Issues in Information Systems
Volume 25, Issue 3, pp. 357-370, 2024

DOI: https://doi.org/10.48009/3_iis_2024_127

Dealing with the challenges of learning recursive programming – helpful functions and incremental approaches to encourage recursive thinking

Dave Smith, *Virginia Military Academy*, *smithdtr@vmi.edu*
Pankaj Chaudhary, *North Carolina A&T State University*, *pchaudhary@ncat.edu*
Azad Ali, *University of Fairfax*, *aali@ufairfax.edu*

Abstract

Computer programming courses are known to challenge many students, especially in their first programming courses. Some topics in computer programming are more challenging than others due to the complexity of the logic underlying the concepts. One such topic that is considered challenging is recursive programming or recursion. Recursion as a teaching topic has long been a point of discussion, especially in relation to how it can be taught effectively. This paper delves into the topic of recursive programming and strategies applied to teach it recursively in a four-year institution. It examines the factors that make recursion challenging to learn and some strategies to teach recursion. The strategy pursued here stems from previous studies using data recursion and functional programming, which uses recursion by default. The data structure of 'list' from the LISP programming language, which is apt to data recursion, is implemented in Python and used for solving problems. The effectiveness of the strategy is assessed. Results indicate that the use and understanding of recursive programming can be enhanced by using recursive data structures and having the students use them in more practical applications. This hopefully will encourage other instructors to use the same/similar strategies to teach recursion more effectively.

Keywords: recursive programming, recursive thinking, recursive learning challenges

Introduction

Recursion is a fundamental technique for solving and coding problems in the computer science discipline. However, it is often challenging for novices since it requires tracing non-linear and hierarchical sequences of execution (Thorgeirsson et al., 2024). Recursion is based on hierarchical function calls. The function provides the solution for a base case, and for a case other than a base case, the function calls itself for a solution to a divided and reduced case. There may be multiple function calls when the given case is divided and reduced. The expectation is that each branch will ultimately reach a base case, and all the function calls will ultimately return to the initial place of call to produce a result. The simplest example of recursion is the calculation of factorial where the base case is factorial(1) = 1. factorial is written as a function. Cases other than base cases are decomposed and reduced progressively. So factorial(8) = 8 * factorial(7), and so on. Each case that is not the base case results in a series of hierarchical function calls. At the bottom of the hierarchy, the function call is made to the base case of 1, which returns 1. The calls then successively return to the first function call where the final result is produced. In other problems, there may be multiple function calls, like in the case of a binary tree, where a call is made to the left child and the right child, which themselves may be binary trees. These strategies for solving a problem are sometimes also known as divide

and conquer strategies. It is also a representative computational thinking strategy that exemplifies, for example, Piaget's notion of reflective abstraction (Cetin & Dubinsky, 2017), but also decomposition, pattern recognition, and algorithmic problem-solving.

Recursion is fundamental to solving a class of problems where each case can be successfully decomposed and reduced. These problems are a natural candidate for the use of recursion for a solution. A solution based on the traditional imperative programming concepts is complex and difficult to implement. The Tower of Hanoi (Rubio-Sanchez, 2017), binary tree traversal, graph traversal, merge sort, quick sort, etc., are examples of algorithms that lend themselves naturally to the use of recursion. In the data structures area, recursion is a powerful tool for solving algorithmic problems. The use of recursive solutions becomes natural when dealing with data structures, which can be conceptualized as recursive. Trees, linked lists, and graphs are the common data structures that are usually taught in an introductory data structures class.

There are several benefits of teaching recursion, the most important of which is that it orients students to solve by dividing the problem into manageable sub-parts. It also makes them more familiar with the mathematical concept of induction. The bottom line is that solutions to a class of problems and dealing and operations with some types of data structures are simply better conceptualized using recursion. Hence students need to learn and understand recursion.

Problem Statement

Recursion is considered a difficult topic for novices to grasp (Anderson et al., 2014). The stumbling block may primarily be because the introduction to programming starts with iterative thinking, and by the time students are introduced to recursion in an introductory programming course or a data structures course, iterative thinking is somewhat entrenched. More so, in terms of a problem-solving approach, most students do not approach the problem with a divide-and-conquer mentality. Also, the fact that some problems can be solved without the use of recursion induces the students not to think recursively.

