13/1 (2015), 57–71
tmcs@math.klte.hu
http://tmcs.math.klte.hu

**Teaching
Mathematics and
Computer Science**

The far side of recursion

Michael A. Wirth

Abstract. Recursion is somewhat of an enigma, and examples used to illustrate the idea of recursion often emphasize three algorithms: Towers of Hanoi, Factorial, and Fibonacci, often sacrificing the exploration of recursive behavior for the notion that a “function calls itself”. Very little effort is spent on more interesting recursive algorithms. This paper looks at how three lesser known algorithms of recursion can be used in teaching behavioral aspects of recursion: The Josephus Problem, the Hailstone Sequence and Ackermann’s Function.

Key words and phrases: recursion, problem solving, Josephus, Hailstone, Ackermann.

ZDM Subject Classification: M50, D40, P50.

1. Introduction

Recursion in programming textbooks often exists as the quintessential triumvirate: Towers of Hanoi, Factorial, and Fibonacci. Hardly inspiring, as although they have their place in teaching recursive problem solving, after numerous decades they have become rather hackneyed and offer a limited vision of recursion. Recursion offers an exceptional example of problem solving through the principle of divide-and-conquer, calving a problem into smaller subproblems, solving them independently and combining their solutions to arrive at a solution to the original problem. The problem is that repeated use of any one particular algorithm may cause it to *wear out.* This may prompt students to gloss over the algorithms intrinsic value because the code is too run-of-the-mill. Fibonacci is a good example - no one questions the viability of the recursive algorithm because it is so pervasive in the literature. In fact, Fibonacci offers a good counter-example of recursion, at least in the context of the binary recursive version most often presented in the literature. It works well for low values, but suffers computationally when faced with large values due to the number of re-calculations performed. In short it is inefficient, but because few introductions to recursive algorithms talk about the stack, it is more difficult for students to comprehend *why* it is inefficient - due in part to the tree-based topology of the recursive algorithm. There are linear versions of the Fibonacci algorithms, but they are rarely discussed [31]. Factorials, whilst being simple in the context of linear recursion, are limited by the size of the factorial which can be calculated. In the classic book by Niklaus Wirth *Algorithm + Data Structures = Programs* [33], he cites both Fibonacci and Factorial as examples of when not to use recursion and to "avoid the use of recursion when there is an obvious solution by iteration". As both Fibonacci and Factorial can be trivially derived using iteration, students often walk away thinking recursion is a half-baked idea - "*a dangerous and unpredictable method*" in the words of Kruse [13]. The last of the trinity, Towers of Hanoi may not be very appropriate for a novice programmer, because although the problem seems easy, and the solution exudes elegance, trying to explain the solution is far more convoluted. That, and there are iterative solutions [9] which may make more sense in the first instance. Illustrative examples used in teaching recursion should possess some qualities that relate to recursive behaviors.

This paper explores the realm of three lesser known algorithms used for illustrating recursion: The Josephus Puzzle, the Hailstone Sequence and Ackermann's Function. These particular algorithms were chosen in part due to their recursive behaviors and also due to their historical significance in the computer science literature. In a survey of fourteen textbooks on C, the top four algorithms encountered were factorial (10), Fibonacci (5), Towers of Hanoi (4), and calculating a numeric sum (3). Other algorithms occur only once or twice include: greatest common divisor, recursive π, counting backwards, binary search, printing a line backwards, power, Quicksort, and directory traversal. None covered any of the three algorithms discussed in this paper. The three problems described offer a different viewpoint on solving problems in a recursive manner. None of them has any real practical application, however they can be used to illustrate many of the characteristics and behaviors of recursion.

2. The Pedagogy of Recursive Algorithms

Recursion is not a trivial topic, and is often considered to be one of the more difficult concepts for students to both understand [2, 10], and apply as a problem solving strategy [24, 11]. Whilst students often gain an understanding of programming concepts from everyday analogies, recursion offers very few such analogies [17, 30]. What does teaching recursion instruction involve? Recursion is more complex than simply having a function call itself - yet this is the intrinsic behavior portrayed in many textbooks. Using recursion as a means of solving a problem alludes to exploring a cornucopia of things: the idea of a stack and how it underpins recursion; the various forms of recursion (e.g. linear, binary, mutual, nested), and the effect of recursion on resources (computational time, memory). Too often recursion is taught by diving into a recursive piece of code, giving little weight to the rationale for using a recursive approach to the problem, its behavior, or its optimality. Textbooks present recursive algorithms in their polished forms, often ignoring the problem solving aspects [6]. Due to this students often exhibit little confidence in deriving a recursive solution from scratch, and a general indifference towards recursion as a problem solving methodology.

