Teaching Recursion in High School
A Constructive Approach

Dennis Komm[1,2](✉)

[1] PH Graubünden, Chur, Switzerland
dennis.komm@phgr.ch
[2] Department of Computer Science, ETH Zürich, Zürich, Switzerland

Abstract. There is little doubt about both the importance and at the same time difficulty of teaching recursion as part of any sophisticated programming curriculum. In this paper, we outline an approach that has its focus on introducing the concept in very small steps, integrating recursion into a K-12 programming class. The main paradigm is to be as constructive as possible in that everything that is introduced is also implemented right away from the first lesson on.

Keywords: Programming education · Recursion · K-12 · Turtle graphics

1 Introduction

Many computational problems have elegant solutions that utilize *recursion*, such as the famous Towers of Hanoi [16]. As a result, recursion should be part of informatics K-12 education as a rather natural application of *problem decomposition* (in particular, *divide-and-conquer*), which in turn is a cornerstone of computational thinking [9,15]. However, the topic is not only important but also rather challenging for novice programmers [4], and therefore a careful introduction is necessary.

In this paper, we propose a concrete roadmap, assuming a setting where students have prior knowledge about imperative programming.

There is a wealth of literature about examples that describe how to teach recursion – or how to "think recursively" – in a high school context, some of which dates back decades by now; see Rinderknecht [13] for a survey. Many of these approaches focus on the idea of introducing divide-and-conquer as a first motivation, practicing this technique of decomposing a given problem into subproblems "unplugged," and after that possibly using a computer to put the ideas into code, see, for instance, Hauswirth [5], Anderle et al. [1], or Sysło and Kwiatkowska [17].

Without criticizing these approaches, in this paper we would like to offer an alternative strategy, which – although we are speaking about a top-down concept – somewhat follows a bottom-up approach. In a nutshell, we would like to be as "hands-on" as possible in that we put programming in the center, and only move to more complex algorithms once terms like *base case* are understood and have been successfully implemented. Thus, we start with very simple programs that very carefully introduce core concepts one after the other, and have the students put everything they learned into code as soon as possible. In particular, the concept of divide-and-conquer is only introduced at the very end, at a point where the students are able to actually implement the ideas described on a high level.

The individual ingredients used in this paper are well known; our main contribution is a detailed recipe that puts them into context. We give step-by-step instructions on how all concepts involved can be taught in a constructive fashion. The result is an exemplary lesson plan with examples and exercises that enable the students to master basic recursion in small steps.

Didactically, our approach is in parts inspired by the PRIMM [14] principle, although we do not always include all of its steps. The students are given code (that can be executed as it is presented), reason about it, modify single parts of it to understand what it does, and only after that write their own (similar) programs.

The following points are crucial to our approach.

– Introduce one concept after the other in small steps.
– Apply new concepts as soon as possible.
– Use complete examples that can be executed as they are presented.
– Consequently build on the students' preknowledge.
– Build bridges to known concepts.

2 Road Map

In the main part of the paper, we describe a road map that consists of seven steps. Within these steps, the single aspects that make up recursive programming are described. Whenever a new aspect is introduced, it is immediately applied by the students by writing (Python) code, which is why we call our approach "constructive." We indeed follow many of Papert's *constructionist* ideas including programming the Turtle in the first steps of our lesson plan [11,12].

The lesson plan can be subdivided into seven steps, with a zeroth that is prepended in the first 15 min. A single lesson takes 45 min.

0. Give a quick outlook on what is going to happen (Lesson 1).
1. Introduce recursion using an example (Lesson 1).
2. Link recursion to something known (Lesson 2).
3. Point out the importance of a base case (Lesson 3).
4. Combine recursion with return values (Lesson 4).
5. Allow multiple calls per function body (Lesson 5–6).
6. Implement known algorithms recursively (Lesson 7).
7. Introduce divide-and-conquer (Lessons 8 and on).

