Letters & Numbers: A Vehicle to Illustrate Mathematical & Computing Fundamentals

S. J. SUGDEN¹, P. A. STOCKS²

¹ School of Mathematical Sciences, Queensland University of Technology, Brisbane, Australia
* Corresponding author: stephen.sugden@qut.edu.au
² Division of Informatics, Bond University, Gold Coast, Australia, Email: pstocks@bond.edu.au

Keywords: Letters & Numbers, mathematics, computer science.

The television quiz program *Letters and Numbers*, broadcast on the SBS network, has recently become quite popular in Australia. This paper explores the potential of this game to illustrate and engage student interest in a range of fundamental concepts of computer science and mathematics. The Numbers Game in particular has a rich mathematical structure whose analysis and solution involves concepts of counting and problem size, discrete (tree) structures, language theory, recurrences, computational complexity, and even advanced memory management. This paper presents an analysis of these games and their teaching applications, and presents some initial results of use in student assignments.

1. Introduction

The television quiz program *Letters and Numbers*, broadcast on the SBS network, has only recently become quite popular in Australia. In various European countries the same essential format (with relatively minor variations) has been popular in Europe for decades; in the UK it is known as *Countdown*, while in France it is *Des chiffres et des lettres*.

This paper explores the game’s potential as a vehicle to illustrate and engage student interest in a range of fundamental concepts of computer science and mathematics. The Numbers Game in particular has a very rich mathematical structure whose analysis and solution involves concepts of counting and problem size, discrete (tree) structures, expression trees, expression tree evaluation, recurrences, formal language theory, parsing and generation, computational complexity, and even advanced memory management.

The fact that so many significant computer science concepts can be presented in the context of this game makes it stand out as an example problem to use for discussion and analysis. It can even serve as a running example to demonstrate different topics arising in the same class. Finally, since it is a game, popular on television, it has engagement appeal as a class example. Students tend to like games, even maths games, as part of learning new theory [1]. An analysis of the *Letters and Numbers* game and its teaching applications in Discrete Mathematics, Mathematics of Computing, and Computer Science is presented. Also discussed is use of the game as a basis for assignment work in a Discrete Mathematics class at Bond University.

This paper is organised as follows. Section 2 discusses the Letters Game. Section 3 discusses the Numbers Game, the underlying language of it, its size, and the computational cost of simple solution procedures. Section 4 describes three programming implementations of Numbers Game solvers. Section 5 discusses the teaching applications of the Numbers Game, and Section 6 discusses the experiences of using the Numbers Game as an assignment in an introductory Discrete Mathematics class. Finally, Section 7 offers our conclusions.
2. Letters game

Nine letters of the alphabet are chosen, with one contestant choosing a sequence of *vowels* or *consonants*. Duplicates are possible. Then both contestants are given 30 seconds in which to form the longest possible English word from a subset of the chosen letters. Such words must either exist in the standard dictionary used on the program or be well-defined extensions of dictionary words. To solve the problem on a computer is straightforward. Given electronic dictionary files (just words as simple text), it is a relatively trivial matter to write code to scan the dictionary for acceptable words. An obvious approach is to first compute the frequency of each letter in the original collection of nine. This information may be stored conveniently as an array, $A$, of 26 non-negative integers. As the dictionary scan proceeds, a similar array $B$ is computed for each word encountered. If, for any dictionary word, each element of $B$ does not exceed the corresponding element of $A$, then we have a “hit” and that word is output as a solution.

There is nothing very profound here, but it can be a useful programming exercise for elementary programming subjects. The first author has used essentially this problem as a programming exercise for undergraduates over several years in the 1990s at Bond University, and details may be found in [2]. Algorithmically, this approach is linear in the size of the dictionary. While potentially more efficient approaches using only permutations of subsets of the original nine letters and a dictionary encoded in an efficient lookup structure like a retrieval tree [3] can be imagined, they can only offer a constant factor speed-up. The linear cost on the size of the dictionary is not prohibitive enough to warrant the extra effort and complexity given the speed of modern processing.

The letters part of the game is not discussed any further, except in the discussion of its use in a student assignment at Bond University in Section 6. It is the numbers part of the game which has a much richer mathematical structure, and that forms the subject of the remainder of the paper.