Eric Roberts posed that "In order to develop a more complete understanding of the topic, it is important for the student to examine recursion from several different perspectives" [18]. In this light recursion should be introduced in a multi-tiered format [32], whereby aspects of recursion are disseminated at different points throughout the curriculum. At the lowest level, this involves exploring recursion as a physical phenomenon, and how it can be used to create visual patterns (e.g. Sierpinski triangles). Then the notion of the stack can be introduced, and simple algorithms which can be explained using iterative means are re-engineered using recursion. A number of papers discuss the importance of learning to solve problems by means of iteration prior to introducing recursion [3]. Approaching the problem using an iterative solution, allows the student to leverage existing knowledge. Exploration of the *stack* - the data structure which reinforces the mechanism of recursion - provides a notion of how recursion works in the machine. This leads to a discussion of the types of recursion (e.g. linear, binary), and the exploration of puzzle-like algorithms such as the Josephus puzzle, which can be solved using iterative solutions (eg. a circular array), and extend them to recursive solutions. Such problems can also be illustrated in a physical-tactile manner. At the intermediary level, a discussion on the resources associated with recursion can be raised. This is where both the Hailstone Sequence and Ackermann's function (as well as the resource hog - binary Fibonacci) play a role - exploring processor time, stack space, and the trade-offs of recursive versus iterative algorithm versions of the algorithms . Once students have a good understanding of the behaviors associated with recursion, more challenging problems can be attempted, (e.g. Quicksort, Towers of Hanoi) and the notion of tail-recursion can be introduced. Finally, advanced topics can be cracked open - maze traversal, Sudoku, 8-Queens, and the task of removing recursion.

3. The Josephus Puzzle

The Josephus puzzle originates from Roman historian Flavius Josephus (37-100). It was mentioned in modern times by W.W. Rouse Ball in his book entitled "Mathematical Recreations and Essays" [22], first published in 1892. In the section on antique problems he discusses the idea of *decimation*, a not uncommon punishment in the early Roman Army. Rouse Ball cites the work of Hegesippus, who authored a Latin adaptation of the Jewish War of Josephus, under the title *De bello Judaico et excidio urbis Hierosolymitanae*. During the Roman-Jewish conflict of AD67, the Romans captured the town of Jotapata. Josephus and forty companions escaped and took refuge in a cave. Josephus discovered that all but himself and another man were resolved to kill themselves to prevent being captured by the Romans. Fearing to show their opposition too openly, they declared that the operation should be carried out in an orderly way, and suggested that they should arrange themselves in a circle and that counting around the circle, every third person should be killed until until there was only one survivor, who would kill himself. It is alleged that he placed himself and the other man in the 31st and 16th place respectively, thus saving their lives. The Romans spared Josephus, and he later became a Roman citizen.

The puzzle involves finding the position of the last survivor, and can be expressed as follows: A group of n people are standing in a circle, numbered consecutively clockwise from 1 to n. Starting with the first person, every kth person is removed, proceeding clockwise. The task is to determine the position of the remaining survivor, $J_k(n)$. For example, if n=6, and k=2, then people are removed in the following order 2, 4, 6, 3, 1, and the last person remaining is number 5. The process of removing people is known as *reduction using a step of size k* [19]. The puzzle was first generalized by Euler in 1775 [7], and British scientist P.G. Tait introduced a general algorithmic rule for the Josephus puzzle in 1898 [26]. The puzzle was first cited in an algorithmic context by Knuth in 1968 [12] who explored a formula for k=2. A formula can be derived for the position of the last person, values of the sequence $J_k(n)$, for a circle of n participants, in which evert kth participant is eliminated. For example $J_2(16)$ proceeds in the following fashion:

