REPORT

お茶の水女子大学自然科学報告　第70巻　第1号、第2号
Natural Science Report, Ochanomizu University, Vol. 70, No. 1, No. 2 (2020)

Improving students' understanding of mathematical induction via programming

Youyou Cong and Akiko Mito

Received January 25 2019

Abstract

Mathematical induction is one of the most challenging topics in high school math. As reported by previous studies, students may write correct proofs by induction without understanding why their proofs are valid. In this thesis, we seek for a way to help students' learning of induction by means of computer programming, in particular, by teaching the concept of recursion. Our contributions are three-fold. First, we design an introductory programming course for high school students, which aims to enhance their comprehension of induction through game programming. Second, we investigate students' conceptual understanding of induction, using a specially designed paper test. Third, we propose a novel approach to teaching recursion, where the teacher explicitly shows the correspondence between recursive functions and proofs by induction.

1 Introduction

Mathematical induction is a technique that allows us to prove propositions over natural numbers. Here is a typical example of a proof by induction:

Prove that the following proposition holds for any natural number n.

$$1 + 3 + 5 + ... + (2n - 1) = n^2 \quad ... (*)$$

Proof: We prove the proposition by induction on n.

(1) Suppose $n = 1$. The left-hand side of $(*)$ is 1, which is equal to 1^2.
Therefore $(*)$ holds.

(2) Suppose $n = k + 1$ and $(*)$ holds when $n = k$.
The left-hand side computes as follows:

$$\begin{aligned} & 1 + 3 + 5 + ... + (2k - 1) + (2(k + 1) - 1) \\ &= k^2 + 2k + 1 \\ &= (k + 1)^2 \end{aligned}$$

Therefore $(*)$ holds.

By (1) and (2), we conclude that $(*)$ holds for all natural numbers n. $\square$

Mathematical induction is one of the most challenging topics in high school math. Aichi Prefectural Education Center conducted a survey in 40 high schools and reported that many teachers and students have difficulties in teaching or learning mathematical inductions [1]. More specifically, they find it hard to explain or understand how to prove inequalities by induction, as well as how induction works. The challenge of teaching and learning mathematical induction has been reported in other countries as well. Palla et al. [8] investigated students' understanding of mathematical induction at three high schools in Greece and reported that more than half of the students were not able to write a complete definition of mathematical induction. To help students understand how induction works, math teachers have been using various models when introducing this proof technique. The most popular models are domino tiles and Hanoi towers; other real-world examples compare inductive reasoning to ladder climbing or a frog's pond crossing [10, 16]. However, some teachers teach induction just as "steps to follow", because they think induction is conceptually difficult for high school students [8]. As a consequence, students may write correct proofs without really understanding why their proofs are valid.

Meanwhile, computer scientists have been attempting to enhance math skills via computer programming. One of the successful attempts is Bootstrap [2], which is an introductory programming curriculum designed for junior and senior high school students. In Bootstrap, students create their own games using the DrRacket programming language [14], in one semester. The curriculum has proven effective in improving students' understanding of concepts from algebra. Schanzer et al. [11] report the outcome of teaching the curriculum at three different schools, by comparing the scores of pre- and post-tests. The results show significant improvements in students' scores of function composition problems and word problems.

In this study, we seek for a way to enhance students' understanding of mathematical induction via programming with *recursion*. Recursion is a common technique to represent iterative computations in computer programs. Interestingly, there is a clear correspondence between a program that uses recursion and a proof that uses induction: both have a base case and an inductive case, and in the latter case, we use some kind of assumption. This means recursion is also hard to understand, and indeed, how to teach recursion has been an active topic of research in the CS education community. However, unlike induction, which always appears in the context of mathematical proofs, recursion can be learned in a more visually appealing way: we can immediately observe how a recursive program works, and in certain programming environments, we can even see the computational steps one by one [3, 4, 6]. This would make learning of recursion easier and more fun than learning of induction.

This thesis is about our PBTS II project on teaching induction by recursion. Our specific contributions can be summarized as follows:

- We present an introductory programming course for high school students (Section 3). The course is a condensed version of the Bootstrap curriculum, where students learn to program with positions and lists in two hours.
- We design a test for evaluating students' understanding of induction (Section 4). To guide students to exhibit the ability we are interested in, we do not ask them to use induction, but to explain how it works and why it is valid. We then report the results of an experiment, where three different groups of students participated in our test.

- We propose an approach to teaching the connection between induction and recursion (Section 5). While the link is obvious to experienced programmers, our experiment suggests that many novices do not identify it themselves. This motivates us to introduce different patterns of recursive functions *in pair with* similarly structured proofs by induction, and explicitly mark the correspondence in the class room.

2 Recursion and Mathematical Induction

Recursion is a programming counterpart of induction, which serves as an alternative to looping constructs such as `for` and `while`. As a simple example of programs that use recursion, let us consider the factorial function. The reader should have seen the definition of the factorial symbol "!" in the high school text book, which is something like the following:

$$n! = n * (n-1) * ... * 3 * 2 * 1$$

This definition is correct, but a bit informal from a mathematical point of view: it uses elipsis "..." to hide an unknown number of iterative multiplications. So, how can we define "!" without using "..."?

Since factorial is defined over natural numbers, we know the argument n is a natural number, and is either 1 or greater than 1. We must make sure that we can compute factorial in both of the two cases. When n is equal to 1, the answer is 1. Then, what about the case where n is greater than 1? By taking a closer look at the definition of $n!$, we find that $n!$ is equal to $n * (n-1)!$. Therefore, we can define the factorial symbol as the following conditional function:

$$fac(n) = \begin{cases} 1 & \text{if } n = 1 \\ n * fac(n-1) & \text{otherwise} \end{cases} \quad (1)$$

Observe that the definition refers to the *fac* function, which is what we are defining right now. In computer science, we call such functions *recursive*.

Now, let us turn this mathematical definition into a computer program. In the DrRacket programming language, we define the factorial function in the following way:

```
(define (fac n) (if (= n 1)
                    1
                    (* n (fac (- n 1)))))
```

The `define` keyword means that the above program is a function definition. What follows next tells us the function being defined is called `fac`, which takes in an argument `n`. The last part is a computation that will happen when the function `fac` has been given an argument `n`. The `if` keyword has the same meaning as in English, that is, we do different things depending on whether a certain condition holds or not. In this case, the condition is `(= n 1)`, which means "n is equal to 1". If this condition holds, we return 1. Otherwise, we return the number obtained by multiplying n by the factorial of $n-1$. We find that in the second case, we are again referring to `fac`, which is the function being defined. In the programming terminology, this use of the `fac` function is called a *recursive* call.

We can easily see a clear correspondence between the mathematical function $fac(n)$ and the DrRacket function `(fac n)`: both functions have two cases $n = 1$ and $n > 1$, and in the latter case, we refer to

the function we are defining. Since the two cases cover all natural numbers, and since we can compute an answer in both cases, we can conclude that the factorial function (be it defined as a mathematical formula or a DrRacket program) is defined on all natural numbers.

The reader might wonder if it is fine to use a function we are defining in its definition. Indeed, if we unconditionally allow self-references, we can construct programs that do not terminate. To ensure that a recursive function returns an answer for any input, we must follow two rules:

1. Any recursive call is made on an argument that is *smaller* than the original one.
2. The answer for the smallest input is defined *without* using recursive calls.

If a function f meets these requirements, then we know that the argument to f becomes smaller at each recursive call. This takes us to the base case at some point. In the base case, f is passed the smallest argument, for which we can produce an answer without calling f. Thus, we obtain the guarantee that the function always terminates.

Back to the definition of the factorial function, we can see that the recursive call is made on $n-1$, which is smaller than n, and the answer for the base case does not refer to *fac*. Therefore, the factorial function is a terminating function. For the reader's enjoyment, here is the whole process of computation that happens when calling *fac* with 3:

$$\begin{aligned} fac(3) &= 3 * fac(2) \\ &= 3 * (2 * fac(1)) \\ &= 3 * (2 * 1) \\ &= 6 \end{aligned}$$

Interestingly, the definition of the factorial function has the exactly same structure as a proof by induction. Let us compare the mathematical definition of *fac* and the proof from the introduction:
As we can see, both the function and the proof consist of two cases. The base case is $n = 1$, and the inductive case is $n > 1$ in the factorial function and is $n = k + 1$ in the proof. Note that these conditions are equivalent since n and k are natural numbers. We then find that the inductive case uses an assumption, which is "the answer of $fac(n-1)$ is defined" in the factorial function, and "the proposition $P(k)$ holds" in the proof. Having defined/proved the two cases, we finish with a universally quantified conclusion, namely "$fac(n)$ is defined for all natural numbers n" and "$P(n)$ holds for all natural numbers n". Thus, we can summarize the correspondence in the following way:

	Case 1	Case 2	Assumption	Conclusion
Recursion	$n = 1$	$n > 1$	$fac(n-1)$ defined	$fac(n)$ defined for all n
Induction	$n = 1$	$n = k + 1$	$P(k)$ holds	$P(n)$ holds for all n

Moreover, the two rules for defining a terminating recursive function correspond exactly to the rules for writing a valid proof by induction. That is, we must make an induction hypothesis on a smaller number k, and we should not use any assumption in the case where $n = 1$.

