

## Recursive Programming Extravaganza

- **Power**
  - Implement an operation that raises a number to a positive integer power using recursive programming
  - *power(base, exp)*
- **Multiplication using only + and -**
  - Implement the multiplication of two numbers with recursive programming
  - *multiply(a, b)*
  - Start with only positive integers
    - Then try to make the solution work for negative integers as well
- **Factorial**
  - Implement the mathematical operation factorial with recursive programming
  - *factorial(n)*
  - The factorial of five:
    - $5! = 5*4*3*2*1 = 120$
    - `print(factorial(5))`
      - output: **120**
- **Print the first n natural numbers**
  - Implement an operation that prints the first n natural numbers using recursive programming
    - *output in a single line (OK, if starts or ends with space)*
  - *natural(n)*
  - Natural numbers: positive integers > 0
    - `natural(5)`
      - output: **1 2 3 4 5**
- **Sum of digits in positive integers**
  - Implement an operation that calculates and returns the sum of digits of a parameter using recursive programming
  - *sum\_of\_digits(x)*
  - Sum of all digits of the 10-base representation of the number
    - number of digits of 254 =  $2+5+4 = 11$
    - `print(sum_of_digits(254))`
      - output: **11**

- **Fibonacci**

- Implement a recursive function that returns a fibonacci number
  - The function takes the position in the fibonacci sequence as a parameter
  - Returns the value of the fibonacci number in that position
- Definition of fibonacci (for  $n \geq 0$ ):
  - $\text{fibonacci}(0) = 0$
  - $\text{fibonacci}(1) = 1$
  - $\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$
- Easy to do with two recursive calls in each iteration
  - The time complexity is then *exponential*:  $O(2^n)$
- Try to do it with only one recursive call in each iteration
  - *Linear* time complexity:  $O(n)$ 
    - **Hint:** *you will need to keep track of at least two numbers in each iteration, and move them on in the recursion*
    - **Hint:** *það þarf að halda utan um tvær tölur í hverri umferð, og fleyta áfram í endurkvæmninni*

- **Ackermann**

- *This one is more for fun than to solve any useful problem*
  - ***I recommend you finish the prefix parser first***
    - It's much more useful and interesting :)
- Read the definition of the Ackermann function
  - [https://en.wikipedia.org/wiki/Ackermann\\_function](https://en.wikipedia.org/wiki/Ackermann_function)
- Once you have it implemented, try some parameters
  - Which parameters do not give a StackOverflow?

- **Prefix parser**

- You are given a base for a program (***PrefixParserBase.zip***)
- In the base there is the class ***Tokenizer*** that splits a string on white-spaces and returns the next token whenever the function ***tokenizer.get\_next\_token()*** is called.
- Read the definition for prefix notation (or Polish notation).
  - [https://en.wikipedia.org/wiki/Polish\\_notation](https://en.wikipedia.org/wiki/Polish_notation)
- Write a recursive function that handles each token from a prefix statement correctly so that the correct result is eventually returned.
- Start by thinking about a very simple prefix statement.
  - **+ 4 3**
    - The first token is a plus, telling us that we can get the next two tokens and add them together:
    - **4 + 3 = 7**
    - *Wondering what to do when the token is a number?*
      - Remember that one number is also a valid prefix statement
      - **42 = 42**
- Then add complexity.
  - **++ 4 3 8**
    - After getting a plus sign, we encounter another operator. This means that instead of a single number, we finish evaluating that operator and return its result onto the first operator.
    - **+(+ 4 3) 8 = + 7 8 = 15**
- Add other operators.
  - **- 4 3 = 4 - 3 = 1**
  - **+ - 4 3 8 = + (- 4 3) 8 = + 1 8 = 9**
- Also add \* and /
  - You must detect when a division by zero is about to occur
    - Raise a *DivisionByZero()* exception