Two non-trivial aspects of recursive programming are (1) having a function return a value and processing it recursively, and (2) having more than one recursive call in a single function body. We introduce both of them carefully, and even avoid them completely in the first steps. Only in step 4 values are computed and returned to the caller, and only in step 5 the programs actually branch. In step 7, both concepts are combined for the first time.

In the following subsections, we give a quick overview over the students' programming experience that is assumed, and then describe the seven steps in more detail.

2.1 Preknowledge

Our lesson plan is designed for grades 11 or 12 on the premise that the students already have some experience with programming in Python. In particular, they can define simple functions that include conditionals and loops, use variables, and implement some standard algorithms such as Binary Search and Bubblesort. We have been paying special attention to the introduction of *variables*. First, we have introduced parameters as a kind of "read-only"-variables and only after that considered variables changing their value during execution. The reason is that many misconceptions arise in particular when transferring the concept of variables from mathematics to informatics [7].

Additionally, the students are familiar with Turtle graphics, in particular the `gturtle` library, which is part of the TigerJython environment [18,19], which the students have used for two to three years by the time this lesson plan should be applied.

We would like to remark that functional programming is not part of this curriculum, and that recursion is therefore introduced after the students gained experience with imperative programming. Dynamic data structures, which may have a recursive nature, have also not been discussed. The students use Python lists utilizing both random access and dynamic aspects, but without having discussed any details of the implementation.

The idea of computational complexity was covered informally by, for instance, reasoning about the number of elements that Binary Search inspects given a sorted Python list of `n` elements. Big-O notation and other formal tools have not been introduced.

2.2 Give a Quick Outlook on What Is Going to Happen

Before any concepts are introduced formally, we excite the students' curiosity by giving them an idea of what is covered within the following weeks. However, it is not our goal to dive into recursive algorithms right away, but only to build up some tension about an interesting and powerful tool that will be introduced and used. We therefore show them pictures of fractals and point out their self-similar structure. It is obvious that these pictures are hard to design with the programming concepts the students know at this point. But lo and behold! Not

Listing 1: A first recursive program

```
def f():
    print("Hello world!")
    f()
```

Listing 2: Using a parameter

```
def f(k):
    print(k)
    if k > 0:
        f(k-1)
```

Fig. 1. Introducing recursion using two simple programs

only will the new tool allow for drawing such pictures with rather little code, but also for solving many interesting problems in a clever and elegant way.

However, before the students can use this exciting tool, they have to understand the way it works, which is what the following seven steps (subsections) are devoted to. For now, there will not be more motivation, but we will soon get back to the things recursion brings to the table.

2.3 Introduce Recursion Using an Example

We start by asking the students what they think happens if a function "calls itself" by introducing a very short toy example of a recursive function without parameters as shown in Listing 1 (Fig. 1). Note that it may be necessary to adjust some of the metaphors we have used so far. Defining a function may have been thought of as "teaching the computer a new word," but this analogy makes somewhat less sense if the definition of a word again contains this word.

Studying Listing 1, we discuss with the students what will happen when executing such a program by expanding it using the blackboard, and then running it on a computer. There are two insights: (1) the program will theoretically run forever, but also (2) this does not seem to be prohibited; so far, there is no obvious reason to write code this way, but also nothing seems to prevent us. The students do, however, know that algorithms are supposed to work in finite time, and thus such behavior is at least somewhat undesired. (Conversely, they are not familiar with the concept of a call stack at this point, and we also do not address the finiteness of the computer's memory.)

So we ask the students what to do about having such a program "run forever," and they already know the answer, namely to use a parameter with a value that is changed with every call. We show them Listing 2 (Fig. 1), asking them to study the code and reason what it does, then to manually execute the program by hand, and finally using the computer in order to see whether they were right.