2.1 Related Work

The challenges with learning induction have long been an active area of research in the math education community. Palla et al. [8] investigate students' comprehension of induction at a high school in Greece. They use a test consisting of three different levels of questions, among which the easiest one asks to write the principle of induction. They report that only 34% of students wrote a complete definition of induction; incomplete answers lacked reference to natural numbers, the base case, or the concluding sentence. Moreover, even some of those who wrote a perfect definition were not able to explain the meaning of induction to others, because they just "knew the definition by heart".

Stylianides et al. [12] investigate education- and math-major students' knowledge of induction. Similarly to Palla et al., they spotted students' difficulty with understanding the necessity of the base case (especially for those majoring in education), and further observed their trouble with identifying the domain of discourse. The latter observation was made through a slightly advanced proof by induction, which deals with a proposition of the form "For all $n \geq 5$, $P(n)$ holds." When proving such a proposition, we start by showing $P(5)$, and there is no need to check cases where $n < 5$. However, Stylianides et al. report that students tend to conclude the proposition only holds in *some* cases, with the observation that cases like $P(3)$ do not hold.

Our work aims at teaching induction by recursion. A similar approach has been taken by Polycarpou [9], who studied how exercises on recursive definitions affect students' performance on proofs by induction. Polycarpou gave her participants (CS-major undergraduates) a recursive definition of an artificial programming language, and asked them to prove its property using induction. After a series of lessons on recursive definitions and structural induction, the participants were given the same test again. The score from the post-test showed a significant increase; in particular, most of the students who correctly understood the recursive definition wrote a correct proof by induction. On the other hand, 40% of students who failed to understand the definition managed to write a correct proof, which is a strange result. Polycarpou conjectured that these students wrote the proof *mechanically*, quoting Thompson's words from [15]: "students appear to be approaching the proof process very algorithmically, having memorized a process instead of understanding its origin."

Leron and Zazkis [7] give an interesting comparison between induction and recursion from an educational perspective. They claim that an inductive definition is usually understood in an *upward* manner: we start from the base case, and go up toward a larger number via the induction hypothesis. On the other hand, a recursive function definition has a *downward* interpretation: we compute the answer of $f(n)$ using the answer of $f(n-1)$, which is obtained using the answer of $f(n-2)$... and so on. The authors also discuss which of induction and recursion is "simpler", and suggest that the latter is more accessible in that programming activities are more appealing than proving propositions, and that the process of computation can be visualized.

3 A Programming Course for Understanding Induction

We have seen that recursion is closely related to induction. For this reason, many computer science teachers mention induction when introducing recursion. Then, can we also do things the other way around? That is, would it be effective to teach recursion first, and use this experience to aid students' learning of induction? To answer this question, we designed a programming course for high school

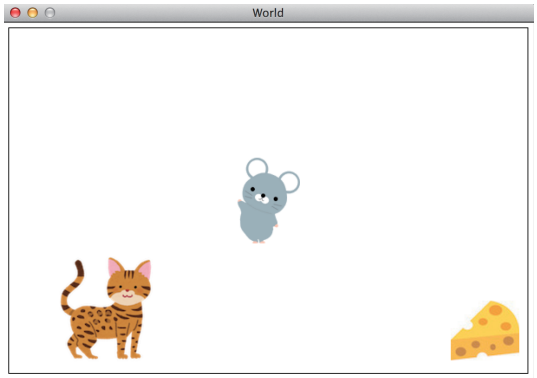


Figure 1: A Mini Game with One Mouse and One Cat

students, and gave two experimental lectures at a public high school in Japan. Our course consists of two parts, in which students program with non-recursive and recursive data. Similarly to the Bootstrap curriculum, students learn various programming concepts by building an interactive game in DrRacket. The goal of the lecture is to define a recursive function on a list, which manipulates the position of multiple cats in the game window. In this section, we give the detail of our course, and report the feedback and observations from our experiment.

Our teaching materials, including the course handout and the games, are available online at:

`https://sites.google.com/site/kawagoegame/`

3.1 Course Description

3.1.1 Part I: Programming with Positions and Simple Functions

In the first lecture, we use a mini game shown in Figure 1. The rule is simple: we move the mouse using the arrow keys and let it get the cheese on the right-bottom of the screen. When the mouse is hurt by the cat, who is moving from left to right, the game is over.

The goal of the first lecture is:

- to understand that we can move the mouse and the cat by modifying their position; and
- to understand that we can modify a position using a function.

Let's play a mini game—how does it work? We begin by playing the mini game, and then observe what moves and what does not. In the game, the mouse moves when given a key input, and the cat moves on its own (i.e., as the time passes). The only object that does not move is the cheese. To understand how the game works, we have to figure out how we can make the mouse and the cat move.

What information do we need to move the mouse and the cat? We write "to move = to change ..." on the blackboard and ask them: "What would you put in the dots?" The correct answer is *position*. For instance, if the cat has moved, it means the position of the cat has changed. Therefore, to move the mouse and the cat, we need to keep track of their positions and modify the positions when necessary.

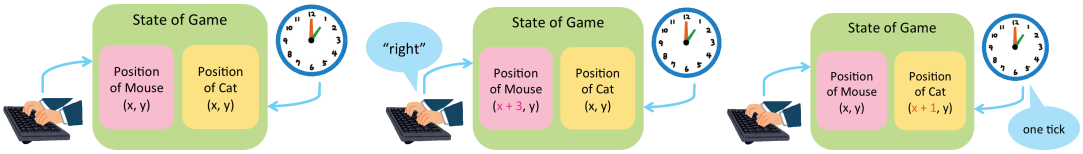


Figure 2: How the One-Cat Game Works

To give the students a better idea, we present three figures that illustrate how the game works (Figure 2). The left-most figure shows that the mouse and the cat are represented as positions in the game. The figure on the middle shows what happens when the user has pressed the right key: the position of the mouse changes from (x, y) to $(x + 3, y)$. When this has happened, we will see on the screen that the mouse has moved a bit to the right. The last figure shows what happens on every clock tick: the position of the cat changes from (x, y) to $(x + 1, y)$. Since this happens on every tick (more precisely, every 1/24 second), we see that the cat is always moving from left to right.

Positions in DrRacket Having motivated the use of positions, we learn how to represent positions in DrRacket. In the universe teachpack, which is a useful DrRacket library for making games, positions are defined as a data structure called `posn`, which has two fields `x` and `y`:

```
(define-struct posn (x y))
```

In DrRacket, each data structure `xxx` comes with a constructor function `make-xxx`. That is, to construct positions, we use the `make-posn` function. For instance, the position (1, 2) is represented as `(make-posn 1 2)`. Note that the parentheses and the spaces in this code cannot be omitted. We then ask the students how to represent the position (5, 0).

After learning how to represent positions in DrRacket, we remind the students of the reason why we are dealing with positions: we move the mouse and the cat by modifying their positions. The next question would be: what kind of operations do we need to modify a position? In the case of the cat, we want to change its position (x, y) to a new position $(x + 1, y)$ on every clock tick. We observe that in the new position, we are referring to the x- and y-coordinates of the original position. This suggests that we need a means to access the value in each field of the `posn` data structure. To extract such a value, we use the selectors `posn-x` and `posn-y`, which are functions that DrRacket automatically generates for the `posn` data structure. As an example, if `p` is `(make-posn 1 2)`, `(posn-x p)` is `1` and `(posn-y p)` is `2`. Now we ask the students what values we obtain by applying `posn-x` and `posn-y` to `(make-posn 0 5)`.

Changing positions using functions Now we have operations to modify positions, but who will be playing the role of applying those operations in our code? Recall that to move the cat, we have to increase the value of the x-coordinate on every tick. What we need, then, is a "magic box" that turns (x, y) into $(x + 1, y)$.

$$(x, y) \longrightarrow \boxed{?} \longrightarrow (x + 1, y)$$

What would the box be? A *function*! A function is something that takes in an input and returns an output. For high school students, functions might be a synonym of "graphs", but graphs can be understood

as displaying the input-output relation of functions, so boxes and graphs are just different representations the same thing.