Sanders et al. (2006) studied various mental models of recursion that students develop. They analyzed students' performances in comprehending and demonstrating the trace of a recursive process. They studied aspects of the flow of control as it is passed forward to new activations, reaches base cases, and is passively returned to the caller at the conclusion of an activation. In conclusion, they stated, "Students struggle to understand recursion, and we need to find good ways to teach the concept.".

From an educator's viewpoint, the problem in the context of recursion boils down to arriving at a set of strategies that can be employed to teach recursion to students effectively. The This manuscript provides a possible strategy towards this end.

Background

There have been several studies on strategies regarding teaching recursion to computer science students (AlZoubi et al., 2015; Ginat & Shifroni, 1999; Goldwasser & Letscher, 2007; Mackay, 2022; Thorgeirsson et al., 2024). Bruce et al. (2005) advocate and champion the use of structural recursion as opposed to functional recursion. They argued that with the advent of object-oriented programming techniques, the teaching of recursion should shift. Emphasis should be on simple recursive structures as linked lists and methods that process them. They emphasize that this should be done before students are introduced to the traditional procedural examples.

To increase the impact of teaching or recursive thinking, they also emphasize that structural recursion should be introduced to the students before the introduction of data structures like arrays. An additional motivation here was also to ingrain the students in object-oriented thinking and not slip back to procedural

thinking. Hence, it seems like structural recursion can achieve the dual aim of teaching recursion effectively and reinforcing object-oriented concepts before the students get entrenched in procedural thinking. Structural recursion brings the important object-oriented features to the fore, too. When using structural recursion in an object-oriented language, an object does not call itself but rather invokes a method on another tangible object that happens to belong to the same class. Each object has its own state information and typically, each has a single active function at any one time (Goldwasser & Letscher, 2007).

Functional programming languages like LISP and its modern dialects Clojure, Scheme, and common LISP use recursion by default (Hinsen, 2009) and rely heavily on recursive data structures. Recursive thinking is the foundation of functional programming. Functional programming language works on the concept of a mathematical function. A mathematical function produces the same output given the same input. A mathematical function returns a value and does not have any side effects like writing to a file or memory. As a result, functional programming languages do not use variables. Functional programming languages also do not use a loop since nothing can be changed between iterations. Recursion replaces loops in a functional language.

LISP stands for list processing and was invented in the 1950s for the use of artificial intelligence. LISP is based on a flexible data structure of nested lists. Given that functional programming languages replace a loop with recursion, it would appear that the introduction of recursion using functional languages instead of imperative languages may be a strategy to follow.

However, due to the proliferation of imperative languages and their applications and then the consequent difficulties in transitioning the thinking to a functional programming paradigm, the next best thing that can be undertaken is using a recursive data structure like the nested list LISP and implementing it using a custom class. Different list operations or methods are defined using recursion. Such an approach can be implemented using any imperative, object-oriented programming language. A language that students are already familiar with can be employed, and instruction in a functional language like LISP can be avoided.

What follows is an explanation of the basic data structures in LISP and the operations/functions on those data structures. LISP has two basic kinds of objects: atoms and lists. These are mutually exclusive with the exception that an empty list or a “nil” list – () is both a symbol and a list. Lists are constructed recursively from atoms, and hence, a given list may contain atoms or other lists as members (Allen & Dhagat, 2020). Examples of lists include: (), (a), (a sam 30), and ((PC 40) 10 9 (10 30)). Primitive list functions in LISP allow selection from the list and construction of the list. The important constructor functions are *cons*, *list*, and *append*. The selector functions are *first* and the *rest*. Following is a brief description of the ‘list’ functions. The discussion is useful when the implementation in an imperative language is considered.

The list in LISP is a clear example of a recursive data structure. For example, enumerating all the elements of a list can be done using a recursive call to the ‘first’ function. Due to the nature of LISP as a functional language, the use of recursion for loops, and the list being treated as a recursive data structure, LISP and the ‘list’ data structure are excellent candidates for introducing and teaching recursive thinking.

However, in the experience of the authors, introducing a functional language like LISP is difficult after the introduction of an imperative language like Python which nowadays is typically the first programming language taught in computing programs. Python, with its emphasis on code readability, simplicity, and the Pythonic way of doing things, makes students somewhat resistant to learning LISP with its functional roots. To work around this issue Python was used to implement a recursive data structure similar to LISP lists.

