MTH 3270 Final Project Weekly Report

Carlos Galvan Jackeline Escalante

Week of 03/30/2020

Click to visit a Live Demo.

What did you try?

For this week report we decided to take our findings and code a programs that both finds the least amount of ones needed to calculate a given number (n), and that it takes a given number and extracts it in to a expression that consist of only 1s, addition, multiplication, and grouping.

Looking back at our notes we noticed that our way of breaking down the number n, is very similar to factoring except with extra steps. With our background of computer science, we concluded that we can use recursive functions to do so. In terms of sudo-code: our function must recognize if the input is even or odd. If the input is even then we divide it by 2 (n/2), and if it is odd it subtract it by one to make it even ((n -1) / 2). It also needs a way to escape the recursive function otherwise it will loop forever or until it creates an error. So to do so, our escape clause will be if the input is one.

In the next section, the program will be split function by function, to describe their role and logic with in the program. A link to a live demo will be provided:

recursionPrinting()

```
def recursionPrinting(n, str):
    if n == 0:  # if no number, then 0
        str = "0"
        return str
    elif n == 1:  # once n has been broken down to 1
        str.append(n)
        return str
    elif (n % 2) == 0:  # is even
        str.append(n)
        n /= 2  # n/2
        return recursionPrinting(n,str)
    elif (n % 2) == 1:  # is odd
        str.append(n)
        n = (n - 1)  # (n - 1) + 2
        return recursionPrinting(n, str)
    else:
        print("Not a valid input")
```

```
>print(recursionPrinting(10,[]))

[10, 5.0, 4.0, 2.0, 1.0]
```

The recursionPrinting() function is like its name states is a recursive function that requires 2, arguments to work. The first argument is the initial number that we would like to find its the smallest number of ones required to write it. The second argument is reserved for an empty list, which when done returns the said list only that it will no longer be empty. For every iteration, the program will append the current state of n to list, until the value of n equals one. The state of n is always changing, so if n is an even number it would be divided by 2, or if its odd it will be subtracted by 1. This process may seem pointless at first glance, but it is crucial for us since the given list acts as a blueprint to be able to generate a mathematical statement.

eqFormat()

```
def eqFormat(n):
    eq = ""
    for i in range(len(n)):
        if n[i] == 1: #starting point
            eq = "1"
        elif n[i] == 2: #the second step, requiered to
    keep things neat
            eq = "( " + eq + " + 1 )"
        elif (n[i] % 2) == 0: #if the number is even,
   then opposite of /2 is ( X 2 )
            eq = "( " + eq + " * ( 1 + 1 ) )"
        elif (n[i] % 2) == 1:# if the number is odd,
   the opposite of - 1 is + 1
            eq = "( " + eq + " + 1 )"
        else: #basic else statement
            print("something went wrong")
    return eq
```

```
> n = [1, 2, 4, 5, 10]
> print(eqFormat(n))

( ( ( ( 1 + 1 ) * ( 1 + 1 ) ) + 1 ) * ( 1 + 1 ) )
```

This function is responsible for translating the n to a fully written out expression. *eqFormat()* requires a list as an argument for it to print it. To get the required argument we must first format the output of *recursionPrinting()* function. Basically we just reverse the function, and convert it back to integers since *recursionPrinting()* creates a list of floats. Once we plug in the formatted list, this new function uses it to add the appropriate strings to base strings to flush out the expression.

The logic go as follows:

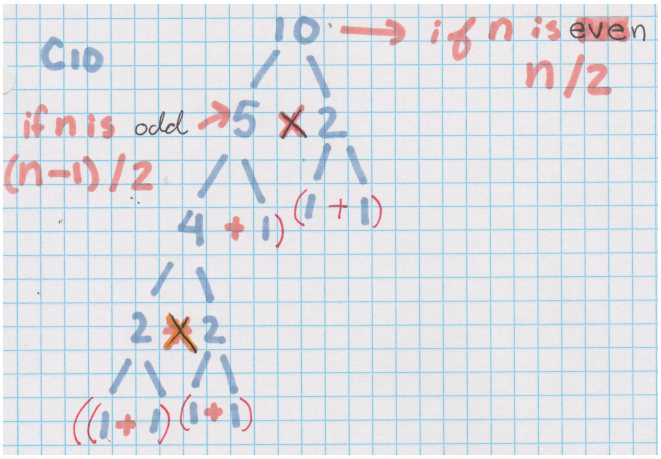
- if the number is one, it indicates that this is the first step to the expression and start it of with a "1"
- if the number is two, it appends a "(" before it and a " + 1" string after it
- if the number is even, it means that we multiply by 2 (since it's the inverse of n/2) so it appends a "(" before it then a " * (1 + 1)" string after it
- if the number is odd, it means that we add one to it (since the inverse is n – 1) appends a "(" before it then a " + 1)" string after it

After the function has finished, it returns a string that represents a mathematical expression that is equivalent to the original number n. We can verify

this by calling a special python function called eval(). The eval() function takes a mathematical expression that is in a string format and then returns the result of it. With that we can prove that our expression is in fact legal.

What did you observe?

When researching our notes for designing the program, we observed the pattern that would be used as the foundation of our code's logic. That pattern is linked into to steps n takes to tun into one and vice versa. This pattern can be seen clearly in the binary tree depiction. Its no coincidence that the entire left side matches up with the list generated by the recursionPrinting() function.



What are your next steps?

Our next step for our project is to disprove the idea that this way generates an expression with the least number of ones. To do so we will run test out different formulas that either build off our current idea or maybe an entirely new approach. If we have the time, we will also want to code at least one of those concepts and compare it with this one.