3. Numbers Game

In this part of the game, one contestant is invited to choose a “combination” of six numbers. In normal mathematical parlance, the term “combination” refers to a *subset*; however, in this game, it does not refer to a set at all, but rather to two set cardinalities. Even this is not quite true as the collection of numbers ultimately chosen may contain duplicates. For example, in response to the invitation to choose a combination, a contestant may typically say “two large and four small numbers”. These are references to two underlying sets of possible small and large numbers, respectively $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and $L = \{25, 50, 75, 100\}$.

In fact it is not the contestant but “Numbers Queen”, Lily Serna, who chooses the numbers by selecting the specified quantities of large and small numbers from two separate collections of face-down cards. Duplicates are possible, so selection may be regarded as being with replacement. For the choice “two large and four small numbers”, an example of the final “set” of chosen numbers might be $\{2, 3, 2, 7, 50, 75\}$. Strictly, this is not a set but a *multi-set* (some authors use the term *bag*), since duplicates may be present.

Extreme choices such as all large or all small are permitted but rarely chosen, as there seems to be a perception that such extremes may make the next part (generating the target number) more difficult. Frequent “combination” choices are three each of large and small, the so-called “family mix” of two large and four small, and less commonly, the “rat-pack” of six small numbers.

Once the six numbers (hereafter called *operands*), with possible duplicates, are selected, Lily then presses a button, which presumably directs a computer to generate a pseudo-random three-decimal-digit number (hereafter called the *target*) from 101 to 999.
inclusive. Contestants then are given 30 seconds to find a way of combining any or all of the six operands, with the usual arithmetic operations of addition, subtraction, multiplication and division, and parentheses where necessary, with the aim of generating the target. There is no special benefit to players in using more or fewer operands; rather the aim is to generate the target exactly, or within 10 either side. It is not explicitly stated, but nevertheless clear that no intermediate result may be non-integer, or negative.

3.1. Problem size and Catalan numbers

Suppose we wish to generate arithmetic expressions with \( n \) operands. The corresponding binary expression tree has \( n \) operand nodes (leaf nodes) plus \( n-1 \) operators (internal nodes), giving \( 2n-1 \) nodes in total. Consider now the interior tree generated by the \( n-1 \) interior operator nodes. The number of distinct such trees is given by the Catalan number of index \( n-1 \), which we denote \( C_{n-1} \).

\[
C_{n-1} = \frac{(2n-2)!}{(n-1)!n!}
\]

(1)

There are four possible operators and these may be chosen \( n-1 \) times with replacement. Thus, the number of choices here is \( 4^{n-1} \). Now graft \( n \) operand nodes onto this tree. Since order matters and operands can only be used once, the number of ways of doing this is the number of permutations of \( n \) objects chosen from 6. This is

\[
\frac{6!}{(6-n)!}
\]

(2)

The multiplication rule now gives us the total number of trees with \( n \) operands:

\[
T_n = \frac{(2n-2)!}{(n-1)!n!} \times 4^{n-1} \times \frac{6!}{(6-n)!}
\]

(3)

This expression ignores the possibilities of semantically equivalent trees. It also ignores unsuitable, semantically invalid trees such as those which generate division by zero or improper final or intermediate results. Examples are \( 5-7 \) or \( 25/4 \) or \( 10/(3-3) \). In our various implementations, the code checks for these and avoids constructions of such (sub)trees.

The expression on the right side of Equation (3) cannot be simplified in any significant way, however if we consider the quotient \( T_n / T_{n-1} \) massive cancellation occurs and, with \( T_1 = 6 \), the useful recursive form of Equation (4) results.

\[
T_n = \frac{8(2n-3)(7-n)}{n} T_{n-1} \quad \text{if} \quad 2 \leq n \leq 6
\]

(4)

The number of trees as given by either Equation (3) or Equation (4) is shown in Error! Reference source not found.. Table 1: Number of expression trees with \( n \) operand nodes