So, what we need is a function that receives the current position of the cat, and returns a new position, where the x-coordinate has been increased by one. We ask the students to write part of this function. As a transition to the exercise, we open a new file with a *wrong* function for moving the cat. When they run the game, the students will see that the cat is not moving, while all other actions remain the same as before. We close the game window and find the following function, which is called on every tick to move the cat:

```
(define (move-cat cat)
  (make-posn (posn-x cat) (posn-y cat)))
```

We explain the constituents of this definition—the name, the argument, and the body—by linking each part to the corresponding part of a simple mathematical function, say $f(x) = x + 1$. Then we look at its meaning: the function takes in a position `cat` and returns a position, because the body is of the form `(make-posn x y)`. What are the x- and y-coordinates of the new position? They are the x- and y-coordinates of the input `cat`, which represents the cat's current position. What does it mean? We are decomposing a given position into x and y and then rebuilding a position (x, y). So this is in fact a function that does not change anything: whatever position we pass to `move-cat`, we obtain the same position as the output. This is why the cat is not moving in the game. What we want this function to give us back is an updated position, where the x-coordinate has been increased by one. This can be done by replacing the code `(posn-x cat)` with `(+ (posn-x cat) 1)`.

Try moving the cat! It's finally time for coding. We ask the student to make the above change, and run the game again. After everyone has got her code running, we wrap up what we have learned so far: (i) the mouse and the cat are represented as positions; and (ii) positions can be modified using functions.

Previewing Part II: How to deal with multiple cats We use the rest of the lecture to show the students what we are going to learn in the second lecture. We present a new game, which has three cats instead of just one, as shown in Figure 3 left. These cats are represented as a *list* of positions. We ask the students to name a few lists in their everyday life, such as shopping lists and play lists. Then we observe their structure and give a definition to lists: lists are a data structure containing elements in some order.

When we have a list, we usually want to do the same thing to every element in the list. For instance, if we have a shopping list, what we want to do is to buy all the elements. Similarly, if we have a play list, we want to play every song. Time is up; we finish the lecture here.

3.1.2 Part II: Programming with Lists and Recursive Functions

In the second lecture, the students will write a function for moving three cats. Our goal is:

- to understand what lists are; and
- to understand how a recursive function works

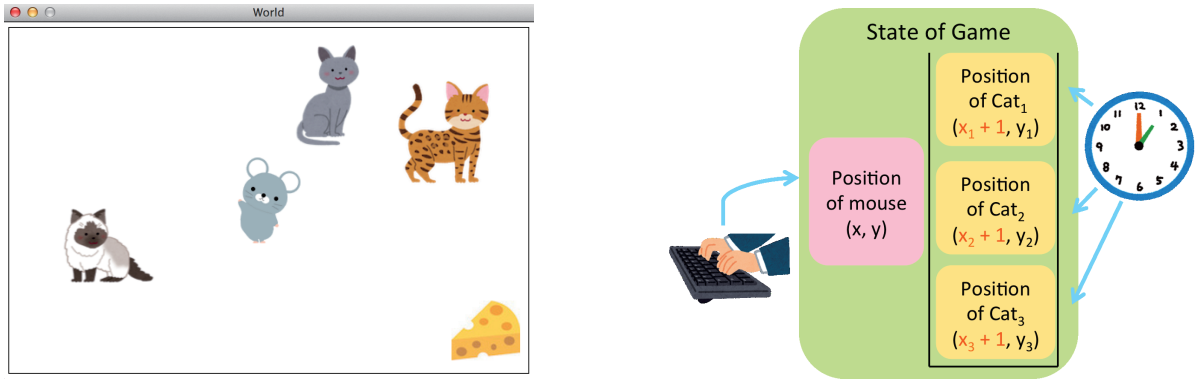


Figure 3: A Three-Cats Game and How It Works

Reviewing Part I We begin with a quick review of the first lecture. Below are a list of important remarks:

- "To move" can be rephrased as "to change position".
- The mouse and the cat are represented as positions.
- Positions can be modified using functions.
- A function receives an input and returns an output.
- We defined a function for moving the cat, which receives the cat's current position and returns a new position where the x-coordinate is increased by one.

After reviewing these, we show the new game to remind the students that we will be dealing with three cats, and that these cats are represented as a list of positions, with a brief review of what lists are. Now the state of the game looks like Figure 3 right: on every clock tick, we update all the three positions. It would be a good idea to roughly describe the motivation of using a list at this point: when representing the cats as a list of positions, we will be defining operations on the cats as functions over a list of an arbitrary length. Thus, when we add more cats or remove existing ones, we do not need to make a lot of changes to our code.

Lists in DrRacket We show several examples of lists containing natural numbers (Figure 4), and explain how to represent those lists in DrRacket. Since a list is a sequence of elements, it is either an empty sequence or a sequence with at least one element. For any non-empty list, we can construct the list by adding elements one by one to an empty list. Therefore, we need two things to construct lists: the empty list, and an operation for adding an element to a list. In DrRacket, we represent the empty list as `empty`, and when we add the element `a` on the top of the list `l`, we write `(cons a l)`. We present the DrRacket-encoding of `list1` from our examples, which looks like `(cons 1 (cons 2 (cons 3 empty)))`, and ask the students how to represent other examples.

Now we have learned how to construct lists. The next question is: what kind of operations do we need when dealing with lists? To answer this question, we use a shopping list containing "carrot", "tomato", and "cabbage". On the slide, we show a picture of a supermarket, where carrots, tomatoes, and cabbages

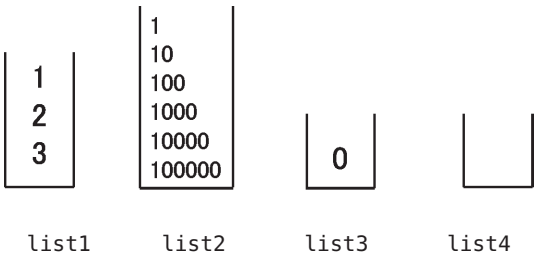


Figure 4: Examples of Lists

	first	rest
list2	1	(10 100 1000 10000 100000)
list3	0	empty
list4	error	error

Figure 5: Dealing with Lists

can be found in this order. Assuming that we buy the vegetables in the list from top to bottom, our first task will be adding a carrot to the cart. Put in a more algorithmic way, we apply the operation "add to cart" to the *first* element of the list. Now we have done what we should do with carrots, so we can remove "carrot" from our shopping list. What remains is to add a tomato and a cabbage in our cart. In other words, we have to apply the operation "add to cart" to every element in the *rest* of the list. Thus, when dealing with lists, we need operations for extracting the first element and the rest of the lists. In DrRacket, these are the keywords `first` and `rest`. For instance, `(first list1)` will give us back the first element of `list1`, namely `1`, and `(rest list1)` gives us the rest of the list, that is, `(cons 2 (cons 3 empty))`. We then ask the students what we obtain when applying `first` and `rest` to `list2`, `list3`, and `list4`. The answers are shown in Figure 5. Note that applying `first` and `rest` to `list4` results in an error because these operations are defined only for non-empty lists.

Dealing with three cats We go back to the game and show the students again the its internal state (Figure 3). We see three arrows from the clock to each cat, meaning that every cat's position gets modified on every tick. Here we recall that in the first lecture, we used a function to move the cat: we defined a function that receives a *single* position and returns a new position. In this new game, we move the three cats also using a function, which will receive *a list of* cats' positions and returns a new list of positions. So what we want is the box in the following figure:

$$\boxed{\begin{array}{c}(x_1,\, y_1)\\(x_2,\, y_2)\\(x_3,\, y_3)\end{array}} \longrightarrow \boxed{?} \longrightarrow \boxed{\begin{array}{c}(x_1+1,\, y_1)\\(x_2+1,\, y_2)\\(x_3+1,\, y_3)\end{array}}$$

An important remark is that when defining a function on lists, we want the function to work for arbitrarily long lists. The figure looks like the function works on a list whose length is exactly 3, but what we are going to define is something more general.

Try moving the cats! Now we open a new file, which contains a *wrong* function for moving the cats. By running the game, we notice that the cats are not moving. Let us take a look at the definition of the function for moving the cats:

```
(define (move-cats cats)
  (if (empty?  cats)
      empty
      (cons (first cats) (rest cats))))
```

The function receives a list of cats' positions, `cats`, and checks if the list is empty or not. If it is empty, that means we have no position to update, so we simply return an empty list. Otherwise, the function constructs a list using `cons`. We ask the students to identify what element we are adding to what list. Thus we find that this is again a function that does nothing: whatever list we pass to it, it returns the same list. This is not what we want; we want a function that updates all the positions in a given list. How can we do that?