2.4 Link Recursion to Something Known

The second step aims at demystifying recursion by building a bridge to known concepts. First, we use the Turtle in order to draw simple shapes. The reason is that the Turtle gives us a tangible notional machine [3,6,8] that can be observed while executing the code. Ideally, the students are familiar with the Turtle since primary school where another programming language, for instance, Logo, was

Listing 3: Infinite spiral

```
1 def line(d):
2     forward(d)
3     right(90)
4     line(d+3)
```

Listing 4: Finite spiral

```
1 def line(d):
2     if d <= 300:
3         forward(d)
4         right(90)
5         line(d+3)
```

Listing 5: Finite spiral

```
1 def line(d):
2     while d <= 300:
3         forward(d)
4         right(90)
5         d += 3
```

Fig. 2. Drawing a spiral recursively and with a loop

used instead of Python. The Turtle is navigated using the self-explanatory commands `forward()`, `backward()`, `left()`, and `right()`. Using the `gturtle` library, no object-orientation is needed; the command `makeTurtle()` creates a canvas with the Turtle in its center, which then executes the commands step-by-step.

The students are now presented Listing 3 (Fig. 2) while being asked to figure out what it draws. The task is again to execute the algorithm by hand and then use the computer afterwards. Executing the code leads to the Turtle drawing an infinite spiral. Applying what was learned in Sect. 2.3, the students are now asked to set the maximum side-length of the spiral to 300 pixels, which can be easily done by inserting an **`if`**-statement in line 2 of Listing 3, resulting in Listing 4 (Fig. 2). This is the same strategy as used in Listing 2, which avoids explicitly defining a base case (although it would not hurt).

Second, we compare Listing 4 to Listing 5 (Fig. 2), which draws the same spiral, but uses no recursion but a loop, pointing out that the main difference is that the variable `d` is now increased in line 5 instead of passing its increased value to an according recursive function call. Further exercises practice converting simple recursive code to non-recursive code and vice versa; an example is drawing a number of steps as shown in Listings 6 and 7 (Fig. 3).

Of course, we have to be careful at this point, as the equivalency to loops is not that easily established in general, but only works smoothly for such simple *tail recursions.*

2.5 Point Out the Importance of a Base Case

So far, we have not used the term *base case* and at the same time avoided Python's **`return`**-statement – although the latter is known to the students. However, novices struggle a lot with the difference between **`return`** and **`print`**`()`. Before actually returning values in the context of recursion, we therefore make an intermediate step and use **`return`** in order to end function calls. The reason is to very explicitly study *what* is actually ended, namely only the current function call, not all calls to this function, and not the whole program.

As an example, consider Listing 7, which was created in the preceding step. We show the students that this code can be rewritten in a way that makes the non-recursive case, the base case, more explicit; see Listing 8 (Fig. 3). Here we use

Listing 6: Iterative stairs

```python
def step(d):
  while d >= 1:
    forward(50)
    right(90)
    forward(50)
    left(90)
    d -= 1
```

Listing 7: Recursive stairs

```python
def step(d):
  if d >= 1:
    forward(50)
    right(90)
    forward(50)
    left(90)
    step(d-1)
```

Listing 8: Base case

```python
def step(d):
  if d == 0:
    return
  forward(50)
  right(90)
  forward(50)
  left(90)
  step(d-1)
```

Fig. 3. Drawing stairs with a loop and recursively

return to simply end the function call, which again can be linked to something known, namely using Python's **break** command within a "**while** True"-loop.

Also here, tasks are given to the students afterwards to consolidate what they have just learned. However, the students also already know that **return** does not simply end a function call but that there is usually an expression appended, which is first evaluated and then returned. (As a matter of fact, **return** returns None in Python, which is not discussed in class at this point.)

2.6 Combine Recursion with Return Values

Now the students understand that functions can call themselves, and by making sure that there is a non-recursive base case that is guaranteed to be called at some point, an infinite execution can be prevented. For the next step, we leave the Turtle for a short time and present the code displayed in Listing 9 (Fig. 4), which is the standard example of recursively computing the factorial of a natural number n. (Note that we have deliberately not defined the factorial of zero, and the **else**-statement in line 4 is not necessary.)