Table 1: Classic LISP list data structure functions

LISP List Function	Output
'cons' adds a member to the front of the list. # (cons 1 nil) # (cons 2 (cons 1 nil)) # (cons 'a '(b c d))	 (1) (2,1) (a b c d)
'list' makes a list of all arguments. # (list 1 2 3 4) # (list 'a '(b c))	 (1 2 3 4) (a (b c))
'append' is normally used with lists and combines the lists # (append '(a b) '(c d))	 (a b c d)
'first' works on a list and returns the first element of the list as a symbol or list. # (first '((a s) d f) # (first (1 2 3 4))	 (a s) 1
'rest' takes a list as an argument and returns the list minus the first element # (rest (1 2 3)) # (rest '((a s) (d f)))	 (2 3) ((d f)

This study focuses on using Python to implement a list class and implement all its operations using structural recursion. Since Python contains a list class as a basic data structure, using a recursive list as a class is somewhat simplified since the students are familiar with the nature of a list and what it can achieve. Once the students have defined the class, it is used throughout the semester since students learn best when fundamental ideas are enforced through repetition and use in different contexts. The use of a recursive list class using Python has also been implemented by Goldwasser and Letscher (2007). However, they do not report any results on the effectiveness of their approach.

Survey of Introductory Computer Science Textbooks

A survey of some popular textbooks was undertaken to ensure that the approach taken here (though previously practiced by some researchers) is unique and not covered in existing introductory computer science textbooks. The objective was to ensure that the approach being considered was not already covered in an existing text.

1. "Java How to Program, Early Objects" (Deitel & Deitel, 2017) is a leading text covering Java programming. The topic of recursion is presented very late in the text. It is covered in chapter 18, following more than 700 pages in which iterative approaches are repeated. Examples of recursion include factorial, Fibonacci, Towers of Hanoi, Fractals, and backtracking. Exercises cover several different contexts, including palindromes, eight queens, and maze traversal.

2. "Introduction to Java Programming and Data Structures" (Liang, 2019) is another leading text for Java. Like in the Deitel text, the topic of recursion is discussed very late in chapter 18. Following a very brief introduction to recursion, the text presents case studies of factorial, Fibonacci, palindrome,

recursive selection sort, directory size, Tower of Hanoi, and Fractals in an expository manner. The text includes a discussion of recursion vs. iteration and a brief introduction to tail recursion. Exercises include computation of various mathematical sequences, count of characters, numeric base conversions, knights tour game, maze traversal, and recursive tree rendering.

3. "Data Structures and Problem-Solving using Java" (Weiss, 1998) presents recursion in chapter 7. The chapter covers "what is recursion," relation to proofs by induction, and basic recursion. Examples include printing number is any base, Fibonacci sequence, fractals, greatest common divisor, and other mathematical functions. The text is widely recognized as an excellent text for data structures. However, it is heavy in mathematics and tends towards advanced concepts such as dynamic programming. Exercises cover a wide range of different contexts ranging from cryptosystem, permutations, expanding C++ including directives, longest increasing sequence, and Knoch star.

4. "Data Structures Abstraction and Design Using Java" (Koffman & Wolfgang, 2021) is another text frequently adopted for a course in data structures. Recursion is presented in Chapter 5 and begins with a section, "Recursive Thinking," followed by a discussion on tail recursion vs. iteration. Examples include recursive linear search, binary search, a few recursive string functions, and several linked list processing functions. Case studies include Towers of Hanoi, counting cells in a blob, and a path through a maze. Exercises include conversion between bases, several linked list problems, finding roots, permutations, and merge sort.

5. "The Scheme Programming Language" (Dybvig, 2009) is a premier text for the Scheme programming language, a dialect of Lisp. With Scheme being a functional language that relies heavily on tail recursion, recursion is introduced in Chapter 2 as the primary technique for operating on the Scheme's foundational list data structure. Simple introductory examples include list length, list copy, member, remove members, and absolute all members. Exercises include append, make list, a form of get, and list comparisons. Recursion is further expounded upon in Chapter 3 with examples to sum a list, factorial, Fibonacci, efficient Fibonacci, processing expressions as a list, and many more. A noteworthy in the introduction of recursion: "It can be tricky to master recursion at first, but once mastered it provides expressive power far beyond ordinary looping constructs". The examples and exercises lay solid ground for mastering recursion. The downside of Scheme language is its unfamiliar syntax, emphasis on lambda expressions, and terminology of car, cdr, and cons as foundational list operations.