Before modifying the code, we try updating the position list by hand. We give the students two minutes to work in pair with the above figure: turn the list on the left-hand side of the box into the one on the right-hand side, using the following operations:

- Add a position to a list (`cons`)
- Extract the first element of a list (`first`)
- Extract the rest of a list (`rest`)
- Update one position (`move-cat`)

If the students are stuck, or have no idea what to do, we give two hints. First, when we worked with the shopping list, what we did for each element in the list was to add to cart. To update the position list, what we want to do for each position is to increase its x-coordinate, which can be done using the `move-cat` function from the first lecture. Second, in the shopping list example, what we did first was to add a carrot, the first element in the list, to the cart. In this exercise, we should start by updating the first position in the list.

After the discussion among students, we show how to update a three-cat list. As we did with the shopping list, we first extract the top-most position, (x_1, y_1), using `first`. This is a single position, so we can update it using `move-cat`. Looking at the original list, we still have two positions to update. So we do the same thing to the second position: we extract (x_2, y_2) using `first`, and update it using `move-cat`. Then we update (x_3, y_3) in the same way. Now the list is empty. An empty list has no position in it, so we do not need to update anything. Then, what is remaining is to produce the updated list, by adding the new positions, $(x_3 + 1, y_3)$, $(x_2 + 1, y_2)$, and $(x_1 + 1, y_1)$, to the empty list in this order. If we write down what we have done, we will obtain the following procedure list:

1. Extract (x_1, y_1) and update it
2. Extract (x_2, y_2) and update it
3. Extract (x_3, y_3) and update it

4. Cons $(x_3 + 1, y_3)$ to an empty list
5. Cons $(x_2 + 1, y_2)$ to the result of 4
6. Cons $(x_1 + 1, y_1)$ to the result of 5

This works well, and if we turn it into a DrRacket program, we will be able to move the three cats. But what if we add one more cat? It is easy to imagine what happens when we update a four-cat list following the above procedure: we will have one more position to update when we finished the third one, and we need one additional `cons` for this position. Now, what if we have 100 cats? The list should consists of 100 lines of "Extract xxx and update it" and 100 lines of "Cons xxx to ...". Do we want to write all these 200 lines of code? Of course we don't. Actually, the first 100 lines are basically doing the same thing, so do the next 100 lines. There should be a cleverer way to do this.

We recall that a list is either an empty list or a list containing one or more elements. Since we want to define a function that works for any list, we have to make sure that the function gives us a correct answer in both cases. We already know what to return when given an empty list. When given a non-empty list, there are essentially two tasks. First, we update the top-most position. We can extract the top-most element of `cats` by `(first cats)`, and update it by `(move-cat (first cats))`. Then, we add the updated position on the top of the result of updating the rest of the list. We can extract the rest of `cats` by `(rest cats)`, but how do we update it? Here we ask the students a question: what does the `move-cats` function do for us? It updates every position in a given list. Then, if we pass `(rest cats)` to `move-cats`, what will we obtain? It would be easier to find the answer with an example. If `cats` was a list containing (x_1, y_1), (x_2, y_2), and (x_3, y_3), `(rest cats)` is a list containing the latter two positions, and `(move-cats (rest cats))` will be a list containing $(x_2 + 1, y_2)$ and $(x_3 + 1, y_3)$. This is exactly the list we want to add $(x_1 + 1, y_1)$ to. So, if we replace `(first cats)` and `(rest cats)` in the wrong definition with `(move-cat (first cats))` and `(move-cats (rest cats))` respectively, we will be able to move the cats. Now, time for coding!

How the function works After everyone has successfully modified the code, we explain how the function updates the list step by step, using the figures in Figure 6. We start the computation by calling `move-cats` with a three-cat list. Since this list is not empty, we update the top-most position and make a recursive call with a two-cat list. In this recursive call, the argument is again non-empty, thus we update the new top-most position, which was originally the second one, and make another recursive call with a one-cat list. We update the last position in a similar manner, and then we call the function with an empty list.

Instead of keep going to obtain the final result, we show these steps again, asking the students to observe the argument to `move-cats` at each step. The first call is made with a three-cat list, then a two-cat list, then a one-cat list, and lastly an empty list. What we can conclude is that at each recursive call, we pass a *shorter* list. This is one of the important observations to understand how the `move-cats` works.

We make another key observation at the next step. When the argument is empty, we evaluate the first branch of the conditional expression, which gives us back an empty list *without* calling `move-cats`. This empty list is the "base" on which we construct the final answer.

To summarize, defining a function using itself, which may sound surprising to beginners, is valid as long as (i) every recursive call is made on a smaller argument; and (ii) we can compute the answer of the base case without recursive calls.

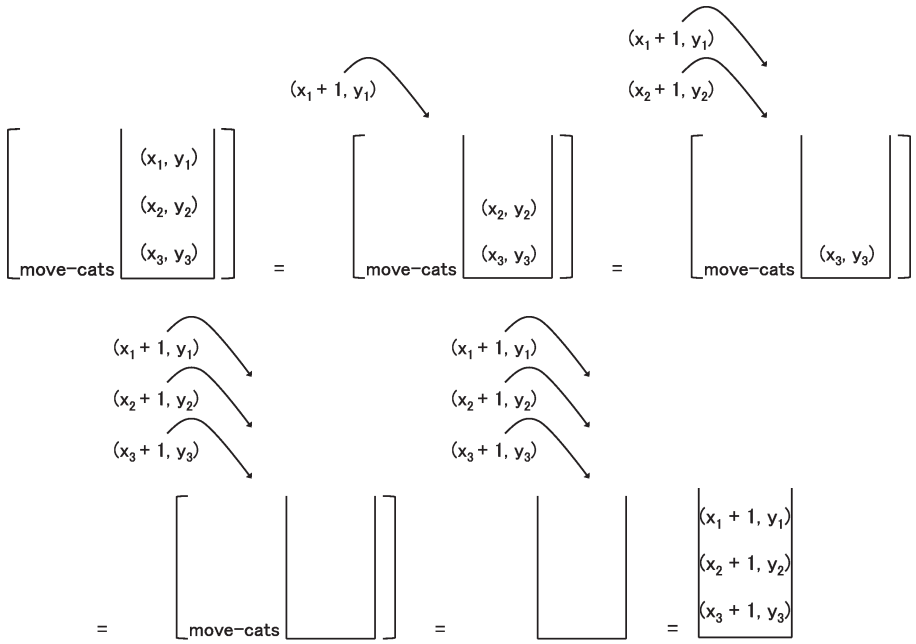


Figure 6: Updating Cats List

	Recursive Function	Mathematical Induction
What we want to show	we can compute `(f l)`	$P(n)$ holds
Base case	`l` = `empty`	$n = 1$
Inductive case	`l` = `(cons a l')`	$n = k + 1$
Hypothesis	we can compute `(f l')`	$P(k)$ holds

Figure 7: Recursive Function and Mathematical Induction

We finish the class by wrapping up what we have learned: (i) a list is a sequence of elements; and (ii) a recursive function computes an answer by using the function itself, which is safe when it is built in a way that we always reach a base case involving no self-reference.

Remark As we explained ealier, the derived rules correspond exactly to the rules for writing a valid proof by induction: (i) every induction hypothesis is made on a smaller number; and (ii) we can prove the base case without any induction hypothesis. What this means is that we have been using mathematical induction when defining the `move-cats` function. Indeed, each part of a recursive function on a list has a corresponding counterpart in a mathematical proof by induction, as shown in Figure 7.

3.2 Experiment

3.2.1 Participants

In 2017, we gave two lectures at Saitama Prefectural Kawagoe Girls' Senior High School[1]. The participants were 42 second-grade students of the "SSH" class ("SSH" is short for Super Science High school). According to our background survey (Figure 8), most students are interested in studying science in the

[1]This study was approved by the ethics committee of Ochanomizu University.