However, instead of starting with this function, we only present it after having discussed thoroughly what happens when functions call themselves. In this step, the first task for the students is to again reason about what the program does without actually executing it. The results are discussed in class, and the algorithm is executed by hand for small values of n. Together with the students, we sketch how the functions are called, and how the computed values are passed

Listing 9: Computing the factorial recursively

```python
1 def fact(n):
2   if n == 1:
3     return 1
4   else:
5     return n * fact(n-1)
```

Listing 10: Computing the sum recursively

```python
1 def sum(n):
2   if n == 1:
3     return 1
4   else:
5     return n + sum(n-1)
```

Fig. 4. Recursive implementations of the factorial and sum

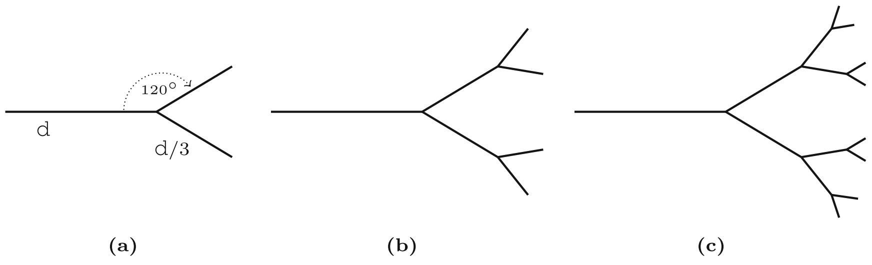


Fig. 5. Three binary trees with increasing depths

to the caller; we do still not formally speak about the concept of a call stack here. After that, we let the students investigate single components of Listing 9, asking them, for instance, what they think happens if we change "**return** 1" to "**return** 2" in line 3, etc.

Only after that, the students write their own code, which involves returning values, but sticks with one recursive call per function body. A sample task is to compute the sum of the first `n` natural numbers recursively, which is of course quite similar to computing the factorial of `n`; see Listings 9 and 10 (Fig. 4).

2.7 Allow Multiple Calls per Function Body

In this step, we return to the Turtle and use it to draw fractals, redeeming our promise from Sect. 2.2. In our approach, the students do not have to learn any new tool or advanced programming techniques besides recursion itself; everything can be done using the very basic movement commands of the Turtle introduced earlier.

Drawing fractals now for the first time adds concrete value to the new technique. Indeed, it is quite obvious that the factorial of some number can also be computed with a loop instead – the similarities between both concepts were even discussed explicitly in step 2 (for such simple cases). As for computing the sum, there is even a closed formula, which may be known by some of the students. However, drawing self-similar shapes using loops is not straightforward at all, but quite elegant using recursion.

We again proceed in small "sub-steps." The first task is to draw the simple structure in Fig. 5a using a simple function with a single parameter `d`, making sure the Turtle starts and ends at the left. Then, using this function, the second task is to draw the shape in Fig. 5b, and the third task to use this function to draw the shape in Fig. 5c.

Of course, no recursion is needed to draw any of these shapes, but our goal is to have the students identify a pattern. The next task therefore asks them to compare the three solutions in Listings 11 to 13 (Fig. 6), study where they differ and what they have in common, before finally trying to use this insight to define a recursive function. More specifically, we ask them to design a Python function `tree()` with two parameters `d` and `n`, the first of which denoting the length of

Listing 11: Binary tree 1

```
def tree1(d):
    forward(d)
    left(30)
    forward(2*d/3)
    back(2*d/3)
    right(60)
    forward(2*d/3)
    back(2*d/3)
    left(30)
    back(d)
```

