Recursion with Factorials

By Richard Born
Associate Professor Emeritus
Northern Illinois University
rborn2@niu.edu

Introduction

The purpose of this lesson is to provide an interactive way for students to investigate factorial recursion. This is accomplished by OzoBlockly programs in which Evo accepts input from the user in the form of Evo button presses. Evo then outputs responses by speaking the value of n! and the number of recursive calls required to compute *n*!. Two different recursion algorithms are compared for efficiency, one with a process called *memoization* and one without.

Every positive integer *n* has a factorial denoted by *n*!, where *n*! = *n* x (*n*-1) x (*n*-2) x (*n*-3) x… x 3 x 2 x 1. As an example, 5! = 5 x 4 x 3 x 2 x 1 = 120. By convention, 0! is 1, the multiplicative identity. Factorials are used in numerous areas in the study of mathematics, especially combinatorics.

In this lesson, we study recursion with factorials. To recur means to occur again repeatedly or periodically. Recursion in computer science is when a function is defined in terms of itself. Although this might imply that there is an infinite set of values for the function, it avoids this by having a base case that does not use recursion. There is then a set of rules that reduces the remaining cases toward the base case, thus avoiding infinite repetition.

For recursion with factorials, the base case could be the fact that 0! = 1. Then the recursive rule would be that *n*! = *n* x (*n* – 1)! For example, 3! = 3 x 2! = 3 x 2 x 1! = 3 x 2 x 1 x 0! = 3 x 2 x 1 x 1 = 6.

Factorial Recursion Function #1

Let's take a look at the first OzoBlockly function "to factorial with x" that we will using to investigate recursive factorials. This function is shown in Figure 1. This function will return the value of *x*!. The variable *facCount* will keep track of the number of recursive calls. We do this so we can get an idea of the efficiency of the algorithm contained in the function.

We first check to see if *x* > 5. This is done because 5! = 120 is the largest number that can be held in an OzoBlockly variable. In the event that *x* > 5, a "fail" block is executed. This block allows us to set our own reasons for a program to fail. Failure will simply cause Ozobot to produce a short animation of red-blue flashes 10 times, and then stop the program.

In the event that *x* = 0, then the variable *fac* is set to 1, the base for our recursion algorithm. Otherwise, *fac* is set to *x* * (*x* – 1)! in a recursive call to our function. The variable *facCount* is also increased by 1 since we just made a recursive call. Finally, the function returns the value of fac to the caller.

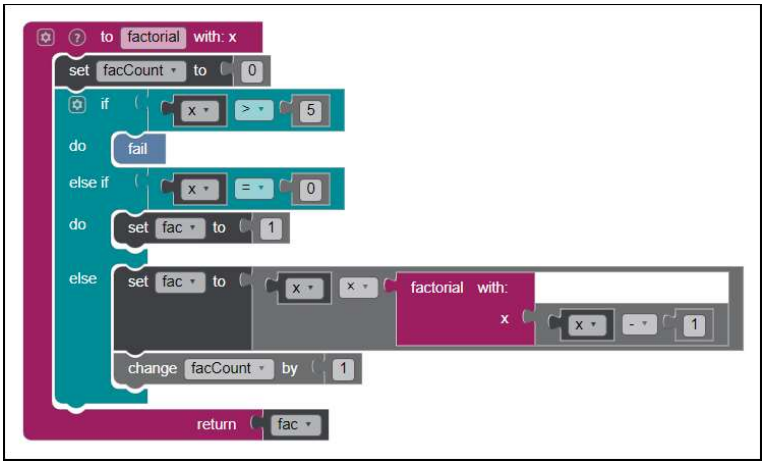

Figure 1

It should be noted that "to factorial with x" is the ***function*** and "factorial with x" is a statement that ***calls*** the function. With the caller inside the function, we clearly have a recursive algorithm.

Running the OzoBlockly Program for Factorial Recursion Function #1

1. Load the program *Recursive Factorials.ozocode* into Evo.
2. Place Evo on the gray circle in the Ozomap (***not*** the memoization map) facing the direction shown by the arrow.
3. Press Evo's button once to turn Evo on.
4. When the top LED turns blue, double-press Evo's button to start the program. Evo's top light will turn white and Evo will begin moving.
5. Upon reaching the intersection labeled "n", the light will turn red and remain red while the program runs. Press Evo's button from 0 to 5 times, representing the value of *n* for which you want to compute the value of *n*!. You will have seven seconds to do this. Each time you press the button, Evo will quickly flash white to let you know that the press was recognized.
6. Evo will then move to the intersection labeled "n!" and will speak the value of *n*!.
7. Evo will then move to the intersection labeled "Recursive calls this round". Evo will speak how many recursive calls were required to compute *n*!.
8. Evo will then move to the intersection labeled "Total recursive calls". Evo will speak the total number of recursive calls since the program was started.
9. Evo will then move around to the intersection labeled "n" and repeat the entire process. In the event t**h**at you click Evo's button more than five times, the program will "fail" as discussed earlier.

Hands on Exercise for Factorial Recursion Function #1

Each time Evo circles the Ozomap will be called a round. The table of Figure 2 is designed for you to collect data on eight rounds. In the shaded row, you should record your guesses as to the number of recursive calls for each of the rounds. ***Do this before starting Evo***. Then start Evo and fill in the rest of the rows from information that is spoken by Evo. If your guesses were correct—great! If not, figure out where you went wrong with the help of your teacher, if necessary.