<table>
<thead>
<tr>
<th>( n )</th>
<th>( T_n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>120</td>
</tr>
<tr>
<td>3</td>
<td>3,840</td>
</tr>
<tr>
<td>4</td>
<td>115,200</td>
</tr>
<tr>
<td>5</td>
<td>2,580,480</td>
</tr>
</tbody>
</table>
3.2. Efficiency of solution generation

A brute force approach to solving the Numbers Game is simply to generate every possible expression, or equivalently, every tree described in Section 3.1, and then evaluate them. Clearly, this is computationally expensive, with cost proportional to the number of possible expression trees. Furthermore, this cost is not just in run time, but also in memory to store the viable structures. A naive Haskell implementation of this search (which uses copying to create new trees, and garbage collection instead of explicit deallocation) running on an Intel i5 processor with 4 Gb of RAM runs for several hours without showing any signs of completion. Clearly, such a naive approach is impractical.

However, there is much scope for optimisation of the solution generation process. First, division by zero is not defined, which introduces the notion of an invalid or illegal expression. If such an expression appears anywhere in the final expression tree, the whole expression is invalid. Hence, when an illegal intermediate expression is discovered, the construction of the remaining possible super expression trees can be halted as none will be valid. Also, while not invalid as expressions, intermediate values that are negative or proper fractions are never used in game play, which leads to the notion of an implicit rule that such expressions are illegal as discussed in the beginning of this section. In every case of our empirical solution generation, expression trees containing instances of these three kinds of invalid expressions account for well more than half of all the possible expressions given the operands, but for the most part represent an order of magnitude more possible trees. Not generating these trees yields significant savings in time and space cost and brings solution generation back into tractable realms of time and space cost, though the space usage is still very high: on the architecture described above, it takes approximately 40 seconds to generate all viable solutions and allocates over 79 Gb of heap space with maximum heap usage at any one time of nearly 0.5Gb.

Beyond invalid expressions, different intermediate expression trees can be constructed that evaluate to the same value. For example, given operands 1, 2, 3, 4, 5, 6, we need only generate one of $2 \times 3$ and $3 \times 2$. Trees representing these two simple expressions are not the same, but the trees both belong to the same equivalence class, with an appropriate definition of equivalence or equivalence relation. It is not difficult to exclude “duplicates” in the sense of this relation, however there are some more subtle forms of equivalence. Commutative equivalence can be checked syntactically, but general value equivalence requires the expression (sub) trees to be evaluated.

Even if equivalently evaluating expressions are not discarded in construction, significant savings in memory can be achieved by using memorisation [4]. Here, each unique (sub) expression tree is constructed only once and stored. When a sub-expression tree would be duplicated in another tree, a pointer into the expression tree storage is used instead. Clearly, this makes the solution generation much more complex, but results in massive memory usage savings.

4. Implementations

The authors have created three implementations of solvers for the Numbers Game: one in Haskell, mentioned in Section 3.2, one in Excel/VBA, and one in Delphi 7. Each uses a modified exhaustive search algorithm.

The Haskell implementation is the most straight-forward and is a naive implementation of brute force generation of all possible solutions, without generating intermediate solutions representing invalid expressions as described previously. It takes about 40 seconds on a modern architecture to solve the Numbers Game for 6 operands, and uses a
great deal of memory. This is because new structures in Haskell are created from old ones by copying, and the forests of expressions trees are stored in ordinary lists. No effort was made to use Haskell structures more efficiently, which is certainly possible. The Excel/VBA model is again a brute-force implementation, but expressions are stored as string values in arrays, as opposed to tree structures in lists. This implementation also considers commutative expression equivalence and does not generate commutatively equivalent expressions. It may take up to about 2 minutes to solve the Numbers Game on a modern architecture, although many examples from the Letters & Numbers program are solved in less than 20 seconds.

The Delphi implementation is a direct translation of the Excel/VBA model into Delphi, again using string values to store expressions and arrays to store collections of expressions. Interestingly, it ran about 100 times as fast as the VBA/Excel model, generating and checking for feasibility all possible 2,398,368 candidate trees/expressions in about 2 seconds on a modern architecture running Windows 7. Its significant out-performance of the Haskell implementation is due to its more compact and efficient string storage of expressions and use of arrays for collections rather than lists. Otherwise, all implementations are essentially the same algorithm.