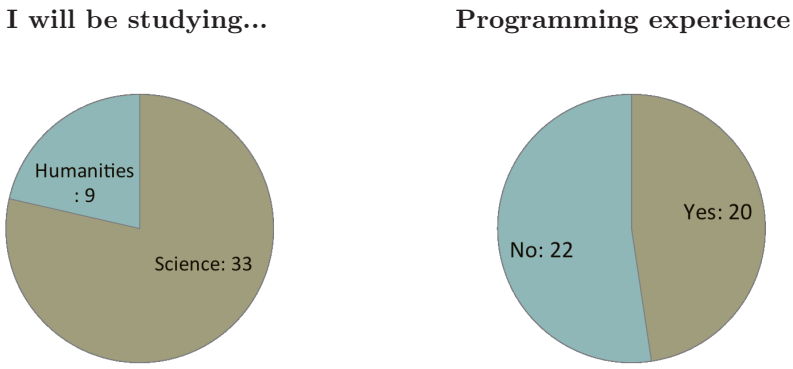


Figure 8: Participants Data

university. Half of the students had some programming experience, such as controlling a car, making a short movie, and creating a game. A few students remember the name of the programming language they used: three students used C, and one student used "Programin" [13], a visual programming language developed by the Ministry of Education.

Note that when we gave the lectures, the students have *not* learned mathematical induction in math class. Therefore we did not explicitly mention induction in our lectures, but we provided a handout explaining the relationship between induction and recursion after they learned induction.

3.2.2 Course Implementation

We taught the two parts of our course on different days: we taught Part I on August 31, 2017, and Part II on September 14. Each lecture was 65-minute long. The last five minutes were used for questionnaire survey.

The lectures were given at a computer room. Every table was equipped with two computers, and a monitor for displaying slides. The classroom also had a whiteboard, which we used when explaining the coordinate system of the game. Due to security reasons, the students had to install DrRacket at the beginning of each lecture, but it did not cause a lot of troubles because installation was very easy.

3.2.3 Use of Japanese Primitives

To help the students feel more comfortable with coding, we provided "Japanese" primitives. For instance, in our game, the empty list is represented as `kara`, which means "empty" in Japanese. The complete list of Japanese primitives used in our course is presented in Figure 9.

3.2.4 Course Evaluation

To see how well the students understood the class, we asked the students to fill out a questionnaire survey at the end of each lecture. We show the complete set of questions in Figure 10. Note that the first three questions were asked only in the first lecture. Note also that the survey was anonymous. We asked the students to write their student number, but the purpose was to analyze their understanding and performance using their background information, rather than to identify individuals. We also allowed the students to write nothing if they did not want to answer the questions.

DrRacket expression	Japanese expression
`make-posn`	`make-zahyo`
`posn-x`	`zahyo-x`
`posn-y`	`zahyo-y`
`empty`	`kara`
`cons`	`ireru`
`first`	`sento`
`rest`	`nokori`

Figure 9: Japanese Primitives

1. Do you have former experience in programming? If you have, what did you do using what programming language?

2. Do you like mathematics?
   1. Yes 2. No

3. What do you want to study in university?
   1. Science 2. Humanities 3. Haven't decided yet

4. Was today's lecture easy or hard? If you have chosen 3 or 4, what was hard to you?
   1. Easy 2. Not hard 3. A little hard 4. Hard

5. Was the lecturer's explanation clear? If you have chosen 3 or 4, what was unclear to you?
   1. Clear 2. Mostly clear 3. Not very clear 4. Unclear

6. Please write your suggestions on our lecture if you have any.

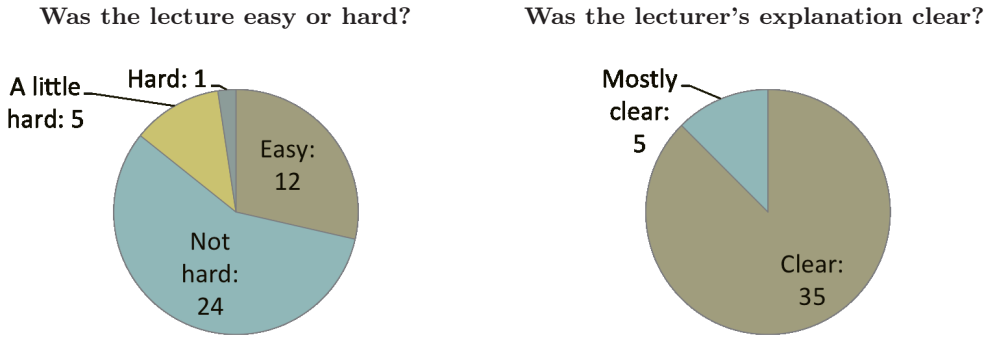
7. Please write your comments (what you found interesting, what you want to learn more, etc.).

Figure 10: Questionnaire Survey

Figure 11 shows the results of the questionnaire survey from the first lecture. As we can see from the graphs, 86% of students found the first lecture easy to follow, and all students answered that our explanation was clear or mostly clear. Note that the number of responses of the second question is 40 because the question was printed on the second page and two students did not notice that. Some of the students' comments are presented in the table below the graphs. The numbers following the text show how many students wrote similar comments.

Figure 12 shows the results of the second lecture. One student was absent on that day, therefore the number of responses is 41. As the first graph shows, 59% of students found the second lecture hard. Nevertheless, most of them still answered that our explanation was clear. It is possible that some students refrained from writing negative comments. Indeed, one student wrote that our explanation was very gentle, but that she could not understand what was happening.

Although the lecture was hard for the students, many of them enjoyed making the game. It is a great



What was hard?	Typing code (2) Concept of function (2)
Suggestions for improvements	I wanted to write more code. (5) The lecture was too slow. (2)
Free Comments	I learned how mathematics is used in games. (6) Writing correct programs was hard. (3) I enjoyed changing the speed of the cat. (2) I liked the mouse and the cat; they are so cute! (2) I was happy when the cat moved. I would like to move the mouse as well.

Figure 11: Results of Questionnaire Survey (First Lecture)

advantage of game programming that the students can immediately see if their code is correct or not by running the game. During the very limited coding time, several advanced students successfully changed the speed of the cat, and taught others how to do that. Such interaction among students is welcome in the classroom, as it increases students' motivation. Our design of the games also seems to have attracted the students a lot.

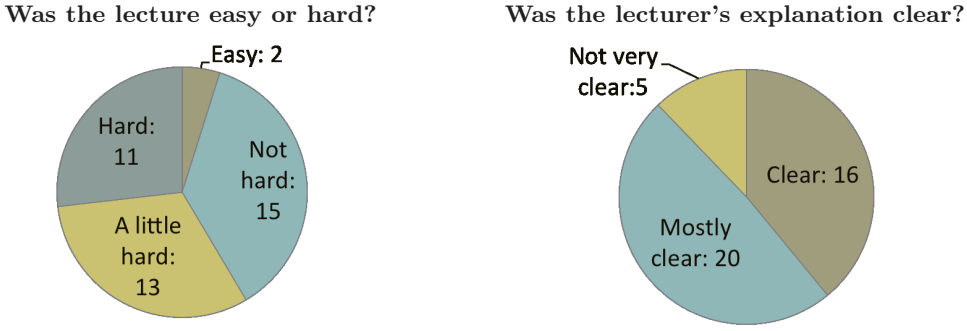
4 Testing Understanding of Induction

Induction is hard, but what exactly do students fail to understand? To answer this question, we designed a test for evaluating one's conceptual understanding of induction. Unlike ordinary math exams, we do *not* ask students to write a proof by induction. Instead, we guide them to observe how inductive reasoning goes, and ask them to describe the process in their own words. In this section, we present the test design, and report the results from our experiments.

4.1 Test Design

Typically, understanding of induction is tested by asking students to write proofs by induction. In the simplest cases, proving by induction is fairly easy. All we have to do is to follow the recipe in our textbook[2]: we first show $P(1)$ holds (the base case), then show $P(k+1)$ holds by assuming $P(k)$ holds (the inductive case), and conclude $P(n)$ holds for an arbitrary natural number n. The base case is usually trivial. The inductive case requires some computation, but once we have figured out the right place to invoke the induction hypothesis, the goal follows in a straightforward manner. After writing a concluding

[2]Note that, in high school textbooks used in Japan, natural numbers are defined as integers greater or equal to 1.



What was hard?	Recursive function Concept of lists Unfamiliar words Everything
Suggestions for improvements	I wanted to code by myself. The repeated explanation was confusing. Shorter explanation would be easier to understand.
Free Comments	I enjoyed making the game. (7) Typing alphabets was hard. (6) The lecture was harder, but the game was more fun. I learned that to ask a computer to do something, we have to give every single instruction. I want to learn more about recursion. I want to try programming at home. I want to build the game from scratch. Please give us another lecture!

Figure 12: Results of Questionnaire Survey (Second Lecture)

statement, we obtain a correct proof. These can be done without understanding what role the base case plays, why it is valid to use the induction hypothesis, or why we can derive a conclusion covering all natural numbers. What this means is that standard exercises on induction do not help us examine students' conceptual understanding.