Listing 12: Binary tree 2

```
def tree2(d):
    forward(d)
    left(30)
    tree1(2*d/3)

    right(60)
    tree1(2*d/3)

    left(30)
    back(d)
```

Listing 13: Binary tree 3

```
def tree3(d):
    forward(d)
    left(30)
    tree2(2*d/3)

    right(60)
    tree2(2*d/3)

    left(30)
    back(d)
```

Fig. 6. Drawing the three trees from Figs. 5a to 5c, respectively

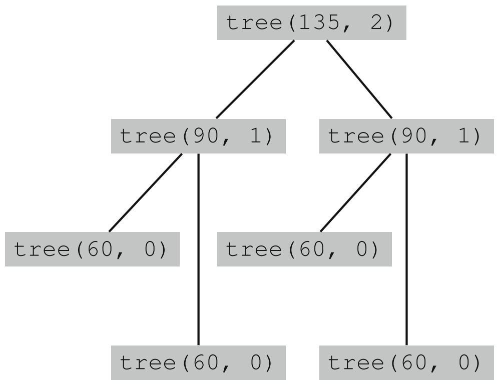
the first line (the “trunk” of the tree), and the second one the *depth* of recursion; see Listing 14 (Fig. 7).

This is the first time that the students are confronted with the concept of *branching*, that is, they now design functions with more than one recursive function call that is executed within a function body. Understanding what is happening here is crucial to master recursion. Therefore, the subsequent task asks the students to trace the calls when executing a concrete call to their new function, for instance, `tree(135, 2)`. The insight is that this also results in a binary tree (see Fig. 7a), whereas computing the factorial leads to a sequence of calls that could be arranged in a linear structure.

The floor is now open to draw further fractals such as, for instance, the Koch curve [10]. Note that we did not start with this example as the first line is only drawn in this example after a sequence of recursive calls, while Listing 14 draws

Listing 14: Recursive tree

```
def tree(d, n):
    if n == 0:
        forward(d)
        backward(d)
        return
    forward(d)
    left(30)
    tree(2*d/3, n-1)
    right(60)
    tree(2*d/3, n-1)
    left(30)
    backward(d)
```



(a)

Fig. 7. Drawing binary trees recursively

Listing 15: Euclid's algorithm

```python
def euclid(a, b):
  while b != 0:
    if a > b:
      a = a - b
    else:
      b = b - a
  return a
```

Listing 16: Euclid's algorithm recursively

```python
def euclid(a, b):
  if b == 0:
    return a
  else:
    if a > b:
      return euclid(a - b, b)
    else:
      return euclid(a, b - a)
```

Listing 17: Binary search

```python
def bsearch(data, x):
  l = 0
  r = len(data) - 1
  while l <= r:
    c = (l + r) // 2
    if data[c] == x:
      return c
    elif data[c] > x:
      r = c-1
    else:
      l = c+1

  return -1
```

Listing 18: Binary search recursively

```python
def bsearch(data, l, r, x):
  if l <= r:
    c = (l + r) // 2
    if data[c] == x:
      return c
    elif data[c] > x:
      return bsearch(data, l, c-1, x)
    else:
      return bsearch(data, c+1, r, x)
  else:
    return -1
```

Fig. 8. Known algorithms and their recursive counterparts

something in any function call. This makes it a lot easier for the students to watch the Turtle execute their code. There is rich literature about other fractals and how to design creative pictures using recursion [10,16].

2.8 Implement Known Algorithms Recursively

The students have now gained some experience with recursive programming, and with fractals also encountered an example where this technique offers an obvious advantage over programming using iteration. For the last two steps, we move to algorithm design with the goal to use recursion to implement divide-and-conquer strategies in step 7 (Sect. 2.9).