5. Teaching Applications

The rich mathematical and computational structure of the Numbers Game make it amenable to use as an example in a wide variety of teaching applications for Discrete Mathematics, Programming, and Complexity Analysis. Such application areas are summarised in Table 2 and discussed here.

5.1. Arithmetic expression trees

A topic we believe is accessible to first year students with only modest background in algebra is that of arithmetic expression trees (AETs). Such trees show the recursive structure of arithmetic expressions and allow abstract representation as a recursive structure (binary tree). This canonical form may then be flattened into a linear form (prefix, infix or postfix string) using various recursive traversal algorithms. A simple, stack-based algorithm can be used for evaluation of postfix expressions, where parentheses are not necessary to preserve meaning. A conventional infix expression becomes ambiguous when more than one binary operator is used. Only by convention does multiplication have precedence over addition, and parentheses are used to force order of evaluation, if necessary. The important distinctions between form and content, syntax and semantics can be pointed out.

5.2. Counting and Recurrences

Determining the size of the Numbers Game problem for a given number of operands is a non-trivial exercise in counting, as discussed in Section 3.1. It provides a practical example of using factorial in counting, and also introduces Catalan numbers. Analysing the number of different tree structures also serves as a vehicle to see the recursive nature of (expression) trees and provides a good example for studying recurrence relations. Even in a weaker class where such topics are not formally included, the Numbers Game offers an opportunity to showcase some of the principles of recursion and recurrence relations, and various means of computing them, and even a brief glimpse at their asymptotic properties [5].

5.3. Data structures and algorithms

Expression tree creation and manipulation is standard fare for a data structures class. The Numbers Game is a simple application requiring binary expression trees. Evaluating an expression tree is an example of post-order traversal. Creating the forest of possible
expression trees for a given set of numbers is an interesting application of permutation generation and recursion.

Also common in data structures classes is using a stack for postfix expression evaluation. Again, the Numbers Game is a good example on which to apply this algorithm. If the possible expressions are considered as strings, then the problem can be solved by generating the possible permutations of operators and operands, pushing them on stacks, and applying the stack evaluation algorithm. The (recursive) approach to generating all permutations may require the management of a collection of stacks-in-the-making, if added complexity is desired.

Finally, the collection of operands can require some linear structure such as an array or list, so solving the game can require multiple data structures and interactions amongst them.

5.4. Languages and language theory

The language of possible expressions in the Numbers Game has very interesting aspects, which makes it a very useful example for teaching concepts of language theory. A simplified version of the game with a small number of operands (say, 2 or 3) can be used for examples of simple finite-state automata, but as examples that are not artificial sequences of a's and b's, as so many are. It may also be a useful example for instructors to discuss relative merits of regular expression notation and finite-state automata. The fact that the language is finite gives the theoretical result that it is a regular language, but it is a regular language without a satisfying representation using either regular expressions or finite-state automata, and can thus serve as a segue into more powerful means of defining languages.

Binary expressions are a classic example used when teaching context-free grammars (CFGs), as they require the interesting notions of precedence and associativity. Initially, it seems the Numbers Game language is just such an expression grammar, but the requirement that the operands be used only once introduces a multiple counting requirement that CFGs cannot represent. Again, this is a useful example for instructors to demonstrate the limitations of CFGs, and the need for yet more expressive language definition tools. So, again, there is a nice segue into the next level, attribute grammars, representing context sensitivity.

Finally, the Numbers Game poses the very interesting language question of generating strings in the language, rather than merely recognising and parsing them as discussed above. Such generation, of course, forms the basis of the brute force method of solving the problem computationally. This is a very challenging programming exercise in typical third generation/object-oriented languages like C or Java, requiring sophisticated use of several data structures and an algorithm for creating permutations of the operands. Such an exercise would be suitable for an advanced programming class. Alternatively, this example can also be used to highlight advantages of programming languages more suitable to the task with richer type mechanisms (such as Haskell), or with more language support for generating structures based on constraints (such as Prolog). This would be an excellent exercise for a comparative programming languages course.