In summary, all the texts examined cover the basics of recursion and include many of the same simple examples: factorial, Fibonacci, base conversions, maze traversals, fractals, and simple list processing functions. Except for Dybvig, most exercises are in different contexts, each of which must first be understood before tackling the problem.

Kaufman and Wolfgang do include an exercise for several link list problems but seem to lack sufficient definition of the context. Dybvig, by far, provides the best coverage of recursion with ample examples and exercises to master recursion; however, the unfamiliar syntax, use of lambda expression, and terminology impede use as an introduction to recursion.

This summary provides some support to the approach being taken in this manuscript in using a recursive data structure implemented in an imperative language and encouraged to be used on multiple occasions in the course.

Recursion Implementation Using Recursive Lists

The problem domain is a list data structure with a restrained set of foundational functions. A list can be thought of as having a head element and the rest of the elements as a body, the body itself being a list with a head and a body. An example of a list containing elements 3, 9, 4, 2, and 5 is shown in Figure 1.

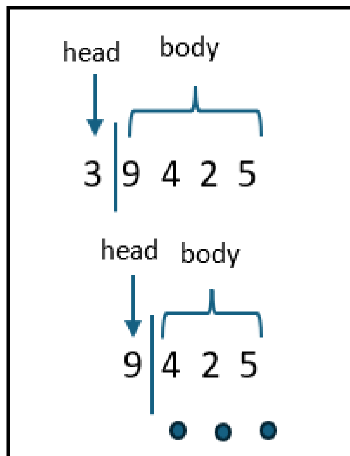

Figure 1: Example view of list 3,9,4,2,5.

Figure 2 presents an alternate conceptual view illustrating the recursive structure. Here, the head precedes a body enclosed in a box. The list with 3 as the head has 9, 4, 2, 5 as the body. List with 4 as the head has 2, 5 in the body. List with 5 as the head has an empty body. This is similar to the recursive list data structure in LISP.

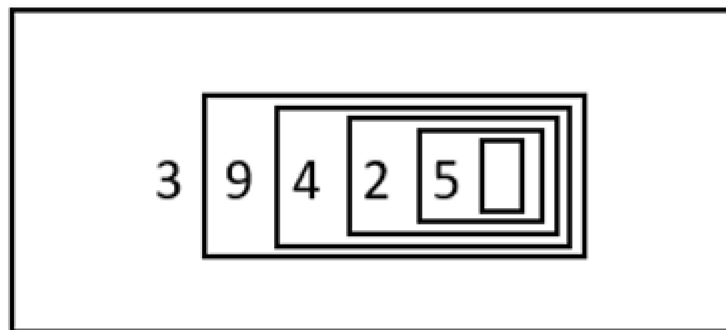

Figure 2: Recursive conceptual view of list 3, 9, 4, 2, 5.

The foundational functions are: *Ls* to create a list, *head* to return the first element of a list, *body* to return the body as a list, *empty* to test if a list is empty, *push* to create a new list with an element as the head and another list as the body, and an *isLs* to test if a variable is an instance of an *Ls* list. In addition, a str function is provided to enable the conversion of a list to a string for printing. The synopsis for these functions is given in table 1. These functions are implemented as a Python module called Ls.

A sample interactive session executing these is given in Figure 3. It should be noted that *Ls* lists are immutable. The result of the push function creates a new *Ls* list, but the list passed as an argument is unchanged. The creation of list z toward the end of Figure 3 illustrates that elements of a list can be

themselves *Ls* lists, thus forming a nested structure. The str function is implicitly called by Python's *print* function.

Table 2: Foundational functions of the Ls list domain

Ls(…)	Construct a list using a variable number of arguments; each argument will be an element in the list. Calling Ls without any arguments returns an empty list.
head(x)	Returns the head element of list x
body(x)	Returns the body of list x as a list. The body could be empty.
push(e, x)	Create new list with e as the head and list x as the body.
empty(x)	Returns True if x is an empty list, otherwise False
isLs(x)	Returns True if x is a list, otherwise False
str(x)	Returns a string representation of a list

```
> from Ls import *            # import all functions of the Ls module
> x =Ls(3, 9, 4, 2, 5)        # create an Ls list assign it to variable x
> print(x)
(3,9,4,2,5)
> print(head(x))
3
> print(body(x))
(9,4,2,5)
> print(head(body(x)))
9
> y = push(0,x)               # create a new Ls list with 0 pushed as the head with x as the body
> print(y)
(0,3,9,4,2,5)
> print(x)                    # list x is not changed by the push of 0 onto x
(3,9,4,2,5)
> print(empty(x))
False
>print(empty(body(body(body(body(body(x)))))))
True
> e = new Ls()                # create an empty list, has no head or body
> print(e)
()
> f = push('A', e)  #create a new Ls with 'A' pushed as the head and empty list e as the body
> print(f)
('A')
> print(e)
()
> print(isLs(x))
True
> print(isLs(1))              # the integer 1 is not an Ls list
False
> z = Ls('A', 10, Ls('one','two', x), Ls(), 'B')  # create an Ls list with nested Ls lists
> print(z)
('A',10,('one','two',(3,9,4,2,5)),(),'B')
```

