20200117

At this point we closed our introduction of the Omega and Theta complexity classifications, and finally turned our attention to a data structure: **the stack**

Consider the problem of evaluating an arithmetic expression such as **3 + 4 * 7 + 8 * (3 - 1)**

Most people in North America have been taught that **parentheses** have highest precedence, followed by **exponentiation**, then **multiplication and division**, then **addition and subtraction**, so the expression above evaluates to 3 + 28 + 16 = 47

But these precedence rules are completely arbitrary. For example, we could keep the rule about parentheses but do everything else in simple left-to-right order ... which would give 114 ... or right-to-left order ... which would give 103 ... or give addition higher precedence than multiplication ... which would give 210 (assuming I have done the calculations properly)

The notation we have used here to write down the expression is called **infix notation** because the operators (*, +, etc) are placed between the operands (3, 4, 7, etc)

In order to evaluate an infix expression correctly we need to know exactly what rules of precedence were assumed by the person who created the expression. Wouldn't it be wonderful if there were a universal way to represent an expression so that no matter what rules of precedence are in use, the method of evaluating the expression is always the same?

There is! It is called **postfix** notation, and it was invented in 1924 by Jan Lukaseiwicz ... because he was Polish, this is sometimes called Polish Postfix notation. In postfix notation, operators come **after** their operands, so "**3 * 4**" (infix) becomes "**3 4 ***" (postfix)

The expression we started with : **3 + 4 * 7 + 8 * (3 - 1)** can be written as **3 4 7 * + 8 3 1 - * +**

We can evaluate this in simple left-to-right order ... we keep going until we hit an operator (the *) and then we apply it to the two numbers just before it: 4 * 7 = 28, and we put the result in place where the 4 7 * was, so now the expression is **3 28 + 8 3 1 - * +**

The next thing we see is the +, which we apply to the numbers just in front of it (3 and 28) and put the result back in the expression, giving

31 8 3 1 - * +

The next operator we find is -, which we apply to 3 1. The expression is now

31 8 2 * +

The next operator is *, applied to the 8 2. This gives

31 16 +

We apply the + to the numbers before it, giving a final result of 47 ... which is exactly the result we expect from the original expression using standard rules of precedence ... but note that we did not need to know those precedence rules. Once we have the expression in postfix form we just evaluate from left to right.

But something magic happened there - I just pulled the postfix version of the expression out of thin air. Can we find a way to convert any infix expression to an equivalent postfix expression?

We can build it up one piece at a time ... for example, we can look at **3 + 4 * 7 + 8 * (3 - 1)** and see the parenthesized part "(3 - 1)" which we can immediately write as "3 1 -" and now that this is taken care of, we can work on the next level of precedence: multiplication and division. We can see that "4 * 7" will become "4 7 *", and that the "8 *" combines with the "3 1 -" to give "8 3 1 - *". Now all that we have left to deal with are the additions. We can resolve these in different ways, but perhaps the simplest is to put the first "+" right after the things it applies to (the "3" and the "4 7 *") and the second "+" after the "8 3 1 - *"

The original expression could also be translated into postfix notation as **3 4 7 * 8 3 1 - * + +**

Let's consider the problem of evaluating a postfix expression. We can do it by traversing the expression from left to right. We "hang on to" the values, and whenever we encounter an operator we apply it to the two most recent values, then replace those values with the result of the operation we just performed. We can visualize this process as making a pile of the values, placing each new value on top of the existing pile. When we do an operation, we remove and use the top value and the next value, then put the result on the pile.

The data structure we use to make this work is called a **stack**. (The classic model is a stack of plates – or pancakes.) With a stack of pancakes, we can add new pancakes to the top of the stack, and we can remove the pancake currently on top – and that's pretty much all we can do. But for our problem, that is all we need.

A stack is our first example of an Abstract Data Type: we specify the operations we need to be able to perform on the data we will store, but we do not specify the details of the implementation. Of course when we actually write code we do need to choose a specific implementation, and the choice we make will often have significant impact on the efficiency of our program. We'll talk about implementation later.

A stack must provide (at least) three operations:

push(x) - add the value x to the top of the stack
pop() - remove the most recently added value, and return it
isEmpty() - return True if there are no values in the stack, and False otherwise

Sometimes other operations are also defined, such as
peek() - return the top value on the stack without popping it
count() - return the number of values on the stack-based

These (and others) are optional because they can be implemented using just push, pop and isEmpty.

In keeping with popular practice, we will imagine that we have implemented a **Stack** class and that we can create a stack with a statement such as
S = new Stack()
Then the operations listed above become methods attached to the stack we create.

A stack is often described as a LIFO (Last In First Out) data structure: the most recently added (pushed) value is the first one removed (popped). The first thing to notice about a stack is that it automatically reverses the order of a sequence:

```
S = new Stack()
S.push(1)
S.push(3)
S.push(7)
S.push(9)
print S.pop(), S.pop(), S.pop(), S.pop()
```

will print **9 7 3 1**

It's a very interesting exercise (and a useful one!) to think about how we can write stack-based algorithms for practical tasks. For example, how might we create an algorithm that sorts the values in a stack, using only the defined operations?

To get a start on this, consider this algorithm. It takes a stack of unique numbers as its argument and returns the stack with the smallest value moved to the top of the stack, and all the other values in their original order. Note that it creates another (temporary) stack, but uses no other data structures.

```
def Min_to_Top(S):        # S is a stack of numbers, all different
        if S.isEmpty():
                return S
        else:
                temp = new Stack()
                min = S.pop()
                temp.push(min)
                while not S.isEmpty():
                        x = S.pop()
                        if x < min:
                                min = x
                        temp.push(x)
                # min is now the smallest value
                # all values have been moved to temp
                # S is empty
                while not temp.isEmpty():
                        x = temp.pop()
                        if x != min:
                                S.push(x)
                S.push(min)
                return S
```

Work through this algorithm to make sure you see how it works. Then modify it so that it will work on sets of numbers that may contain duplicates. Then write a sort algorithm for a stack that uses Min_to_Top. Your algorithm will most likely be in $O(n^2)$ where n is the size of the stack. Can you write an $O(n * \log n)$ stack sorting algorithm?

Now back to our original problem. Here is an algorithm that uses a stack to evaluate a postfix expression:

```
Let E be a string that represents an expression in postfix form
Assume E has a “next” method that returns the next token in E

S = new Stack()
while we haven’t processed all of E:
    x = E.next()
    if x is a value:
        S.push(x)
    else:
        # x is an operator
        let n be the number of values required for x (usually 2)
        pop the top n values from S
        y = the result of applying operator x to the values just
            popped off S
        S.push(y)
return S.pop()
```

This algorithm will fail if E is not well-formed. As an exercise, improve the algorithm by using the isEmpty() method to avoid problems.

It is worth noting that postfix notation is very important in computer science because it gives a good model how arithmetic is actually carried out in a computer. When we write a high-level statement like

C = A + B

it gets translated into assembly language sort of like this:

load the contents of address A into a CPU register
load the contents of address B into another CPU register
add the contents of those two registers and store the result in another CPU register
copy that register to address C

In other words the addition is really carried out in a postfix way: we identify the operands, then execute the operation on them