [15].

Recursive algorithms can be categorized according to the number and type of the recursive function calls. A common classification distinguishes the following types: linear (of which tail recursion is a special case), multiple (or exponential), nested, and mutual. The last two types are considered to be especially complex, and, consequently, are generally not treated in depth in CS1/2 programming courses or textbooks. Moreover, they are usually explained by using a single basic example. Nested recursion is taught with the Ackermann function, whereas mutual recursion is seen through the following pair of functions for determining the parity of a nonnegative integer:

\[
\begin{align*}
\text{is}\_\text{odd}(n) &= \begin{cases} 
\text{FALSE} & \text{if } n = 0 \\
\text{is}\_\text{even}(n-1) & \text{if } n > 0 
\end{cases} \\
\text{is}\_\text{even}(n) &= \begin{cases} 
\text{TRUE} & \text{if } n = 0 \\
\text{is}\_\text{odd}(n-1) & \text{if } n > 0 
\end{cases}
\end{align*}
\]

While nested recursion is rather infrequent, mutual recursion appears often in advanced computer science topics (e.g., parsers). This paper provides several simple, intuitive and attractive algorithms that rely on mutual recursion, the majority of which solve combinatorial problems involving Fibonacci numbers \(F_n = F_{n-1} + F_{n-2}, F_1 = 1, F_2 = 1\). The goal is to present algorithms that can help students focus on problem decomposition, induction and functional abstraction. Additionally, several of these algorithms provide solutions that may be easier to design, prove, and comprehend than other strategies based on direct recursion. The problems have been chosen carefully so that they can be implemented easily with imperative programming languages,
avoiding issues related to mechanisms such as parameter passing, control stack, etc. (see [13]). A pedagogical evaluation suggests that mutual recursion, in comparison with direct recursion, is not as hard as it seems when: (1) determining the result of a (mathematical) function call, and, most importantly, (2) designing algorithms for solving simple problems.

The rest of the paper is organized as follows. Section 2 presents and analyzes the proposed combinatorial problems and their solutions. Section 3 shows the experimental results, while a discussion and conclusions are reported in Sec. 4.

2. COMBINATORIAL PROBLEMS

Recursion is often explained in CS1/2 courses with the aid of mathematical functions, such as the factorial, power, or binomial coefficient. An interesting characteristic shared by these functions is their relation with combinatorics (permutations, variations with repetition, combinations). However, most texts only focus on their recursive definitions, instead of providing (combinatorial) problems that can be decomposed recursively, and whose solutions exhibit the same structure inherent in the functions. For instance, it is easy to see that the number of different permutations \( f_n \), of \( n \) different elements, can be decomposed as \( f_n = n \cdot f_{n-1} \), with \( f_0 = f_1 = 1 \). Due to the abundance of basic combinatoric examples, there is a wide range of valuable combinatoric problems for teaching recursion [11].

Another rich class of combinatorial counting problems involves Fibonacci numbers. A collection of such problems can be found in [7]. Simpler ones can be decomposed easily into \( f_n = f_{n-1} + f_{n-2} \), like the number of ways to climb a flight of \( n \) steps leaping one or two steps at a time (“Leonardo’s leaps” problem). More challenging problems need other Fibonacci identities (e.g., involving the computation of the even or odd terms of the Fibonacci sequence, where \( f_n = 3f_{n-1} - f_{n-2} \)). This section presents and analyzes solutions based on mutual recursion for several of these problems. The goal is to focus on problem decomposition and the concept of induction, rather than strictly concentrating on mathematical functions.

2.1 Fibonacci Rabbits Problem

The origin of the Fibonacci numbers and their sequence can be found in a thought experiment posed in 1202. It was related to the population growth of pairs of rabbits under ideal conditions. The objective was to calculate the size of a population of rabbits during each month according to the following rules:

- Originally, a newly-born pair of rabbits, one male and one female, are put in a field.
- Rabbits take one month to mature and never die.
- It takes one month for mature rabbits to produce another pair of newly-born rabbits, who will thereafter mate together.
- The female always produces one male and one female pair every month from the second month on.

2.1.1 Distinguishing Baby and Adult Pairs

This problem can be approached by considering baby and adult pairs separately [10]. Let \( B_i \), \( A_i \), and \( T_i \) be the number of baby, adult and total number of pairs of rabbits, re- spectively, at month \( i \). Since every adult pair produces a new pair of babies each month, the number of baby pairs at month \( i \) is equal to the number of adult pairs in the previous month \( i - 1 \):

\[
B_i = \begin{cases} 
1 & \text{if } i = 1 \\
A_{i-1} & \text{if } i \neq 1
\end{cases}
\]

Furthermore, the number of adult pairs at month \( i \) is the sum of the adult pairs in the previous month \( i - 1 \), plus the number of baby pairs that will have matured over the previous month:

\[
A_i = \begin{cases} 
0 & \text{if } i = 1 \\
A_{i-1} + B_{i-1} & \text{if } i \neq 1
\end{cases}
\]

The following identities can be easily proven:

\[
T_i = A_i + B_i = A_{i+1} + B_{i+2} = F_i
\]

Thus, \( A_i \), \( B_i \), and \( T_i \) are Fibonacci numbers.