Figure 3: Sample interactive session using functions of the Ls module.

Sample Problems for Recursive Thinking

The sample domain, though simple, provides a wealth of problems that can readily be solved using recursion. Some sample problems are presented below, and their solutions are presented in Figure 4.

1. A ***count*** function to return the number of elements in a list can be solved recursively by first considering an empty list, in which case the count is 0. If not empty. the number of elements in a list will be more than the number of elements in the body. The number of elements in the body is obtained by recursively calling count passing the body of the list.

2. A ***get*** function to return an element of a list at a given index position will involve an additional consideration: error conditions due to invalid parameters. Calling *get* on an empty list is an error condition, as there are no elements that can be returned from an empty list. Furthermore, without defining the meaning of a negative index, a negative index would be an error. Next, if the index is zero, return the head of the list. Otherwise, return the element by recursively calling *get*, passing the body of the list, and a decremented index.

3. Before presenting functions to return a modified list, first consider how to perform a copy of a list. A ***copy*** function is to return a copy of a list. Here, if the list is empty, create and return a new empty list. Otherwise, push the head of the list on the copy of the body of the list. A copy of the body is obtained by recursively calling *copy* and passing the body.

4. An ***append*** function creates a new list. The new list, which is a copy of the original list with an element appended, can be obtained through a minor modification to the *copy* function. The *append* function takes an additional argument, which is the element to be appended. To append to an empty list, return a new list using the element. Otherwise, like *copy*, *push* the head of the list on the list resulting from recursively calling *append,* passing the body of the list and the element.

5. Other functions to return a modified list, such as ***remove*** and ***insert***, involve combinations of concepts used in the *index* function and *copy* function.

6. The ***startsWith*** function, to determine if a list starts with all the elements of another list, involves several cases. First, if the other list is empty, then return “True”, as all lists start with an empty list. Conversely, return “False” if the list is empty, as an empty list cannot start with a non-empty list. If the heads of each list are equal, then recurse passing the body of each list. Otherwise, return “False”.

7. Relational functions, such as ***equal*** or ***lessThan*,** involve similar logic structures as *startsWith* with minimal changes.

8. The ***indexOf*** function introduces a strategy for solving another class of recursive problems. It is a public function, a form that, when directly called, does not perform recursion. Instead, it initiates a call to another internal form of the function where recursion occurs. The alternate internal form is indicated by a name beginning with an underscore (‘_’). The internal form ***_indexOf*** takes the index as an argument in addition to the original list and a target list. If the list *startsWith* the target list, the index is returned. If the list is empty, then -1 is returned, as the target cannot be found. Otherwise, *_indexOf* is recursively called with the arguments of an incremented index, the body of the list, and the target list. Recursion is initiated by the public *indexOf* function with an argument of 0 for the index.

9. The ***reverse*** function is another problem involving a public form and an internal form ***_reverse***. *_reverse* is similar to *copy* except the new list is constructed while descending into recursion as opposed to coming out. Here the public *reverse* calls the *_reverse* passing a new empty list on which the reversed list is constructed.

10. The ***sum*** function, to sum all number elements in a list, works with nested lists. That is, the *head* could be a list. If a list is empty, then 0 is returned. If the *head* is a number, then return the addition of the head with the sum of all elements in the *body*. If the *head* is a list, return the addition of the sum of all elements in the *body* with the sum of all elements in the *head*. Here, recursion is performed on both the *body* and the *head*. If the *head* is neither a number nor a list, then return the sum of all elements in the body.