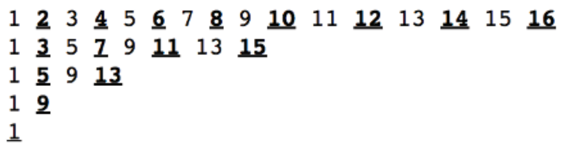

1 **2** 3 **4** 5 **6** 7 **8** 9 **10** 11 **12** 13 **14** 15 **16**
1 **3** 5 **7** 9 **11** 13 **15**
1 **5** 9 **13**
1 **9**
1

Figure 1. An example of $J_2(16)$ from a reduction viewpoint

The answer is 1 for any value of n which is a power of 2, because the sub-problem is always a multiple of 2. The recursive “breaking down” of the problem is clearly illustrated. A value of n which is not a power of 2 is more challenging - for there are now two possibilities, n being odd or even.

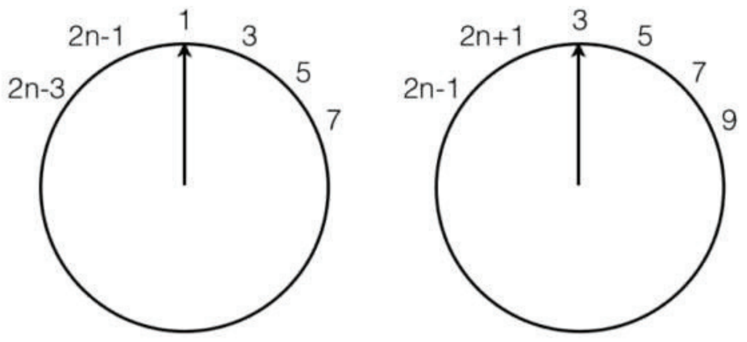
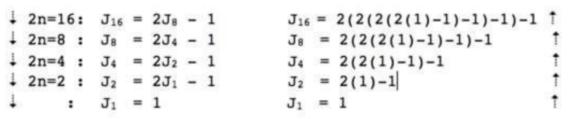


Figure 2. Even ($2n$) and odd ($2n + 1$) number of participants after the first round of elimination

If there are only $2n$ people at the beginning, then after the first cycle, only the odd numbers are left. The next person to eliminate will be 3 (Fig.2 left). This is similar to the original, except that with n people removed, a new enumeration ensues where each persons number is doubled and decreased by 1. This is the case used when n is a power of 2, as each subproblem is guaranteed to have an even number of people. Consider the example shown in Fig.3, showing the winding and unwinding of the recursive process.



$$\begin{array}{llll}
\downarrow\ 2n=16: & J_{16} = 2J_8 - 1 & J_{16} = 2(2(2(2(1)-1)-1)-1)-1 & \uparrow \\
\downarrow\ 2n=8\ : & J_8 = 2J_4 - 1 & J_8 = 2(2(2(1)-1)-1)-1 & \uparrow \\
\downarrow\ 2n=4\ : & J_4 = 2J_2 - 1 & J_4 = 2(2(1)-1)-1 & \uparrow \\
\downarrow\ 2n=2\ : & J_2 = 2J_1 - 1 & J_2 = 2(1)-1 & \uparrow \\
\downarrow\ \quad\ : & J_1 = 1 & J_1 = 1 & \uparrow
\end{array}$$

Figure 3. An example for $J_2(16)$

Alternatively, if there are $2n+1$ people originally, the persons numbered 2, 4, 6,...,$2n$, and 1 are eliminated, leaving all odd numbers *except* 1 (Fig.1 right). In this case the new numbers are doubled and increased by 1. Combining these two situations, and including the base case, the following recurrence relation evolves:

$$\begin{aligned}
J_2(2n) &= 2J_2(n) - 1(n \geq 1) \\
J_2(2n+1) &= 2J_2(n) + 1(n \geq 1) \\
J_2(1) &= 1
\end{aligned}$$

This can be easily transformed into a recursive function. Consider the following function in C.

```
int josephus(int n)
{
    if (n == 1)
        return 1;
    if (n % 2 == 0)
        return 2 * josephus(n / 2) - 1;
    else
        return 2 * josephus(n / 2) + 1;
}
```

A more generic solution which works for any value of n and k, can be formed in an iterative manner:

```
int josephusI(int n, int k)
{
    int i, pos=1;
```

```
        for (i=1; i<=n; i=i+1)
            pos = (pos + k) % i;
        return pos;
    }
```