The presented algorithm is a particular case of the following general mutually recursive functions:

\[
B_i = \begin{cases} 
\beta_0 & \text{if } i = 1 \\
\beta_i + A_{i-1} & \text{if } i \geq 2
\end{cases}
\]

\[
A_i = \begin{cases} 
\alpha_0 & \text{if } i = 1 \\
\alpha_i + A_{i-1} + B_{i-1} & \text{if } i \geq 2
\end{cases}
\]

for \( \beta_0 = 1 \), \( \beta_i = \alpha_i = \alpha_0 = 0 \). These functions are also related to Fibonacci numbers:

\[
A_n + B_n + \alpha_i + \beta_i = A_{n+1} + \beta_i = B_{n+2} = \beta_0 F_n + (\beta_i + \alpha_i)F_{n+1} + \alpha_i F_{n+2} - \alpha_i
\]

Since each constant is associated to a specific type of node on the recursion tree, it is easy to see that the cardinality of many subsets of its nodes is also a Fibonacci number. This analysis may be useful when addressing, for example, computational complexity (see [8] for a study on this topic with Fibonacci numbers).

2.1.2 Genealogical Tree

The problem can also be tackled using the Fibonacci identity \( F_n = 1 + \sum_{i=1}^{n-2} F_i \). This decomposition is possible since the structure of the recursion tree for the recurrence relation is equivalent to the genealogical tree formed by the rabbit population growth process (see Fig. 1, where the numbers represent the months in which the rabbits are born). Replacing the summation with another function gives a new algorithm based on mutual recursion:

\[
F_i = \begin{cases} 
1 & \text{if } i = 1, 2 \\
1 + H_{i-2} & \text{if } i \geq 3
\end{cases}
\]

\[
H_i = \begin{cases} 
0 & \text{if } i = 0 \\
F_i + H_{i-1} & \text{if } i \geq 1
\end{cases}
\]

Nevertheless, it must be noted that mutual recursion does not appear naturally in this example as the result of performing problem decomposition.
2.2 Water Treatment Plants Puzzle

In the “Water treatment plants puzzle” [3, 7] there are \( n \) cities, each with a water treatment plant, located along a side of a river (see Fig. 2). Each city generates sewage water that must be cleaned in a plant (not necessarily its own) and discharged into the river through pipes, where additional pipes connect neighboring cities. If a plant is working it will clean the sewage water from its city, plus any water coming from a pipe connecting a neighboring city, and discharge it to the river. However, a plant may not be working, in which case the water from its city, plus any water coming from a neighboring city, must be sent to another city. Given that water can flow in only one direction inside a pipe, the problem consists of determining the number of different ways it is possible to discharge the cleaned water into the river for \( n \) cities. It can be solved by modeling the water flow between cities, or the water discharge at each city, which has a more intuitive solution based on mutual recursion. In what follows, we will assume that the cities are arranged from left to right.

2.2.1 Water Flow Between Cities

For the \( n - 1 \) pipes connecting the cities there are three possibilities denoted by the characters: (‘N’) if there is no flow; (‘R’) if water flows to the right; and (‘L’) if water flows to the left. In this situation the valid configurations involve strings of length \( n - 1 \) where the substring “LR” does not appear, since it would mean that a city sends its water to more than one city.

Let \( A_n \) be the solution to the problem. It can be defined recursively as [3]:

\[
A_{n+2} = 3A_{n+1} - A_n
\]

which is a Fibonacci identity for even or odd sequence indices. For this problem, given the initial conditions \( A_1 = 1 \) and \( A_2 = 3 \), \( A_n = F_{2n} \). Despite the fact that the formula uses direct recursion, our classroom experience indicates that deriving it is rather tricky. Given the value of \( A_{n+1} \), adding a new city increments the number of solutions to \( 3A_{n+1} \) (appending an ‘N’, ‘R’ or ‘L’). However, the new strings that end in “LR” must not be counted, and the number of such solutions is exactly \( A_n \) (see Fig. 2a). Most students do not understand this subtraction easily.