We propose a different method. Instead of asking students to *use* induction, we ask them to *observe* what is happening in a proof by induction and *explain* how induction works in their own words. Below, we show the complete test we designed in this study [5].

> A student (S) and a math teacher (T) are talking about a proof by induction. Read the conversation and answer the questions.

Prove that the following equality holds for all natural numbers n.

$$1 + 3 + 5 + ... + (2n - 1) = n^2 \quad ... (*)$$

Proof: We prove the statement by induction on n.

(1) Suppose $n = 1$. The left-hand side of $(*)$ is 1, which is equal to 1^2.
Therefore $(*)$ holds.

(2) Suppose $n = k + 1$ and $(*)$ holds when $n = k$.
The left-hand side computes as follows:

$$\begin{aligned} & 1 + 3 + 5 + ... + (2k - 1) + (2(k + 1) - 1) \\ &= k^2 + 2k + 1 \\ &= (k + 1)^2 \end{aligned}$$

Therefore $(*)$ holds.

By (1) and (2), we conclude that $(*)$ holds for all natural numbers n. □

S: The proof concludes that $(*)$ holds for all natural numbers in just two steps. Why can we make such a conclusion without checking individual numbers one by one?

T: Good question. A natural number n is either 1 or greater than 1, and in the latter case, we must be able to represent n as $k + 1$ for some natural number k. Now observe the proof; the steps (1) and (2) correspond exactly to these two cases.

S: But in the second step, we use the assumption that $(*)$ holds when $n = k$. Why is this valid? What if this assumption is false?

T: Ah, OK. To see the reason, it would be a good idea to try proving the assumption, namely $(*)$ holds when $n = k$. Let's prove this again by induction. What is the first step?

S: We prove the case where $k = 1$, which is trivial.

T: Right. In the next step, we prove the case where $k = k' + 1$, assuming $(*)$ holds when $k = k'$. By the way, how can we represent k' using k?

S: Since $k = k' + 1$, $k' =$ $\underline{(a)}$.

T: Yes. Recall that the statement we are proving right now, namely "$(*)$ holds for all natural numbers k", is what we need to prove the original statement "$(*)$ holds for all natural numbers n". So we should be able to represent k' using n as well.

S: $k' =$ $\underline{(b)}$.

T: Perfect. Now we go back to our proof. We are trying to prove the case where $k = k' + 1$, assuming $(*)$ holds when $k = k'$. Here, we are making a new assumption, so we might want to prove this as well. That is, we prove $(*)$ holds for all natural numbers k'.

S: We first prove the case where $k' = 1$, then prove the case where $k' =$ $\underline{\quad (c) \quad}$ $+ 1$ by assuming $(*)$ holds when $k' =$ $\underline{\quad (c) \quad}$.

T: Yes. Now, how can we represent $\underline{\quad (c) \quad}$ using k'?

S: $\underline{\quad (c) \quad} = \underline{\quad (d) \quad}$.

T: Can you also represent $\underline{\quad (c) \quad}$ using k?

S: $\underline{\quad (c) \quad} = \underline{\quad (e) \quad}$.

T: And using n?

S: $\underline{\quad (c) \quad} = \underline{\quad (f) \quad}$.

T: Did you notice something?

S: Every time we make a new assumption, we use a $\underline{\quad (g) \quad}$ number!

T: That's right! So, if we repeat this, ...

Question 1. Fill in the blanks (a) to (g).

(Correct answers: (a) $k - 1$, (b) $n - 2$, (c) k'', (d) $k' - 1$, (e) $k - 2$, (f) $n - 3$, (g) smaller)

Question 2. Complete the teacher's last response using the following keywords:
$n = 1$, $n = 2$, $n = 3$, for all natural numbers n.
(Expected answer: we will eventually reach the case where $n = 1$. Since we have already shown that $(*)$ holds when $n = 1$, we know that $(*)$ holds when $n = 2$, which tells us it holds when $n = 3$. Thus we can conclude that $(*)$ holds for all natural numbers n.")

4.2 Experiment

4.2.1 Participants

We gave our test to three groups of students described below[3].

- High1: 37 11th grade students of the SSH class at Kawagoe Girls' High School. All of them participated in the programming course we presented in the previous section. Note that SSH students are *not* selected according to their academic skills, but they usually have greater desire to learn things beyond textbooks[4].
- High2: 43 11th grade students of a non-SSH class at the same high school. Similarly to the SSH class, most students in this class have a keen interest in science. They also share the same math

[3]This study was approved by the ethics committee of Ochanomizu University.
[4]Personal communication with a science teacher at the school.

Question 1

	7	6	5	4	3	2	1	0
High1	22	6	2	0	1	1	0	3
High2	16	9	4	0	2	2	7	3
CS	33	3	4	1	1	0	1	0

Question 2

	3	2.5	2	1.5	1	0.5	0
High1	2	3	1	1	1	0	29
High2	1	3	1	0	0	1	37
CS	2	4	0	1	6	3	27

Figure 13: Students' Scores

teacher with the SSH students, which means there is no difference in the way the two groups were introduced to mathematical induction.

- CS: 43 second-year undergraduate students at the computer science department of Ochanomizu University. All the students had passed a CS 1 course in the C programming language, where they learned recursion via sorting algorithms.

4.2.2 Result of Question 1

Figure 13 left shows students' scores of the first question. We had seven blanks in total, and most students correctly filled in six or seven blanks. The average scores of the three groups were 5.86, 4.72, and 6.44, respectively. We found that several students in the second group made careless mistakes like $k = k' + 1 \to k' = 1 - k$, resulting in failure to answer subsequent questions.

4.2.3 Result of Question 2

Figure 13 right shows students' scores of the second question. We set the following evaluation criteria:

1. The student clearly states that if we always use a smaller number for a new induction hypothesis, we will eventually reach the case where $n = 1$, which we known holds.
2. The student identifies the domino-like implications, that is, if $(*)$ holds when $n = 1$, then it holds when $n = 2$, which implies it holds when $n = 3$... and so on.
3. The student concludes that the proposition holds for all natural numbers from the above two facts.

For the first question, we gave 0.5 points if the answer mentions one of "reach the base case" or "the base case holds" but not both.

Let us first show two examples of good answers. To make it clear which criteria are satisfied, we write scores as addition of three numbers $a + b + c$, where a, b, c correspond to criteria (1), (2), and (3), respectively.

> If we repeat this, n will eventually become 1. Since we have shown $(*)$ holds when $n = 1$, we know it holds when $n = 2$, which implies it holds when $n = 3$. Thus we can conclude that $(*)$ holds for all natural numbers n. (High1, $1 + 1 + 1$)

This is one of the best answers we had in our experiment. It addresses the base case, explains the chain of implications, and makes a correct conclusion. An almost-perfect answer from the CS group further summarizes the mechanism using dominos:

> ... The reason why it suffices to show the two cases can be explained in terms of dominos: if we know that falling of one domino results in falling of the next domino, then we can let all dominos fall by letting the first one fall. (CS, $0.5 + 1 + 1$)

We next give several examples of incomplete answers.

> If $(*)$ holds when $n = 1$, then it holds when $n = 2$. If $(*)$ holds when $n = 2$, then it holds when $n = 3$. By repeating this, we can show that $(*)$ holds for all natural numbers n. (High2, $0 + 1 + 1$)

The student correctly describes the implications, but she does not mention that we will reach the base case, which we have already proven. This implies that the student may not see the necessity of this case.

> Since the equation holds when $n = k$ and $n = k + 1$, we can conclude that $(*)$ always holds. (High1, 0)

This answer again lacks the base case, but further suggests the student's misunderstanding of the relationship between $P(k)$ and $P(k + 1)$: we use the former to prove the latter, instead of separately proving the two propositions.

> Since the proposition holds when $n = 1$, 2, and 3, we know that it holds for all natural numbers n. (High2, 0)

This is a completely non-logical answer. Making a general conclusion from a limited number of individual observations means the conclusion might be wrong; such reasoning is not valid in mathematics. We had two similar answers from the CS group as well.

> The value of k'' $(= n - 3)$ is -2 when $n = 1$, -1 when $n = 2$, and 0 when $n = 3$. Thus we can represent all natural numbers by substituting a larger number for n. (High2, 0)

We had three answers (from High1 and High2) similar to the above one. The student is trying to "represent" an arbitrary natural number using n, but the above numbers are *not* natural numbers. This suggests that the student either lacks the understanding of what natural numbers are, or she ignores how natural numbers are defined when proving things by induction.