This calculates r as the position of the last remaining person (starting at position 0) by cycling through each value of i. This can easily be converted to a recurrence relation which starts with n, and cycles back:

```
    f(1,k) = 0
    f(n,k) = (f(n-1,k) + k) mod n
```

This provides students with experience converting an iterative construct into a recurrence relation, which can then be implemented as a recursive function (with a starting position of 0):

```
    int josephusR(int n, int k)
    {
        if (n == 1)
            return 0;
        else
            return (josephusR(n-1, k) + k) % n;
    }
```

The Josephus puzzle provides a good transition from an iterative algorithm to a recursive algorithm. Many recursive problems are more challenging to solve in a non-recursive manner, however due to it’s circular nature Josephus is not that difficult using a circular array, or circular linked list. Josephus has been used by Eusebi [8] to illustrate recursion in APL2, and by Augenstein and Tenenbaum [4] as an example of program efficiency using arrays, lists and tree structures. The Josephus puzzle can also be used in the context of kinesthetic, or tactile learning, a process in which students learn by actively carrying out physical activities rather than by passively listening to lectures [23]. In this situation, students can either play with the puzzle in the manner of a board game, or perform an activity such as having students stand in a circle and re-enact the puzzle by *counting-out* aloud. Kinesthetic activities offer a way to enhance the understanding of the problem so that students can derive an algorithmic solution to the puzzle.

4. Hail Stone Sequence

The January 1984 issue of *Scientific American* contains an article on a sequence of numbers known as a the *hailstone sequence*, but also known as the *Collatz Conjecture*, the *3n+1 problem*, the *Syracuse problem*, *Hasse's algorithm*, and *Kakutani's problem.* It is credited to German mathematician Lothar Collatz in 1937, however Collatz did not publish anything on the conjecture until 1986 [5], stating that he did not publish anything earlier because he could not solve the problem. The Collatz Conjecture states that "starting from any positive integer n, repeated iteration of [the Collatz function] eventually produces the value 1" [14]. This has been shown for every number just shy of 2^{60} [21]. The series is formed using the following algorithm:

(1) Pick a number.
(2) If it's odd, triple the number and add one.
(3) If it's even, divide the number by two.

Why is the sequence likened to a hailstone? It is so named because the values are subject to multiple ascents and descents, akin to hailstone formation in a cloud [16]. Also, once the value hits a power of two, it moves towards termination in a rapid manner, as does a hailstone. As a function it can be defined in the following manner:

$$f(n) = \begin{cases} n/2 & \text{if } (n \bmod 2) = 0 \\ 3n+1 & \text{if } (n \bmod 2) = 1 \end{cases}$$

For example, starting with n = 11, the sequence takes longer to reach 1: 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. In reality, it has no real world application, but it provides an interesting exercise in the structure of recursion. Naturally, this problem can also be solved using iteration. Consider the following iterative function, which takes the number as input, and rolls out the sequence.

```
void hailstoneI(int x)
{
    printf("%d ", x);
    while (x != 1)
    {
        if (x % 2 == 1)
            x = 3 * x + 1;
```

```
                else
                    x = x / 2;
                printf("%d ", x);
            }
        }
```

Recursively, it is also easy to define, because the task of deriving a hailstone series is involved with processing a number a series of times until its value turns to 1.

```
        void hailstoneR(int x)
        {
            printf("%d ", x);
            if (x != 1)
                if (x % 2 == 1)
                    hailstoneR(3*x+1);
                else
                    hailstoneR(x/2);
        }
```

Both the iterative and recursive algorithms offer good solutions to deriving the hailstone series. Is one better than the other? If we work on the pretext that the algorithms stop after the sequence 4, 2, 1 is reached, then both work reasonably well. In this case it just relates more to computational overhead than anything else. If the value of x is set to 84, then the hailstone series calculated is:

```
        84, 42, 21, 64, 32, 16, 8, 4, 2, 1
```

Iteratively this involves executing the loop nine times, with the use of one local variable (x), and very little arithmetic overhead. Recursively, there are 10 calls to the function **hailstoneR**, each one creating its own instance of the local variable x. Run it with x=57, and we get

```
        57, 172, 86, 43, 130, 65, 196, 98,
        49, 148, 74, 37, 112, 56, 28, 14,
        7, 22, 11, 34, 17, 52, 26, 13,
        40, 20, 10, 5, 16, 8, 4, 2, 1
```