5.5. Computational complexity and advanced programming

The Numbers Game is an interesting teaching example in computational complexity analysis for two reasons. First, the time cost analysis requires calculation of the number of possible expression trees, and therefore interesting application of Catalan numbers and recurrence relations. Secondly, it is an example with very significant space cost, which is not so common in standard data structures examples. The Numbers Game is an example for students where,

\(^1\) Linear in the number of possible expression trees.
even though the time cost may seem reasonable, the space cost definitely prompts exploration of more efficient solutions.

**Table 2: Summary of Teaching Applications**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counting and problem size</td>
<td>Discrete Mathematics</td>
</tr>
<tr>
<td>Discrete structures and trees</td>
<td>Discrete Mathematics</td>
</tr>
<tr>
<td></td>
<td>Data Structures and Algorithms</td>
</tr>
<tr>
<td>Expression trees</td>
<td>Discrete Mathematics</td>
</tr>
<tr>
<td></td>
<td>Data Structures and Algorithms</td>
</tr>
<tr>
<td>Catalan numbers</td>
<td>Discrete Mathematics</td>
</tr>
<tr>
<td>Expression evaluation (Stacks and Trees)</td>
<td>Data Structures and Algorithms</td>
</tr>
<tr>
<td>Tree traversal</td>
<td>Data Structures and Algorithms</td>
</tr>
<tr>
<td>Finite State Automata</td>
<td>Discrete Mathematics</td>
</tr>
<tr>
<td></td>
<td>Programming Languages and Compilers</td>
</tr>
<tr>
<td>Regular Expressions</td>
<td>Discrete Mathematics</td>
</tr>
<tr>
<td></td>
<td>Programming Languages and Compilers</td>
</tr>
<tr>
<td>Computational complexity</td>
<td>Theoretical Computer Science</td>
</tr>
<tr>
<td></td>
<td>Data Structures and Algorithms</td>
</tr>
<tr>
<td>Formal grammars</td>
<td>Programming Languages and Compilers</td>
</tr>
<tr>
<td>Parsing and generation</td>
<td>Programming Languages and Compilers</td>
</tr>
<tr>
<td>Memory allocation and de-allocation</td>
<td>Programming Languages and Compilers</td>
</tr>
<tr>
<td>Garbage collection</td>
<td>Programming Languages and Compilers</td>
</tr>
<tr>
<td>Memoisation</td>
<td>Programming Languages and Compilers</td>
</tr>
</tbody>
</table>

Algorithmic aspects of more efficient solutions have been discussed in Section 3.2, but additional memory manipulation aspects can be applied to the problem: explicit memory allocation/deallocation, garbage collection, and memoisation. These are important topics in advanced programming or compiler design classes, and again the Numbers Game serves as a useful running example to explore them. Apart from providing a programming application for these concepts, these aspects of improving efficiency provide a useful comparison between programming languages and a forum for discussing the current trend in programming languages away from explicit memory management and away from explicit pointers into memory. For example, in Java one cannot control memory allocation and deallocation, and in, say, Haskell or Python, one cannot implement memoisation. This is not to suggest that these languages are deficient, but rather to provide an interesting demonstration of the price paid for the expressive superiority of these languages.

6. Experience using Letters and Numbers as a student assignment

In Bond University’s Bachelor of IT, there is just one mathematics unit, known as Analytical Toolkit. It consists of a typical set of introductory discrete mathematics topics, ending with a few weeks of very basic introduction to probability and statistics. In the January semester of 2012, it was decided to set an assignment for the students based on the Letters & Numbers game. Since most of the class were first semester students, and, from past experience, their mathematical backgrounds are typically very poor, this exercise had to be carefully planned. Further, most students in this class do not have any programming experience. How could we present sufficient background material for the students firstly to understand the problem, and
secondly to implement some kind of a solution? It was decided that the idea could only become feasible if we:

1. reduced the scope of the problem by relaxing the Numbers Game to 3 operands instead of 6,
2. cast the problem into an environment where at least some of the solution logic may be expressed with having to write code, and
3. supplied the class with some skeleton code which outlines an overall solution strategy.