2.2.2 Water Discharge at Each City

Another solution based on mutual recursion can be developed by modeling the direction in which the water is discharged at each city. Again, there are three possibilities: (‘V’) directly down to the river; (‘>’) to the city to the right; and (‘<’) to the city to the left. In this case, valid configurations are represented by strings of \( n \) characters, where the substring “> <” is not permitted, nor a beginning with ‘<’, nor an ending with ‘>’. According to these constraints, the problem can be decomposed into smaller similar sub-problems. By fixing, for instance, the way in which the leftmost city discharges its water, the following three functions can be formulated (see Fig. 2b):

\[
f_v(n) = \begin{cases} 2 & \text{if } n = 1 \\ f_v(n-1) + f_s(n-1) + f_< (n-1) & \text{if } n > 1 \end{cases}
\]

\[
f_s(n) = \begin{cases} 1 & \text{if } n = 1 \\ f_v(n-1) + f_s(n-1) & \text{if } n > 1 \end{cases}
\]

\[
f_< (n) = \begin{cases} 2 & \text{if } n = 1 \\ f_v(n-1) + f_s(n-1) + f_< (n-1) & \text{if } n > 1 \end{cases}
\]

where, \( f_v(n) \), \( f_s(n) \), and \( f_< (n) \) denote the total number of valid configurations for \( n + 1 \) cities, assuming that the leftmost city discharges its water down to the river, to the right, or to the left, respectively (note that \( f_v(n) = f_s(n) \)). The solution is given by:

\[
A_n = f_v(n-1) + f_s(n-1) = f_s(n)
\]

Finally, it is easy to prove that \( f_v(n) = F_{2n+1} \), and \( f_s(n) = F_{2n} \). Hence, the solution is a Fibonacci number.

2.3 ‘Steven and Todd’ Problem

The ‘Steven and Todd’ problem [6, 7] is an interesting example of how people with different educational backgrounds can provide completely different solutions or proofs (in this case mathematicians and computer scientists).

Consider the sequence \( \alpha = \{a_1, a_2, \ldots, a_k\} \) of a subset of the first \( n \) positive integers that have been arranged in increasing order \( 1 \leq a_1 < a_2 < \cdots < a_k \leq n \). The problem consists of finding the total number \( t_\alpha \) of sequences \( \alpha \) that start with an odd number and thereafter alternate in parity, that is,

\[
\alpha = \text{odd, even, odd, even, ...}?
\]

The solution to the problem is:

\[
t_\alpha = t_{\alpha-1} + t_{\alpha-2} = F_{n+2}
\]

where \( t_1 = 2 \) for \( \alpha = \emptyset, \{1\} \), and \( t_2 = 3 \) for \( \alpha = \emptyset, \{1, 1, 2\} \).

A formal proof for this problem can be found in [6]. This section describes an alternative - and perhaps simpler - approach to the problem, which consists of using an algorithm to construct each of the possible configurations, and then prove that the solution is \( F_{n+2} \). For simplicity of implementation, however, an isomorphic problem will be addressed: calculating the number of sequences sequence \( \delta = \{d_1, d_2, \ldots, d_k\} \) of a subset of the first \( n \) positive integers that have been arranged in decreasing order \( n \geq d_1 > d_2 > \cdots > d_k \geq 1 \). In this case, the sequences start with an integer of the same parity as \( n \), and thereafter alternate in parity. Note the equivalence between the \( \delta \) and \( \alpha \) sequences \( (d_i = n + 1 - a_i) \).
### Table 1: Percentages of correct evaluations for several function calls.

<table>
<thead>
<tr>
<th>function</th>
<th>linear</th>
<th>tail</th>
<th>multiple</th>
<th>mutual</th>
</tr>
</thead>
<tbody>
<tr>
<td>A&lt;sub&gt;3418&lt;/sub&gt;</td>
<td>83%</td>
<td>76%</td>
<td>91%</td>
<td>97%</td>
</tr>
<tr>
<td>B&lt;sub&gt;8,6&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C&lt;sub&gt;2,3&lt;/sub&gt;</td>
<td>91%</td>
<td>83%</td>
<td></td>
<td>100%</td>
</tr>
<tr>
<td>D&lt;sub&gt;7&lt;/sub&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>% of correct evaluations</td>
<td>76%</td>
<td>91%</td>
<td>83%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 2: Contingency table showing the relationship between the type of recursion used and the number of correct implementations in problem (4).

<table>
<thead>
<tr>
<th>problem (4)</th>
<th>mutual</th>
<th>linear</th>
</tr>
</thead>
<tbody>
<tr>
<td># correct implementations</td>
<td>26</td>
<td>9</td>
</tr>
<tr>
<td># incorrect implementations</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

where % and / represent the remainder and quotient of integer divisions, respectively. Table 1 shows the percentages of correct evaluations in the post-test for several function calls. The difficulty appears to lie in the number and complexity of the mathematical operations to be carried out, rather than in the sequence of operations that will lead to a particular solution, which is claimed to account for much of the complexity associated with mutual recursion. Note that the percentages are higher for tail recursive functions. Additionally, every student obtained a correct answer for the mutual recursion case.