Before we get there, however, we again build a bridge to something known in this step. As mentioned earlier, the students know about basic algorithms that have been implemented using loops so far. As an example, consider Euclid's algorithm to compute the greatest common divisor of two numbers `a` and `b`. We studied a very simple version without the modulo operator, as shown in Listing 15 (Fig. 8). We compare this known variant of the algorithm to the recursive implementation in Listing 16 (Fig. 8). Together with the students, we identify

Listing 19: Merging two sorted lists

```python
def merge(leftlist, rightlist):
  result = []
  i_left = 0
  i_right = 0
  while i_left < len(leftlist) and i_right < len(rightlist):
    if leftlist[i_left] < rightlist[i_right]:
      result.append(leftlist[i_left])
      i_left += 1
    else:
      result.append(rightlist[i_right])
      i_right += 1
  return result + leftlist[i_left:] + rightlist[i_right:]
```

a pattern that is similar to what we discovered when studying Listings 4 and 5 namely that the functions have in essence a very similar structure, but instead of explicitly decreasing the values of the variables `a` and `b` in lines 4 and 6 (Listing 15), respectively, these exact values are passed when calling the function recursively in lines 6 and 8 (Listing 16), respectively.

The next task revisits the Binary Search algorithm as presented in Listing 17 (Fig. 8). As this algorithm is, although somewhat easy to describe on a high level, very hard to implement [2], the single steps are recalled before asking the students to implement a recursive version as shown in Listing 18 (Fig. 8). The solution again follows the same structure as the original implementation, but instead of decreasing `r` in line 9 or `l` in line 11 (Listing 17), the values are accordingly passed in lines 7 and 9 (Listing 18), respectively.

2.9 Introduce Divide-and-Conquer

Only now we consider the students ready to have a close look at the divide-and-conquer strategy – this is our last step, not the first. In order to demonstrate the power of this technique, we start with the well known Mergesort algorithm. As mentioned in Sect. 2.1, the students already have some basic knowledge about sorting a Python list of `n` numbers in increasing order using, for instance, the Bubblesort algorithm. Depending on the level of the class, some simple observations about the number of swapping operations may have been made, exhibiting a quadratic complexity.

So we now return to the topic of sorting `n` numbers and present the observation to the students that "merging" two sorted lists to a single sorted list can be implemented in a straightforward fashion with two pointers, comparing their smallest elements, and inserting the smaller of the two at the end of an initially empty list. After thoroughly discussing this strategy, we ask the students to implement it in Python, resulting in a function as shown in Listing 19.

Listing 20: Mergesort recursively

```
 1 def mergesort(data)
 2   if len(data) <= 1:
 3     return data
 4   mid = len(data) // 2
 5   leftlist = mergesort(data[:mid])
 6   rightlist = mergesort(data[mid:])
 7   result = []
 8   i_left = 0
 9   i_right = 0
10   while i_left < len(leftlist) and i_right < len(rightlist):
11     if leftlist[i_left] < rightlist[i_right]:
12       result.append(leftlist[i_left])
13       i_left += 1
14     else:
15       result.append(rightlist[i_right])
16       i_right += 1
17   return result + leftlist[i_left:] + rightlist[i_right:]
```

With this, the center piece of Mergesort is built, and we discuss in class how the merging can be applied recursively in order to sort a given list. This is done on the blackboard in an unplugged fashion, but as soon as it is understood, the students can use their new tool to transfer the ideas into code. The result is shown in Listing 20. Essentially, all that needs to be done is to rename the function from Listing 19, slightly alter it to take only a single list as parameter (line 1), take care of the base case (empty lists and lists with one element can be considered sorted, lines 2 and 3), then split the given list in half using Python's known *slicing* notation, and apply the algorithm to the two resulting lists recursively (lines 4 to 6).

The complexity of this algorithm can be discussed on a high level, without introducing mathematical recursive functions, but arguing about the size of the tree representing the calls. Note that Listing 20 can be implemented in an even less cumbersome manner using Python's `pop()` function to take the smaller of the two elements out of the respective list instead of using the two pointers `i_left` and `i_right`. However, since `pop(0)` has linear time complexity, this would reduce studying Mergesort to absurdity.

