

Understanding Recursion

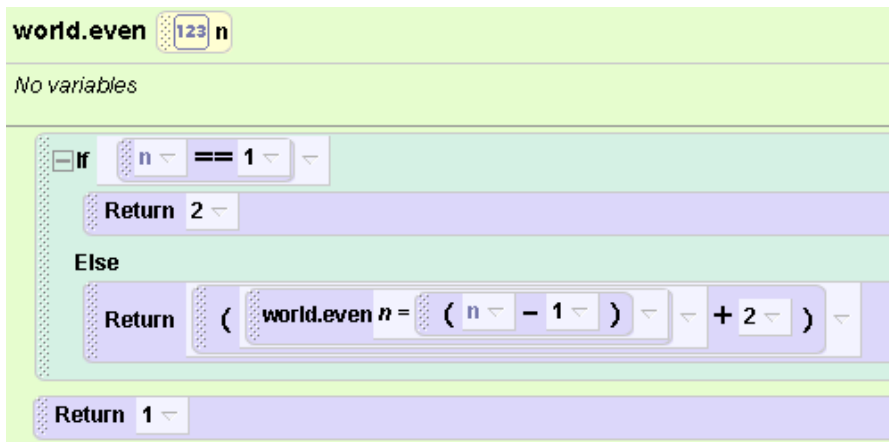
In this session you will do activities to give you a better feel about recursion. You should do the work in pairs. If you get stuck creating a statement or other problem, you can seek help from others.

Finding a number

In this part you will run an Alice program that creates a result. Get the Alice program “8 even orig.a2w” from your peer leader. Run the program and enter some small to modest integers when it asks for a value. Can you figure out what the program does? (Don’t read ahead to get hints!) You can ask others to see if you have what they think is correct.

Look at the code and determine the base case. What is the smaller problem that is being solved? You can ask others to see if you have what they think is correct.

To better see what the program is doing, you will add print statement so you can tell what it does at each step. The original code is:



The screenshot shows the Alice programming environment. At the top, the function name is "world.even" with a parameter "n" and a value of "123". Below this, it says "No variables". The code is as follows:

```
if (n == 1)
  Return 2
else
  Return ( world.even n = ( n - 1 ) + 2 )
Return 1
```

We want you to see each time the function is called and what is happening right before it returns a value. The problem is that it makes a recursive call in the “else” and then immediately returns. This makes it impossible to add a print statement. To accomplish the goal, you need to store the value you are going to return. In a computer program you use a variable to accomplish this. You probably noticed the line in Alice methods/functions for variable but we have not yet discussed them in class. Think of a variable as a box (memory location in your computer) that can hold a value. Once you put something into the box (variable), you can look at it. If you change what is in the box, whatever was there disappears and is replaced with the new item.

The modified code will look like:

```

world.even 123 n
123 value = 1
print In even, n = n
if n == 1
  print In even at base case so return 2
  Return 2
Else
  value set value to ( world.even n = ( n - 1 ) + 2 )
  print In even, about to return value
  Return value
Return 1

```

Here are a few hints on creating this code:

- You create a variable in a similar way to creating a parameter. Just click the variable button. The variable will be set equal to 1 by default in the area defining it. The value does not matter since you will set it before you use it.
- To add the statement to set the value of your new variable, drag the tile for the value into the code. In the menu that pops up, choose any number. Drag the function “even” over the value to create the recursive call to set value. To change the parameter sent to “even”, click on the arrow next to its value and choose “expressions” and then the parameter you want. By clicking on the arrow you can do math to subtract 1 and then add 2 to the entire value. Make sure the parenthesis are like above so it is correct. You want to send the parameter n-2 and then add 2 to what is returned. You need to remove the original return statement and add the one to return value.
- When you drag in the print statement, choose “object”, then “even”, and finally click on the variable or parameter you want to print. The print will have an empty tile at the front that you can click to add in the text. If you do it the other way around, you cannot add in the variable

Once you have the code set up, run it with a small value to see what happens. For 3, you should get:

In even, n = 3.0

In even, n = 2.0

In even, n = 1.0

In even at base case so return 2

In even, about to return 4.0

In even, about to return 6.0

the value of world.my first method.___Unnamed0___.___Unnamed0___ is 6.0

Try some other value and convince yourself you feel comfortable with what is going on. If not, ask some questions.

Modifying the program

Create a new function named `odd`. Hopefully you figured out that the original program produced the n th even number where n is the value you give when it runs. In your new function, create a recursive routine to give the n th odd number. It is very similar to `even`. You probably cannot copy the “even” code into “odd” so you will need to recreate it. Be careful to choose parameters/variable for your “odd” function and not from “even” (they will both show as choices).

Once it works correctly, try adding print statement like in `even` to see what it does and how it is different.

See how others did their program and compare answers.

Creating a function to do sums

The next step is to create a recursive function that calculates the sum of the first n numbers. The first and most important step is to state the problem recursively. From that you can usually create the recursive solution easily. Hint: it isn't very different from what you have seen so far! Once you think you have accomplished this step, talk to others to see what they came up with.

Now implement your function. Test it out (start with small values!). Add print statements to trace what it does. Talk with others to make sure you are comfortable with the final code.

Finding a formula for the answer

Gauss, a famous mathematician from a couple of hundred years ago, annoyed his grade school teachers by doing his work too quickly (or so the story goes). To keep him busy, his teacher told him to find the sum of the first 1000 numbers. He came back with the answer in a short amount of time. He didn't have a computer so he could not do what you did above. He also didn't do it by adding them very quickly. He worked out the formula so he could calculate the sum of the first n numbers by only doing some simple arithmetic. Use your program to get some results and see if you can work out the formula. If you get stuck, work with others. Your peer leader can help if everyone gets stuck.

Using loops instead

As a final step (if you have time and energy), you can code the sum of the first n numbers using a loop. It should be fairly easy but you need a variable to hold the sum. If you just change the function you originally call, you should get the same answer as your recursive code.

To be honest, the loop is easier to do and understand. However, the recursive code hopefully helped you see how things work. Some other examples in class should show why recursion is often useful in practice.

Very slow recursion

Obvious recursive solutions are often the best but they can be very slow. The n th Fibonacci number is defined as:

$$F(n) = F(n-1) + f(n-2)$$

where $F(1) = 1$ and $F(2) = 1$. Thus, $F(3) = F(2) + F(1) = 1 + 1 = 2$. $F(4) = F(3) + F(2) = 2 + 1 = 3$. $F(5) = 5$. Get the code `fib.a2w` from your peer leader. Run it for small values of n . As you increase the value, you will see that it take a **long** time. By looking at the code and/or adding print statements see if the group can figure out why.