4.2.4 Follow-up Questions

For those students who had no idea what to write, we asked them what they found difficult in the test. Here are some representative answers:

- I understand the conversation, but I have no idea how it is related to the second question.
- I don't see the reason why the student's observation (using a smaller number for the induction hypothesis) helps us explain how induction works.
- I don't know how to use the keywords.
- I got confused by the variables.

There seems to be a huge gap between Questions 1 and 2. Indeed, in all the three groups, half of the students who got seven points in Question 1 did not write anything in Question 2. We included the smaller-number observation in the conversation as a hint for answering Question 2, but it looks like the hint was not clear to students without a solid understanding of induction.

4.2.5 Identified Challenges

The students' answers tell us that, when teaching mathematical induction, we must clearly explain:

- why we need a base case;
- what role an induction hypothesis plays; and
- how mathematical induction differs from inductive reasoning in everyday life.

Since the standard textbook tends to provide the simplest pattern—first $n = 1$, then $n = k + 1$—as a "formal" definition of induction, students may put the case $n = 1$ without knowing why they need this case or doubting whether it really serves as a base case for the current proof. We have to let students begin by identifying the structure of the data we are dealing with, and then find an appropriate base case. We should also note that the lack of base case may take us outside the relevant domain, causing non-terminating reasoning. For instance, when proving $P(n)$ without showing $P(1)$, we would assume $P(0)$, $P(-1)$, ... ad infinitum. These assumptions are not valid since P is defined only on natural numbers.

Having proved the base case, the next step is to figure out how to prove cases with a larger number. Since we already know the base case holds, we wish to use this fact when proving the inductive case. Here, what we do is to show the existence of a "bridge" from $P(k)$ to $P(k + 1)$. Since k is smaller than $k + 1$, iteratively putting new assumptions takes us all the way down to the base case, which we know holds. Thus we can show $P(n)$ holds for an arbitrary n, using the fact we established in the base case, and the bridges we built in the inductive case.

Lastly, we have to make sure that the students are able to distinguish between proving by induction and checking individual cases. Although called "induction", mathematical induction is *deductive*; it is different from what we call induction in a non-mathematical context, which lets us derive a general but potentially wrong conclusion from non-exhaustive observations.

4.2.6 Upward vs. Downward Reasoning

By analyzing the answers from Leron and Zazki's point of view, we noticed that we had significantly more "downward" answers from the CS group. Specifically, 13 CS students wrote things like "the number we use for the induction hypothesis becomes smaller and smaller, and eventually it becomes 1". In contrast, we only had four such answers from the high school groups. This result seems to suggest that experience in recursion affects one's view of inductive process. As our test asks students to go downward, it might have been easier for CS students to write an expected answer.

5 Teaching The Connection between Induction and Recursion

As we saw in the previous chapter, even CS-major students tend to lack solid understanding of mathematical induction. Then, how well do they see the correspondence between induction and recursion, and would it help them understand how induction works? Through a series of experiments, we found that identifying the correspondence is a rather difficult task. In this chapter, we report the identified challenges, and propose a way to teaching the correspondence.

5.1 How Well Do Students Identify The Correspondence?

In the spring of 2018, we investigated how well CS students identify the correspondence between induction and recursion. The participants of this study consisted of the CS group students from the experiment described in Section 4.2, and were given the following problem:

Observe the typical structure of a proof by induction:

1. When $n = 1$, $P(n)$ holds.
2. When $n = k + 1$, $P(n)$ holds if $P(k)$ holds.

Therefore, $P(n)$ holds for all natural numbers n.

Now, observe the following OCaml functions:

```
(* times_list : int list -> int *)
let rec times_list l = match l with
  [] -> 1
| first :: rest -> first * times_list rest

(* times_tree : int tree_t -> int *)
let rec times_tree t = match t with
  Empty -> 1
| Node (t1, n, t2) -> times_tree t1 * n * times_tree t2
```

How are the proof and functions related to each other? Fill in the blanks in the following table:

	Case 1	Case 2	Assumption	Conclusion
Proof	$n = 1$	$n = k + 1$	$P(k)$ holds	$P(n)$ holds for all n
`times_list`	`l =`	`l =`		
`times_tree`	`t =`	`t =`		

Figure 14 shows the expected answers, and the number of correct answers we obtained. As we can see, most students understand which case corresponds to the base case, but less students identify the inductive case. Among those who failed to identify the inductive case, four of them wrote the definition (such as `first * times_list rest`). Finding the induction hypothesis was even more challenging, and we found that several students did not understand the intension of this question. For instance, one student answered "`times_list rest` returns an integer". This is something we need to safely perform the multiplication `first * times_list rest`, but it is not what we need to make the recursive computation work. Another student answered "the argument of the recursive call is smaller than the original one", which is necessary for ensuring termination but is again not an induction hypothesis. A bit surprisingly, the hardest problem was the last one. Here we show two kinds of commonly found mistakes:

- "`times_list l` holds for all lists"
- We can compute the product of a given integer list.

The first one is inappropriate because `times_list l` is not a proposition but a computation. The second one lacks a universal quantifier for the argument list, which is important in a proof by induction.

From these observations, we conclude that we should not expect students to identify the correspondence by themselves. Instead, we must explain it clearly in the classroom.

	Case 1	Case 2	Assumption	Conclusion
Proof	$n = 1$	$n = k + 1$	$P(k)$ holds	$P(n)$ holds for all n
`times_list`	`l = []`	`l = first :: rest`	`times_list rest` defined	`times_list l` defined for all lists `l`
Correct Ans	30	21	12	4
`times_tree`	`t = Empty`	`t = Node (t1, n, t2)`	`times_tree ti` defined	`times_tree t` defined for all trees `t`
Correct Ans	31	19	11	5

Figure 14: Expected Answers and Results (Number of Participants: 36)

5.2 Teaching The Correspondence

In this section, we describe our approach to teaching the link between induction and recursion. Suppose we want to define the factorial function in the C programming language, which is used in the freshman courses at Ochanomizu University. We first recall the familiar definition from high school math:

$$n! = n * (n-1) * ... * 1$$

We point out the problem with the "...", and show a more formal mathematical definition:

$$fac(n) = \begin{cases} 1 & \text{if } n = 1 \\ n * fac(n-1) & \text{otherwise} \end{cases} \tag{2}$$

and the corresponding C program:

```
int fac(n : int) {
    if (n == 1) {
       return 1;
    }
    else {
    return (n * fac(n - 1))
    }
}
```

We then ask the students if they find something "weird" in this definition. After someone has mentioned the self-reference, we show how the computation goes:

$$\begin{aligned} fac(3) &= 3 * fac(2) \\ &= 3 * (2 * fac(1)) \\ &= 3 * (2 * 1) \\ &= 6 \end{aligned}$$

So, everything looks well. We note that the factorial function indeed works for any natural number, because it satisfies the two rules we discussed in Chapter 2:

1. Any recursive call is made on an argument that is *smaller* than the original one.
2. The answer for the smallest input is defined *without* using recursive calls.

We next ask the students if this reasoning process sounds familiar to them. Once they realized the resemblance to induction, we write the following proof template:

1. $P(1)$ holds.
2. $P(k+1)$ holds assuming $P(k)$ holds.
3. Therefore $P(n)$ holds for all natural numbers n.

We now explain how the function definition and the proof correpond to each other: the two cases, the hypothesis, and the universally quantified conclusion. Lastly, we present the following table:

	Case 1	Case 2	Assumption	Conclusion
Recursion	$n = 1$	$n > 1$	$fac(n-1)$ defined	$fac(n)$ defined for all n
Induction	$n = 1$	$n = k + 1$	$P(k)$ holds	$P(n)$ holds for all n

After introduction of recursion, we usually work through more recursive functions in the subsequent lessons. In Ochanomizu University, the next thing students do with recursion is sorting. Among various sorting algorithms, *quick sort* is a popular example that relies heavily on recursion. When sorting an array (*i.e.*, a sequence of elements) using quick sort, we begin by picking a pivot value p. We next reorder the array so that elements smaller than p are put in the left half, and elements larger than p are put in the right half. Then, we sort each of the left and right halfs by repeating the previous steps, obtaining a completely sorted array. Note that the last step uses recursion, where the argument is the left or right half of the array. In the C programming language, the quick sort algorithm can be implemented as follows (assuming that there is a globally defined array called `table`):

```
void sort (int left, int right) {
     int i, j;
     int pivot;

     if (left < right) {
        pivot = table[(left + right) / 2].key;
        i = left;
        j = right;
        while (i <= j) {
              while (table[i].key < pivot) { i++; }
              while (table[j].key > pivot) { j--; }
              if (i <= j) {
                 swap (i, j);
                 i++;
                 j--;
                 }
              }
             sort (left, j);
             sort (i, right);
    }
}

void quicksort () {
     sort (1, n);
}
```

The function `sort` takes in two integers `left` and `right`, which represent the indices of the left and right extents of the array we wish to sort. After setting the pivot value to the middle element of the array, we

search for elements that are currently in the left half but should be put in the right half, and elements that are currently in the right half but should be put in the left half, then swap them via the `swap` function. When there is no more element to move, we recurse on the left and right halfs of the array by making two recursive calls. To start sorting of the whole array, we simply call the `sort` function with arguments `1` and `n`, where `n` is the length of `table`.