At this point, all concepts have been introduced to write recursive code, and in this step, all have been combined in order to implement a fast sorting algorithm. Now the students are ready to investigate other examples of recursive implementations of the divide-and-conquer strategy or study other problems with natural recursive solutions such as listing all words of a fixed length over some fixed alphabet or the initially mentioned Towers of Hanoi.

Acknowledgement. The author thanks Tobias Kohn for many interesting discussions that improved the paper.

References

1. Anderle, M., Forišek, M., Steinová, M.: Teaching recursion and dynamic programming before college. Bull. EATCS 129 (2017)
2. Bentley, J.: Programming Pearls, 2nd edn. O'Reilly Media, Newton (1999)
3. du Boulay, B., O'Shea, T., Monk, J.: The black box inside the glass box: presenting computing concepts to novices. Int. J. Man-Mach. Stud. **14**, 237–249 (1981)
4. McCauley, R., Grissom, S., Fitzgerald, S., Murphy, L.: Teaching and learning recursive programming: a review of the research literature. Comput. Sci. Educ. **25**(1), 37–66 (2015)
5. Hauswirth, M.: If you have parents, you can learn recursion. Bull. EATCS 123 (2017)
6. Hromkovič, J., Kohn, T., Komm, D., Serafini, G.: Combining the Power of Python with the Simplicity of Logo for a Sustainable Computer Science Education. In: Brodnik, A., Tort, F. (eds.) ISSEP 2016. LNCS, vol. 9973, pp. 155–166. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46747-4_13
7. Kohn, T.: Variable evaluation: an exploration of novice programmers' understanding and common misconceptions. In: Proceedings of SIGCSE 2017, pp. 345–350 (2017)
8. Kohn, T., Komm, D.: Teaching Programming and Algorithmic Complexity with Tangible Machines. In: Pozdniakov, S.N., Dagienė, V. (eds.) ISSEP 2018. LNCS, vol. 11169, pp. 68–83. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02750-6_6
9. Komm, D., Hauser, U., Matter, B., Staub, J., Trachsler, N.: Computational Thinking in Small Packages. In: Kori, K., Laanpere, M. (eds.) ISSEP 2020. LNCS, vol. 12518, pp. 170–181. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-63212-0_14
10. Liss, I.B., McMillan, T.C.: Fractals with turtle graphics: a CS2 programming exercise for introducing recursion. In: Proceedings of SIGCSE 1987, pp. 141–147 (1987)
11. Papert, S.A.: Mindstorms: Children, Computers, And Powerful Ideas, 2nd edn. Basic Books, New York (1993)
12. Papert, S.A.: Introduction: what is logo? And who needs it? In: Logo Philosophy and Implementation, pp. IV-XVI. Logo Computer Systems (1999)
13. Rinderknecht, C.: A survey on the teaching and learning of recursive programming. Inform. Educ. **13**, 87–119 (2014)
14. Sentance, S., Waite, J.: PRIMM: exploring pedagogical approaches for teaching text-based programming in school. In: Proceedings of WiPSCE 2017, pp. 113–114. ACM (2017)
15. Wing, J.M.: Computational thinking. Commun. ACM **49**(3), 33 (2006)
16. Stephens, R.: Essential Algorithms: A Practical Approach to Computer Algorithms Using Python and C#. Wiley, New York (2019)
17. Syslo, M.M., Kwiatkowska, A.B.: Introducing Students to Recursion: A Multi-facet and Multi-tool Approach. In: Gülbahar, Y., Karataş, E. (eds.) ISSEP 2014. LNCS, vol. 8730, pp. 124–137. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09958-3_12
18. TigerJython. https://tigerjython.ch. Accessed 5 July 2021
19. WebTigerJython. https://webtigerjython.ethz.ch. Accessed 5 July 2021