Iteratively, this uses approximately the same resources as for x=84, recursively though it's another story. There are now 33 calls to **hailstoneR**, with 33 instances of x. For x=73, there are 116 calls to **hailstoneR**, with 116 instances of x. Now a pattern begins to emerge. The recursive solution begins to draw on extraneous resources. To better see the effect we have to run the programs with the final sequence 4, 2, 1 repeating continuously. For the iterative solution there is no problem, it will likely iterate to infinity and beyond. When the recursive solution is run, the program eventually crashes.

Compiling the code in C in Xcode results in the following message, "Loading 262039 stack frames", followed by the debugger being initiated, and implying that the upper limit of recursive calls was reached. Each time we call a function, it's address is put in a *stack*, which is used to manage the function calls. When we run **hailstoneR** for the first time, it is *pushed* on to the stack. It isn't taken off until the last call the **hailstoneR** is reached, and they are all *popped* from the stack. Since the are 262,039 instantiations of **hailstoneR** on the stack, it overflows and the program crashes. This is a great example of stack overflow caused by excessively *deep* recursion, or in this case *infinite* recursion, because the program will recurse ad infinitum. Apart from a lesson in the use of stacks, it is also beneficial to experiment with how different compilers/languages deal with infinite recursion. The program is compiled with gcc, and similar programs are compiled with Python, and Fortran. Compilation with another C compiler (gcc) or Fortran (gfortran) results in the pervasive catch-all "Segmentation fault", which really does not provide much feedback for the user. Python returns with "RuntimeError: maximum recursion depth exceeded", implying the interpreter restricts the depth of the stack (typically 1000 frames).

Due to its mutually exclusive nature, Hailstone is also a good candidate to illustrate the notion of *mutual* recursion, whereby one of more functions call each other. A simplistic example is shown below which uses two additional functions to perform the tasks of calculating the hailstone sequence. The function **hs_mutual** calls either **hs_odd** or **hs_even** depending on whether x is odd or even. The function called then calls **hs_mutual** in turn, which again calls the appropriate mutual function, and so on.

```c
int hs_mutual(int x)
{
    printf("%d ", x);
    if (x != 1)
        if (x % 2 == 1)
            return hs_odd(x);
        else
            return hs_even(x);
}
```

```
int hs_even(int x)
{
    return hailstone_mutual(x/2);
}

int hs_odd(int x)
{
    return hailstone_mutual(3*x+1);
}
```

5. Ackermann's Function

Ackermann's function was originally conceived in 1928 by Wilhelm Ackermann [1], and has been used extensively in the past for studies in computational efficiency [27]. Ackermann considered a function of three variables $A(m, n, p)$, the "p-fold iterated exponentiation of m with n", a recursive function which is not primitive. This was later simplified by both Peter and Robinson [15, 19] to the more common two-variable definition. It is interesting partially because of its highly recursive nature, and does not require large integer values. Its limitation? - it doesn't really have any applications, apart from being used in the early development of systems/languages as a means of measuring performance. Sundblad [25] used Ackermann's function to explore various implementations of Algol, PL/I and Simula on various operating systems, whereas Wichmann [27] described the performance of Ackermann's function on 35 language/system pairs. This was followed by a study in 1977 [28], and another extensive one in 1982 [29].

Ackermann's function has the following recurrence relation:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \text{ and } n \geq 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

How does it work? Consider the calculation of A(1,2):

```
A(1,2)
= A(0, A(1,1))
= A(0, A(0, A(1,0)))
= A(0, A(0, 2))
= A(0, 3)
= 4
```

This seems simple, but the larger m and n become, the more complex the recursion becomes. One of the most interesting aspects about Ackermann's function is that it performs very little computation apart from the computation of A(m,0), and the recursive calls. It grows very quickly (even for small values of m and n, Ackermann(m,n) is extremely large) and it cannot be computed with only definite iteration (a completely defined **for** loop for example); it requires indefinite iteration (i.e., recursion). It can be solved iteratively using a complex stack - but this makes it limiting. A C function to implement Ackermann's function is given as follows:

```
int ackerman(int m, int n)
{
     if (m == 0)
          return(n+1);
     else if (n == 0)
          return(ackerman(m-1,1));
     else
          return(ackerman(m-1,ackerman(m,n-1)));
}
```