Compared to the factorial function, the `sort` function looks less like a proof by induction. What makes the connection harder to see is the fact that the function receives two integers. These integers tell us the range of the array we wish to sort, hence we can regard the function as being defined on the length of the array. With this in mind, let us look at the function body. It has an `if` expression, whose condition is `left < right`. This condition essentially means "if it is non-trivial to sort the array"—if the condition does not hold, that means the array has no element, in which case the array is already sorted. In a proof by induction, non-trivial computation happens in the inductive case, therefore the condition represents "Case 2". We then find that the case uses two induction hypotheses, namely `sort (left, j)` and `sort (i, right)`. This is because we have divided the problem into two smaller subproblems: sorting of the left half, and sorting of the right half. Now we are left with one last question: what is the task for "Case 1", where we have a sorted array? The answer is "nothing"! Indeed, we can see that the `if` expression is missing an `else` clause, which is fine because there is nothing to do in that case. As we have covered both the non-trivial and trivial cases, we conclude that we can sort an array of any length using the `sort` function.

The `sort` function has a close connection with the following mathematical proof.

Prove that $x^n + y^n$ is an integer when both the sum and the product of x and y are integers.

(1) Suppose $n = 1$. In this case, $x^n + y^n$ is obviously an integer.

(2) Suppose $n = 2$. In this case, $x^n + y^n = (x + y)^2 - 2xy$, which is an integer.

(3) Suppose $n = k + 2$, and $P(k), P(k + 1)$ hold. In this case, $x^n + y^n = (x + y)(x^{k+1} + y^{k+1}) - xy(x^k + y^k)$, which is an integer.

Therefore, the proposition holds for all natural numbers n.

The connection between `sort` and the above proof is summarized in Figure 15.

	Case 1	Case 2	Assumption	Conclusion
`sort`	$\mid$ array $\mid < 1$	$\mid$ array $\mid \geq 1$	`sort(left, j)` and `sort(i, right)` sorted	`sort` can sort any array
Proof	$n = 1, 2$	$n > 2$	$P(k)$ and $P(k + 1)$ hold	$P(n)$ holds for all n

Figure 15: Correspondence between `sort` and Induction Proof

5.3 Experiment

In the fall semester of 2018, we incorporated our approach into a freshman course called "Data Structure and Algorithm". So far, we have done two experiments. On November 29, we introduced the students to the factorial function, showing its relationship to a typical proof by induction. On December 6, we gave a full lecture on the quick sort algorithm, also with a discussion of how it relates to an induction proof. To evaluate students' understanding, we further asked the students to participate in the following table-filling task:

	Case 1	Case 2	Assumption	Conclusion
Proof $fac(n)$	$n = 1$	$n = k + 1$	$P(k)$ holds	$P(n)$ holds for all n
Correct Ans	18	12	15	16

The last row shows the number of correct answers for each blank (the total number of students was 32). We find that a lot of students could not fill in the first two blanks correctly. One frequent reason was that the students misunderstood the question: they wrote what to do in the base and inductive cases, instead of what condition holds. For the second blank, there were also students who wrote an answer involving the variable k, which is not used in the factorial function. Similarly to our previous experiment, multiple students answered that the induction hypothesis is "$fac(n-1)$ holds". However, we got significantly more universally quantified conclusions, which we consider is the effect of our teaching method.

6 Conclusions and Perspectives

In this thesis, we demonstrated our approach to teaching induction and recursion. To make the learning of induction more appealing, we designed an introductory programming course where students build a mini game using recursive functions. Although the course was only given at one high school, the participants' feedback tells us what worked well and what could be improved. We then shift our attention to the students' actual understanding of induction, and give a test to both high school students and CS-major university students. The results once again justified the folklore that being able to use induction does not mean being able to explain how it works; in particular, we identified students' difficulties in understanding the role of the base case and the induction hypothesis. Lastly, we proposed to explicitly teach the connection between induction and recursion, and implemented our method in an undergraduate course. While the experiment is still ongoing, our emphasis on the universal quantifier seems to help students figure out what a recursive function does for us.

Nowadays, programming is finding its way into school education all over the world. This means, future students might encounter concepts like coordinates and induction in programs before learning them in math class. We believe that the lessons learned from our research would help both programming and math teachers find out an effective way to teaching difficult topics. We also plan to look for other notions from mathematics that can be learned by means of programming.

Acknowledgements

We gratefully acknowledge Ghourabi Fadoua for advising and encouraging us over the past four years. We also thank Saori Yabumoto, Hein Sprong, Marieta Braks, Matthias Felleisen, and Andreas Abel

for making our global internships fruitful, and to Katsuhisa Kagami, Hideyuki Majima, Yoko Iwata for providing us valuable feedback. Special thanks to Kawagoe Girls' High School and CS students at Ochanomizu University for participating in our lectures and surveys. Finally, we thank the Leading Graduate School Promotion Center for offering us wonderful opportunities and financial support. The unique experience we had in the Leading Program would absolutely be useful in our future career.

References

[1] Aichi Prefectural Education Center. A study on reinforcing teaching: What are easy to teach but difficult to understand. http://www.apec.aichi-c.ed.jp/shoko/kyouka/math_ishiki/kyoukashido_math.pdf, 2010.

[2] Bootstrap. http://www.bootstrapworld.org/.

[3] J. Clements, M. Flatt, and M. Felleisen. Modeling an algebraic stepper. In *European symposium on programming*, pages 320–334. Springer, 2001.

[4] Y. Cong and K. Asai. Implementing a stepper using delimited continuations. In *7th International Symposium on Symbolic Computation in Software Science (SCSS '16)*, volume 39 of *EPiC Series in Computing*, pages 42–54, 2016.

[5] Y. Cong and A. Mito. Induction via recursion: A proofs-as-programs approach to math education, 2018. Presented at 7th International Workshop on Trends in Functional Programming in Education (TFPIE 2018).

[6] T. Furukawa, Y. Cong, and K. Asai. Stepping OCaml, 2018. Presented at 7th International Workshop on Trends in Functional Programming in Education (TFPIE 2018).

[7] U. Leron and R. Zazkis. Computational recursion and mathematical induction. *For the Learning of Mathematics*, 6(2):25–28, 1986.

[8] M. Palla, D. Potari, and P. Spyrou. Secondary school students' understanding of mathematical induction: Structural characteristics and the process of proof construction. *International Journal of Science and Mathematics Education*, 10(5):1023–1045, Oct 2012.

[9] I. Polycarpou. Computer science students' difficulties with proofs by induction: An exploratory study. In *Proceedings of the 44th Annual Southeast Regional Conference*, ACM-SE 44, pages 601–606, New York, NY, USA, 2006. ACM.

[10] G. Ron and T. Dreyfus. The use of models in teaching proof by mathematical induction. *International Group for the Psychology of Mathematics Education*, 2004.

[11] E. Schanzer, K. Fisler, S. Krishnamurthi, and M. Felleisen. Transferring skills at solving word problems from computing to algebra through bootstrap. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 616–621, New York, NY, USA, 2015. ACM.

[12] G. J. Stylianides, A. J. Stylianides, and G. N. Philippou. Preservice teachers' knowledge of proof by mathematical induction. *Journal of Mathematics Teacher Education*, 10(3):145–166, Jun 2007.

[13] The Ministry of Education, Culture, Sports, Science and Technology. Programin. http://www.mext.go.jp/programin/.

[14] The Racket Programming Language. https://racket-lang.org/.

[15] D. R. Thompson. Learning and teaching indirect proof. *Mathematics Teacher*, 89(6):474–482, 1996.

[16] P. Zorn. *Understanding real analysis*. CRC Press, 2017.

Youyou Cong and Akiko Mito
2-12-1, Ookayama, Meguro-ku, Tokyo 1528550, Japan
cong@c.titech.ac.jp, akiko.mitokzd@gmail.com