These were achieved by putting a reduced version of the problem into Microsoft Excel 2010 with VBA code and Excel formulae and tables. For the Letters part of the problem the class was supplied with public domain word lists, again in Excel. A session on elementary VBA was also taught.

To give an idea of what was supplied to the students, and what they in turn were required to produce, Section 6.1 presents an abridged version of the suggested steps for the Numbers part for the game. Figure 1 shows a sample snapshot of the Excel spreadsheet for the Numbers Game.

![Figure 1: Excel model of Numbers Game for 3 operands](image)

Based on student feedback, this exercise was a success. Further, Table 3 presents the results of a survey of students about the assignment.

### 6.1. Numbers Game recommended steps

1. You are given a skeleton spreadsheet with essential areas already named. Please use these names, although you are free to define any extra names that you may need.
2. Note that at least some names are defined so that the VBA code may use them as a reference point for the Offset function. You may quickly check the cells determined by any name simply by clicking on it in the Name Box; alternatively use the Name Manager for complete details of any Name definition.
3. The user enters three numeric operands (numbers from 1 to 100) in A2:C2.
4. The target number must be from 1 to 100 and controlled by a slider (scrollbar).
5. Write formulae to put all permutations, without replacement, of the three input operands in the range A5:C10.
6. Put all permutations, with replacement, of the four fundamental arithmetic operations in the range E2:F17.
7. When the button Generate Expressions is clicked, your code must generate all 192 arithmetic expressions (corresponding to all binary expression trees) which may be formed from the three operands. Put these in the range H2:I193. Column H has expression; column I has the formulae which will compute the values of the expressions.

8. Column J uses an Excel lookup function to find the location of the first solution.

9. Column K uses this location to find the first solution, and displays it.

10. The user is expected to slide the scrollbar to search for solutions. When a solution appears, it must be automatically highlighted by use of conditional formatting.

11. Test your model with the input operands 3, 4, 5 and determine which targets may be generated.

### Table 3: Student survey responses

<table>
<thead>
<tr>
<th>Question</th>
<th>SA</th>
<th>A</th>
<th>N</th>
<th>D</th>
<th>SD</th>
<th># Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before the assignment, I did not think of the numbers game as a structural problem</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>It was interesting to see the numbers game as a structural problem</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Before the assignment, I did not see how a search for the solution could be constructed algorithmically</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>I could understand the algorithm, but I found coding it difficult</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>This assignment has increased my skills in applying patterns and structures to solve problems</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>I am competent at arithmetic</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>I enjoy arithmetic</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>I understand English grammar well and use it correctly</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>

### 6.2. Student feedback on the assignment

Although the class was small and the number of responses even smaller, feedback was uniformly positive, as shown in Table 3, and with the following comments.

1. “Very interesting assignment for applying algorithms to solve problems.”
2. “Found it to be quite fun and enjoyable however due to my programming background it was easy. Other new programmers might think otherwise as many people had difficulty with the syntax.”
3. “Although there was a steep learning curve I found this assignment very interesting.”

### 7. Conclusion

The games from the show *Letters & Numbers* are demonstrated to be useful vehicles for computation and mathematical concepts, especially the Numbers Game which is very rich. The treatment of the Numbers Game as a simple syntactical structure problem, rather than a complex algebraic problem, is a tremendous metaphor for how computation is and can be applied to many seemingly unsolvable or difficult to solve problems. Concepts from a wide variety of typical Computer Science and Discrete Mathematics subjects in a degree
programme apply to the numbers game, making it a useful teaching vehicle with wide application.

This paper has presented an analysis of the show’s games, along with a discussion of how it can used for teaching concepts ranging from simple counting and tree structures up to complexity analysis and advanced memory management. At every stage, the requirements of solving the Numbers Game strongly affirm why Mathematics is important in a Computer Science or Information Technology degree.

One final comment: since the time of writing of the initial draft of this paper, one of the authors has moved to a different university. Consequently, there is expected to be a much greater opportunity to test the ideas described herein with much larger numbers of students, and importantly, with many having significantly stronger mathematical backgrounds. The current plan is therefore to adapt the present material with a view to setting a more challenging assignment, and to make some attempt to measure learning.

8. References