11. The ***flat*** function to inline nested lists (e.g. flat(Ls(1,Ls('A', 'B', C'), 2)) returns a list (1, 'A', 'B', C', 2) involves a public form and internal form ***_flat***. Like *reverse*, the public calls *_flat* passing the source list and a new list on which the flat list is constructed. If the source list is empty, the flat list is returned. If the head for the source list is a list, then recursion is performed on both the *head* and the *body*, passing the returned list from recursion on the *body* as the flat list when recursing on the *head*. If the *head* is not a list, then *push* the *head* on the list returned by recursing on the *body*.

Many other operations can be visualized and implemented using recursive lists and recursive calls. Figure 5 lists a number of these operations along with a short description. To teach students recursion, a combination of functions outlined above and in Figure 5 can be assigned to the students. They can be assigned in stages, with the simple operations being implemented first, followed by the more complex operations. Since Python is used and students are familiar with the syntax, the focus is purely on recursive strategies to solve the problem (this syntactic familiarity is retained with other languages, too). Figure 6 shows a class implementation in Java (which was employed by the authors before the raging popularity of Python).

```
def count(ls):
  if empty(ls):
    return 0
  return count(body(ls)) + 1

def get(ls, index):
  if empty(ls) or index < 0:
    raise Exception()
  if index == 0:
    return head(ls)
  return get(body(ls), index – 1)

def copy(ls):
  if empty(ls):
    return Ls()
  return push(head(ls), copy(body(ls)))

def append(ls, e):
   if empty(ls):
     return Ls(e)
```

```
def indexOf(ls, target):
  return _index(0, ls, target)

def _reverse(ls, revLs):
  if empty(ls):
    return revLs
  return _reverse(body(ls), push(head(ls), revLs))

def reverse(ls):
  return _reverse(ls, Ls())

def sum(ls):
  if empty(ls):
    return 0
  if isNumber(head(ls))
    return head + sum(body(ls))
  if isLs(head(ls))
    return sum(head(ls)) + sum(body(ls))

def _flat(ls, flatLs):
  if empty(ls):
```

<table>
<tr>
<td>

```
   return push(head(ls),append(body(ls)))

def startsWith(x, y):
  if empty(y):
    return True
  if empty(x):
    return False
  if head(x) == head(y):
    return(startsWith(body(x), body(y)
  return False

def _indexOf(index, ls, target):
  if startsWith(ls, target):
    return index
  if empty(ls):
    return -1
  return _index(index + 1, body(ls), target)
```

</td>
<td>

```
    return flatLs
  if isLs(head(ls))
    return _flat(head(ls), _flat(body(ls), flatLs))
  return push(head(ls), _flat(body(ls),flatLs))

def flat(ls):
  return _flat(ls, Ls());
```

</td>
</tr>
</table>

Figure 4: Solutions to Sample Problems.

eq(x,y) and other relational comparisons	Returns True if x equals y, where x and y can be Ls lists	> print(eq(Ls(1,2,3), Ls(1,2,3))) True > print(eq(Ls(1,2,3), Ls(1,2,1))) False
insert(e, ls, index)	Returns a list with element e inserted into list l at index position	> print(insert(99, Ls(1,2,3),2) (1,2,99,3)
remove(ls, index)	Returns a list with element at index position removed	> print(remove(Ls(1,2,99,3),2) (1,2,3)
concat(x,y)	Return a list with list x concatenated with list y	> print(concat(Ls(1,2),Ls(3,4)) (1,2,3,4)
sublist(ls, index)	Return a sublist containing the element at the index position and all subsequent elements.	> print(sublist(Ls(1,2,3,4),2)) (3,4)
sublist(ls, start, end)	Return a sublist containing the element at the start index position and all subsequent elements up to but exclusive of element at the end index position.	> print(sublist(Ls(1,2,3,4),1,3)) (2,3)

endsWith(ls, endLs)	Returns True is the last elements are the same as the end list	> print(endsWitht(Ls(1,2,3),Ls(2,3))) True > print(endsWitht(Ls(1,2,3),Ls(2,4))) False
lastIndexOf(ls, target)	Returns the index of the last occurrence of the target list	> print(endsWitht(Ls(1,2,1,3),Ls(1))) 2
repeat(ls, times)	Returns a list by repeating a list a given number of times	> print(repeat(Ls(1,2,3),3)) (1,2,3,1,2,3,1,2,3)
replaceAll(ls, src, rpl)	Returns a list with all source element replace with a replacement element	> print(replaceAll(Ls(1,Ls(2,1)3),1,5))) (5,(2,5),3)
max(ls) and other aggregates	Returns the maximum element of a list	> print(max(Ls(2,1,Ls(3,5),4)) 5
fullCount(ls)	Returns the number of elements including elements in nested lists	> print(fullCount(Ls(2,1,Ls(3,5,7),4)) 6