One of the more interesting aspects of Ackermann, from a resource perspective is that it takes some time to run, unlike many contemporary algorithms. In the early days of computing, writing efficient code was extremely important, but many algorithms which once showed different run-time profiles now are milliseconds apart - using Ackermann provides students with a better understanding of program efficiency, and the notion of *depth of recursion*. For example Ackermann(3,2) produces a recursive depth of 30. Ackermann also introduces the notion of *nested* or *embedded* recursion. For a more advanced exploration of Ackermann, it is possible to derive (i) a non-recursive algorithm, or (ii) a tail-recursive solution, and compare them to the basic recursive solution. Such experimentation amongst both different types of recursive approaches, and iterative versions allows students to explore resource *hogging*. For example, to calculate Ackermann(4,1)=65533 using recursion on a 2.5 GHz Intel Core i5 takes 24.4 seconds. The same algorithm run with the nonrecursive algorithm using a stack and unstructured jumps [20] takes 81.59 seconds. A tail-recursive version of the function runs at 17.79 seconds. A second non-recursive version of Ackermann which uses a stack, but no goto statements runs in 50.82 seconds. Students gain a greater understanding of the effect of different algorithms on computational efficiency.

6. Discussion

There is nothing inherently new about these recursive algorithms, but they have been somewhat neglected in the context of teaching recursion. In the case of the Josephus Problem, the core problem can be illustrated in both a visual and kinesthetic manner, can be solved iteratively, and then extended to form a recursive solution. Whilst offering a good visual analogy and an example of linear recursion, it also provides differing levels of difficulty. The Hailstone sequence offers a good comparison of the resources involved in iterative versus recursive solutions. It also introduces the notion of stacks and their role in recursion: stack overflow, stack frames and infinite recursion. In addition the recursive solution can be easily morphed to one illustrating mutual recursion. Ackermann’s function has been extensively used in the past in the context of benchmarking. Nowadays it is often overlooked, but apart from introducing the concept of nested recursion, it also explores resource handling. Why does this matter? Many of the approaches to teaching recursion fail to discuss underlying mechanisms such as the stack, make very little comparison with iterative methods, and blindly discuss singular approaches to recursive algorithms. Apart from learning to translate from iteration to recursion, it may be just as important to learn how to remove recursion. Ackermann is more *complicated* than traditional recursion examples, and therefore offers more challenges for recursion removal. Ironically, whilst dealing with linear, mutual and nested forms of recursion, none of these algorithms deals with *binary* recursion. In truth, binary recursion used in examples such as Tower of Hanoi can be challenging to understand, especially until students can fully comprehend the tree-like structure of the recursive calls.

7. Conclusion

Recursion is not just a way of coding problems, it doesn’t exist in a vacuum where a “function calls itself” to some end. It is a method of solving problems, which regardless of the applicability of a particular problem to real world scenarios, requires a student to think outside the box. This involves an exploration of the mechanisms that facilitate the recursive process, the resources it uses, and a discussion of whether or not the result is better that an iterative solution. Recursion is a challenging concept to teach, at any level. While Fibonacci, Factorial and Towers of Hanoi provide an insight into problem solving recursively, they are not the only algorithms available. It is time to move beyond these algorithms to the far side of recursion.

References

[1] W. Ackermann, Zum Hilbertschen Aufbau der reellen Zahlen, *Math Annalen* **99** (1928), 118–133.

[2] J. R. Anderson, R. Farrell and R. Sauers, Learning to program in LISP, *Cognitive Science* **8** (1984), 87–129.

[3] Y. Anzai and Y. Uesato, Learning recursive procedures by middle school children, *Proc. Conf. of the Cognitive Science Society* (1982), 100–102.

[4] M. Augenstein and A. Tenenbaum, Program efficiency and data structures, *ACM SIGCSE Bulletin* **9**, no. 3 (1977), 21–27.

[5] L. Collatz, On the motivation and origin of the (3n+1) problem, *J. Qufu Normal University, Natural Science Edition* **3** (1986), 9–11 (in Chinese).