A major goal was to evaluate the students’ abilities to design recursive algorithms. Students were asked to implement four algorithms: (1) the quotient of an integer division using linear (non-tail) recursion, (2) the same quotient using tail recursion, (3) a combinatorial coefficient, and (4) the solution to the following problem:

Two employees, A and B, work in a warehouse which must be emptied of parcels. They take turns to remove the parcels using a different strategy: employee A always removes one parcel, while employee B removes two parcels if the number of parcels left is even, and one if it is odd. If employee A always removes the first parcel, calculate the number of turns needed to empty a warehouse containing n parcels.

The percentages of correct implementations in the post-test for the four problems were 50% (1), 36% (2), 76% (3), and 55% (4). Additionally, the fourth problem can be implemented through mutual or linear recursion. Table 2 shows the relationship between the type of recursion used and the number of correct implementations for problem (4), where five students did not provide any solution. These results suggest that students do not have more difficulty in understanding and applying mutual recursion than direct recursion. Rather, the complexity arises from the formulation of the problem. Furthermore, not only did they prefer mutual recursion over linear on the fourth problem (35 vs. 18), but also 74% of the implementations based on mutual recursion were correct, while only 33% of the solutions based on linear recursion had a proper implementation.

Comparing the type of recursion used for problem (4) in the pre-test vs. the post-test, it is worth mentioning that 10 students switched from implementing a solution based on linear recursion in the pre-test to coding a mutually recursive algorithm in the post-test. None of these 10 students had a correct solution in the pre-test, but seven designed an appropriate implementation in the post-test. Furthermore, students who did not provide any solution in the pre-test did not prefer any particular type of recursion (six mutual and six linear solutions). However, out of these 12 students, three implemented a correct solution in the post-test by using mutual recursion, while only one arrived at a proper implementation based on linear recursion.
4. DISCUSSION AND CONCLUSIONS

This paper provided a collection of problems with mutually recursive solutions that are derived through induction, and can be understood by CS2 students without any previous exposure to functional programming. Emphasis is on problem decomposition, where it is not necessary, or even desirable, to analyze recursion trees (this may be more suitable when explaining more advanced concepts, such as backtracking, memoization, computational complexity, etc.).

The experiments indicate that students did not find difficulties in evaluating simple mutually recursive mathematical functions. Therefore, determining the sequence of operations that will lead to a particular solution is not necessarily more difficult for mutual recursion. Furthermore, the ability to analyze the structure of the recursion tree, and evaluate the result of the various recursive calls, is not necessarily related to the ability to decompose problems in a recursive manner. Studies show that the key to understanding functional abstraction is “not to think too hard” [12], in the sense that students must focus on what the algorithm does, rather than on how it does it.

The article reports several results related to learning mutual recursion (the authors are not aware of any other study involving an experimental evaluation focused on this topic). Our conclusion is that mutual recursion is not as hard as it seems. The chances of obtaining a correct solution increased when using mutual recursion in problem (4), see Sec. 3. This reflects the fact that a problem may have a more intuitive mutually recursive solution, depending on how it is expressed. Therefore, it is important that students be exposed to mutual recursion at some point in their curricula. A natural place would be a functional programming course. However, results indicate that students assimilate mutual recursion well in a CS2 imperative programming course.

There exist other problems (besides combinatorial) with mutual recursion. One example is the ‘Cyclic Towers of Hanoi’ [1], an elegant and illustrative variant of the Towers of Hanoi problem, where the disks must move in only one direction.

Figure 3 shows the solution to move n disks from tower X to Y using Z in clockwise direction, and from X to Z using Y in anticlockwise direction. It constitutes an ideal companion to the original problem since the fundamental reasoning about the movement of the disks, as well as the decomposition of the problem into sub-problems, remain basically the same.

We believe that the proposed algorithms provide an alternative and enlightening point of view of recursion, which can enhance students’ comprehension and assimilation of problem decomposition, functional abstraction and induction.

Finally, the problems discussed in this paper, along with other algorithmic or combinatorial problems, can be easily used in a problem-based learning framework.

5. ACKNOWLEDGEMENTS

Research partially supported by the Spanish MEC projects: WEST/FAST TIN2006-15578-C02-01, and TIN2004-07568. We would like to thank Dr. J. A. Velázquez-Iturbide, as well as the anonymous reviewers, for their helpful comments.

6. REFERENCES