Figure 5: More List Problems to be Solved Using Recursion.

```
package cis213.util;
import java.util.NoSuchElementException;
public class Ls<E> {
       private E car;
       private Ls<E> cdr
       private Ls(E car, Ls<E> cdr) {
              this.car = car;
              this.cdr = cdr;
       }
       private Ls(int i, E... parms) {
              if (i < parms.length) {
                     car = parms[i];
                     cdr = new Ls<E>(i + 1, parms);
              } else {
                     car = null;
                     cdr = null; // Ls with an empty cdr is the end marker node }}
       public Ls(E... parms) {
              this(0, parms);     }
       public E car() {
              if (cdr == null) {
                     throw new NoSuchElementException(); }
              return car;  }
       public Ls<E> cdr() {
              if (cdr == null) {
                     throw new NoSuchElementException(); }
```

```
            return cdr;  }
      public Ls<E> cons(E car) {
            return new Ls<E>(car, this);     }
      public boolean isEmpty() {
            return cdr == null; }
      public String toString() {
            return "(" + toRemainingString();      }
      private String toRemainingString() {
            return isEmpty() ? ")" : car.toString() + (cdr.isEmpty() ? "" : ",")
                        + cdr.toRemainingString();}
      public int size() {
            if (isEmpty()) {
                  return 0;
            } else {
                  return 1 + cdr.size();    }      }
     public boolean equals(Object other) {
            if (!(other instanceof Ls)) {
                  return false;      }
            Ls<?> otherLs = (Ls<?>) other;
            if (isEmpty() && otherLs.isEmpty()) {
                  return true;
            } else if (isEmpty() || otherLs.isEmpty()) {
                  return false;
            } else if (car.equals(otherLs.car())) {
                  return cdr.equals(otherLs.cdr);
            } else {
                  return false; }} }
```

Figure 6: Java Implementation of Recursive List and Operations (Documentation removed)

Results from Class Implementation

The recursive list data structure and its associated functions/operations have been assigned to the students in a Data Structure and Algorithms course over the last five years at two four-year institutions. Initially, the recursive list data structure was implemented using Java with foundation functions/operations expressed as methods. In its original form, the assignment adopted terminology directly from Lisp and Scheme. That is the names for the foundational methods user were *car*, *cdr*, and *cons* (refer to Figure 6) instead of *head*, *body*, and *push* used with Python. Given the somewhat more complex syntactical structure of Java and its stricter conceptual grounding, the exercise was better received when implemented in Python.

The assignment is given in a lecture introducing the terminology, concepts, and foundational methods. The source file for the Ls class is distributed to the class and then executed in a demonstration of each foundational method, as exemplified in Figure 2. The *copy*, *startsWith*, and *indexOf* methods are then solved in class with student participation. The *copy* method lays the foundation that new lists are constructed on the return of recursion. The method *startsWith* is a method that involves several cases to consider. The *indexOf* introduces the idea of a public method and an internal method (in Figure 4 such methods begin with an underscore). It is the internal method in which recursion occurs in *indexOf*. The public form simply calls the internal method with appropriate data. The development and demonstration of these methods is performed in an Integrated Development Environment (IDE). The IDE includes a debugger, enabling execution to be suspended on each recursive call. On suspending execution, the call

stack and internal variables of the Ls list are examined and explained. The class is then assigned a set of 10 to 12 selected problems to be completed outside of class. Hints are provided for most of the problems.

Overall, while students have initially struggled with this assignment, on completion of the assignment most express the problems have succeeded in promoting recursive thinking. The use of Python facilitates the focus on concept rather than syntax is a bonus. The use of the terms *car*, *cdr*, and *cons* was an initial impediment to understanding the problem domain, and that has been alleviated with the use of *head* and *body*. Once an understanding of the concepts has solidified and the first few problems have been solved, progress through the remaining problems improves significantly. The students demonstrated better recursive thinking as opposed to when they were taught the traditional recursive algorithms like factorial, tower of Hanoi, and others. The success of this enhanced understanding stems from the use of recursive data structures and associated operations. Direct comparative measurement of this was not undertaken due to the complexity of introducing both the traditional methods and then the recursive objects in a single class and the apparent inability to remove bias due to the order effect bias (Hausman, 2003). However, the performance on assignments and student participation unambiguously points to a better understanding of using recursive objects/data structures.