[6] M. C. Er, On the complexity of recursion in problem-solving, *Int. J. Man-Machine Studies* **20** (1984), 537–544.

[7] L. Euler, Observationes circa novum et singulare progressionum genus, *Opera Omnia, Series Prima, Opera Mathematica* **7** (1923), 246–261.

[8] E. V. Eusebi, Operators for recursion, *ACM SIGAPL Quote Quad* **15**, no. 4 (1986), 190–194.

[9] R. Franklin, A simpler iterative solution to the Towers of hanoi problem, *SIGPLAN Notices* **19**, no. 8 (1984), 87–88.

[10] B. Haberman and H. Averbuch, The case of base cases: Why are they so difficult to recognize? Student difficulties with recursion, *ACM SIGCSE Bulletin* **34**, no. 3 (2002), 84–88.

[11] H. Kahney, What do novice programmers know about recursion, *SIGCHI Conf. on Human Factors in Computing Systems* (1983), 235–239.

[12] D. Knuth, *The Art of Computer Programming, v.1 Fundamental Algorithms*, Addison Wesley, 1968, 158–159.

[13] R. L. Kruse, On teaching recursion, *ACM SIGCSE Bulletin* **14**, no. 1 (1982), 92–96.

[14] J. Lagarias, The $3x + 1$ problem and its generalizations, *American Math Monthly* **92** (1985), 3–23.

[15] R. Péter, Konstruktion nichtrekursiver Funktionen, *Math. Ann.* **111** (1935), 42–60.

[16] C. A. Pickover, *Wonders of Numbers*, Oxford University Press, Oxford, 2001, 116–118.

[17] P. L. Pirollo and J. R. Anderson, The role of learning from examples in the acquisition of recursive programming skills, *Canadian Journal of Psychology* **39**, no. 2 (1985), 240–272.

[18] E. Roberts, *Thinking Recursively*, John Wiley & Sons, New York, 1986.

[19] R. M. Robinson, Recursion and double recursion, *Bull. Amer. Math. Soc.* **54** (1948), 987–993.

[20] J. S. Rohl, *Recursion via Pascal*, Cambridge University Press, 1984.

[21] E. Roosendaal, On the 3x + 1 problem, 2011, `http://www.ericr.nl/wondrous/index.html`.

[22] W. W. Rouse Ball, *Mathematical Recreations and Essays*, Macmillan and Co., 1905.

[23] P. Sivilotti and S. Pike, The suitability of kinesthetic learning activities for teaching distributed algorithms, *ACM SIGCSE Bulletin* **39**, no. 1 (2007), 362–366.

[24] R. Sooriamurthi, Problems in comprehending recursion and suggested solutions, *ACM SIGCSE Bulletin* **33**, no. 3 (2001), 25–28.

[25] Y. Sundblad, The Ackermann function, A theoretical, computational and formula manipulative study, *BIT* **11** (1971), 107–119.

[26] P. G. Tait, On the generalization of the Josephus problem, *Proc. Royal Society of Edinburgh* **22** (1898), 432–435.

[27] B. A. Wichmann, Ackermann's Function: A study in the efficiency of calling procedures, *BIT* **16** (1976), 103–110.

[28] B. A. Wichmann, How to call procedures, or Second thoughts on Ackermann's function, *Software - Practice and Experience* **7** (1977), 317–329.

[29] B. A. Wichmann, Latest results from the procedure calling test, Ackermann's function, *NPL Report DITC 3/82* (March 1982), `http://history.dcs.ed.ac.uk/`.

[30] S. Wiedenbeck, Learning recursion as a concept and as a programming technique, *ACM SIGCSE Bulletin* **20**, no. 1 (1988), 275–278.

[31] M. Wirth, Fibonacci beyond binary recursion, *Teaching Mathematics and Computer Science* **6**, no. 1 (2008), 173–185.

[32] M. Wirth, A multi-tiered pedagogical framework for teaching recursion, 2014, unpublished manuscript.

[33] N. Wirth, *Algorithm + Data Structures = Programs*, Prentice-Hall, 1976.

MICHAEL A. WIRTH
SCHOOL OF COMPUTER SCIENCE
UNIVERSITY OF GUELPH
GUELPH, ONTARIO N1G 2W1
CANADA

E-mail: `mwirth@uoguelph.ca`

(Received July, 2014)