Round #	1	2	3	4	5	6	7	8
n (entered by you with button presses on Evo)	0	2	1	3	4	5	3	5
Your ***guess*** for the number of recursive calls this round								
n! (from Evo)								
Recursive calls **this round** (from Evo)								
Total recursive calls (from Evo)								

Figure 2

Factorial Recursion Function #2

Hopefully, you have noticed that factorial recursion function #1 was lacking in efficiency. When 3! was called twice, in rounds 3 and 7, ***both*** rounds required 3 recursive calls— each calling 3!, 2!, and 1!. Similarly, when 5! was called in rounds 6 and 8, each round required 5 recursive calls. It would be more efficient if the values for 3! and 5! had been saved somewhere and then grabbed the second time that they were called.

When 4! was called in round 5, 4 recursive calls occurred. But back in round 4, 3! was computed. It would have been more efficient if the value of 3! had been remembered somewhere, and that this value was then recalled to help in computing 4!

A technique known as memoization (*not* memorization) can be used to reduce the number of computer cycles required when making identical recursive function calls. With this technique, results of function calls with specific inputs are saved in a data structure. With OzoBlockly, we will use the ***array*** data structure. When a specific call is made the first time, the result is written to the appropriate array element. Then when that specific call is made again, it is retrieved from the array *rather than computing it recursively all over again*. An array makes an ideal data structure here since we can directly access any array memory location without a need to search the array.

Figure 3 shows the pseudocode for the process of memoization. Our modified OzoBlockly function "to factorial with x" will reflect this pseudocode.

if ***n* = 0, then return 1**
else if ***n* is in the array, then return the array's value for *n***
else **{Calculate *n* * (*n* – 1)!;**
Store the result *n*! in the array;
Return the result}

Figure 3

Figure 4 shows the OzoBlockly code for our modified factorial recursion function. The fail case and the case where x = 0 are the same as for the original function. Also, the counting process for recursive calls is the same as it was in the original function.

Notice that two arrays have been declared, *memo* and *memoset*, both of length 6. Array elements are therefore identified with indices from 0 through 5, inclusive. All elements of both arrays are automatically initialized at the value 0 when the program starts running.

Element [*n*] in the array memoset, will be set to the value 1 when the corresponding value of *n*! is calculated for the first time. Similarly, the value of *n*! is also written to memo[*n*]. These can be seen in the *else* clause of both the pseudocode of Figure 3 and the OzoBlockly code of Figure 4.

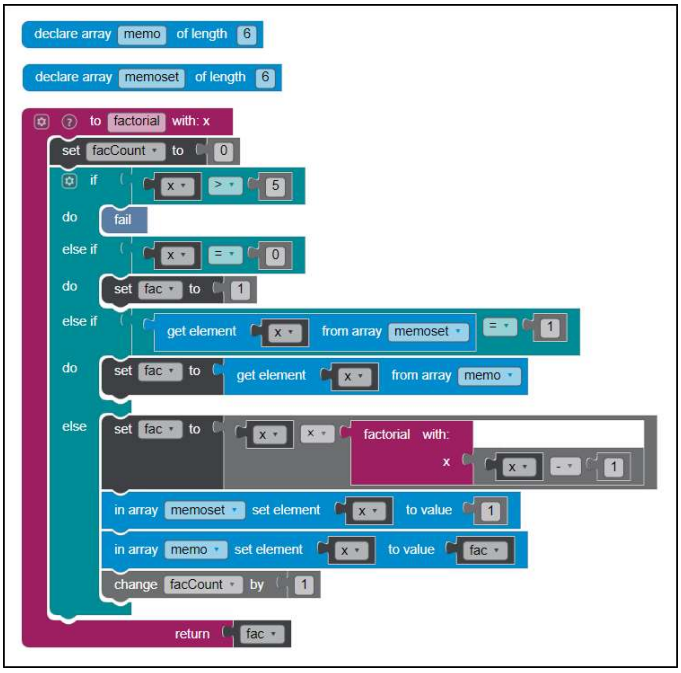

Figure 4

The *else if* clause checks to see if *n*! has been previously computed. This is done by looking to see if *memoset*[*n*] has been set to 1. If so, then the value of *n*! is grabbed from *memo*[*n*].

Hands on Exercise for Factorial Recursion Function #2

For this exercise you will need to load Evo and run the OzoBlockly program *Recursive Factorials with Memoization.ozocode.* Also, you need to use this program with the Ozomap "Recursion with ***Memoization***". You will notice on this Ozomap that five intersections have been added. Evo will speak the current values in the ***memo*** array at the end of each round—*memo*[1], *memo*[2], etc., through *memo*[5]. This will let you know which factorials have already been computed from the start through the current round.

Each time Evo circles the Ozomap will be called a round. The table of Figure 5 is designed for you to collect data on eight rounds. In the shaded row, you should record your guesses as to the number of recursive calls for each of the rounds. ***Do this before starting Evo***. Then start Evo and fill in the rest of the rows from information that is spoken by Evo. If your guesses were correct—great! If not, figure out where you went wrong with the help of your teacher, if necessary.

Round #	1	2	3	4	5	6	7	8
n (entered by you with button presses on Evo)	0	2	1	3	4	5	3	5
Your ***guess*** for the number of recursive calls this round								
n! (from Evo)								
Recursive calls **this round** (from Evo)								
Total recursive calls (from Evo)								
memo[1] (from Evo)								
memo[2] (from Evo)								
memo[3] (from Evo)								
memo[4] (from Evo)								
memo[5] (from Evo)								

Figure 5

Wrap-up Questions related to the table of Figure 5:

1. Why are there 0 recursive calls in round 1?
2. Why are there 0 recursive calls in round 3?
3. Why is there only 1 recursive call in round 4?
4. If we had done a round 9, how many recursive calls would that round have had and why?
5. No matter how large the number of rounds, what is the maximum number of ***total recursive calls*** that we would see?