Conclusions and Future Work

This paper posits and presents a recursive data structure of a list and its associated functions/operations to teach recursion. This is based on previous research in this area to solve the difficulty of understanding recursion amongst novice computer science students and designing effective ways of promoting recursive thinking. The approach presented here using Python is relatively simple. The operations to be defined recursively for the list data structure involve a small number of statements, with most functions or operations being less than 10 statements. While each problem may initially be a challenge, the concept of recursion takes hold, and recursive thinking is engaged, as each operation is defined recursively in code.

These problems have been used as an assignment in a data structures course for over 5 years with success in promoting recursive thinking. Currently, the measurement of success is acknowledged to be subjective. However, it has face validity across two universities and multiple instructors. Efforts are underway to design and conduct formal objective measurements using exercises pre and post-intervention where the intervention is the proposed teaching method. The students will be taught recursion using the traditional textbook method and solve a sample of problems, then the intervention will be introduced, and students will solve a sample of problems post-intervention. An increase in understanding of the concept of recursion will be measured using the quality of code. Potential collaboration with other computer science instructors is being sought and explored. If the objective measurement supports the teaching method, then the hope is that this will make it into the traditional computer science textbooks teaching recursion.

References

Allen, C., & Dhagat, M. (2020). *Lisp Primer.* https://colinallen.dnsalias.org/lp/lp.html

AlZoubi, O., Fossati, D., Di Eugenio, B., Green, N., Alizadeh, M., & Harsley, R. (2015, September). A hybrid model for teaching recursion. In *Proceedings of the 16th Annual Conference on Information Technology Education* (pp. 65-70).

Anderson, J. R., Pirolli, P., & Farrell, R. (2014). Learning to program recursive functions. In *The nature of expertise* (pp. 153-183). Psychology Press.

Bruce, K. B., Danyluk, A., & Murtagh, T. (2005). Why structural recursion should be taught before arrays in CS 1. *ACM SIGCSE Bulletin*, *37*(1), 246-250.

Cetin, I., & Dubinsky, E. (2017). Reflective abstraction in computational thinking. *The Journal of Mathematical Behavior*, *47*, 70-80.

Deitel, P. J., & Deitel, H. M. (2017). Java how to program, early objects (11th ed.). Pearson Education Limited.

Dybvig, R. K. (2009). *The Scheme programming language*. MIT Press.

Ginat, D., & Shifroni, E. (1999). Teaching recursion in a procedural environment—how much should we emphasize the computing model?. *ACM SIGCSE Bulletin*, *31*(1), 127-131.

Goldwasser, M., & Letscher, D. (2007, October). Teaching strategies for reinforcing structural recursion with lists. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion* (pp. 889-896).

Hausman, J. (2003). Sources of bias and solutions to bias in the consumer price index. *Journal of Economic Perspectives*, *17*(1), 23-44.

Hinsen, K. (2009). The promises of functional programming. *Computing in Science & Engineering*, *11*(4), 86-90.

Koffman, E. B., & Wolfgang, P. A. (2021). *Data structures: abstraction and design using Java*. John Wiley & Sons.

Liang, Y. D. (2019). *Introduction to Java Programming and Data Structures, 12th edition.* Pearson, 2

Mackay, S. (2022, March). What Does Literature Tell Us About Recursion?. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 2* (pp. 1173-1173).

Rubio-Sanchez, M. (2017). *Introduction to Recursive Programming*. CRC Press

Sanders, I., Galpin, V., & Götschi, T. (2006, June). Mental models of recursion revisited. In *Proceedings of the 11th annual sigcse conference on innovation and technology in computer science education* (pp. 138-142).

Thorgeirsson, S., Lais, L. C., Weidmann, T. B., & Su, Z. (2024, March). Recursion in Secondary Computer Science Education: A Comparative Study of Visual Programming Approaches. In *Proceedings of Deitel, Paul & Deitel, Harvey, "Java How to Program Early Objects", 10th edition, Pearson, 2015.*

Weiss, M. A. (1998). Data structures and problem solving using Java. *ACM SIGACT News*, *29*(2), 42-49..