KEY

Week 1	Exercise 3	Ok Computer	20 Minutes

Goals:
Counting operations, computation time, algorithmic thinking, problem solving, pseudocoding

Description:
You have access to a primitive computer. It has four basic operations:

1. It can store an integer value in a named location in memory, ex. X = 0.
2. It can add two integer operands, ex. A + B. Given two operands, A and B, the add operation returns the sum of the two operands.
3. It can negate one integer operand. Let's call it A. That is, given A, the negate operation returns -A. If the input A is positive the result will be negative and if the input A is negative the result will be positive.
4. It can loop using a basic conditional check. The condition is checked at the top of the loop. If the condition is true when the top of the loop is reached, the process will repeat the contents of the loop. If the condition is false when the top of the loop is reached, the process will skip the contents of the loop. This is typically called a `while` loop.

When you program this computer, you always keep track of the number of **ops** that will be made over the entire run of the program as a measure of how long the program took to execute:

1. Storing a value will be considered free (**+0 ops per store**)
2. Adding counts as 1 op (**+1 ops per add**)
3. Negation will be considered to require 1 op (**+1 op per negation**)
4. The conditional check in the loop will require 1 op (**+1 op per condition check**)

Tasks:
Model the following functions for this computer and compute the number of **ops** for each implementation:

- Add two numbers together given a left and right operand. Call it something like:
  ```
  fx = add(left, right)
  ```
- Subtract two numbers given a left and right operand.
  Hint: Right operand is always subtracted from the left in subtraction.
  ```
  fx = sub(left, right)
  ```
- Multiply two numbers given a left and right operand
  ```
  fx = mul(left, right)
  ```

Hints:

- We are working with `int` types and operations, so there are no fractions or decimals, only whole numbers, positive and negative, including zero.
- `add` implements the basic machine add op.
- You can subtract by adding a negative number
- Multiply is the sum of the left operand right operand number of times. It involves a loop.

Extension:
Model these functions if you have time. Make sure to count **ops**:

- Divide two numbers given a left (dividend) and right (divisor) operand.
  Hint: Right operand (divisor) is "divided into" the left (dividend)
  ```
  fx = div(left, right)
  ```
- Compute the power given a base and exponent
  ```
  fx = pow(base, exponent)
  ```

Questions:

1. For which functions is the number of **ops** constant regardless of the value of the input?
   `add` and `sub` always involve the same number of operations every time. It does not matter how big or small the input variables are, these functions always use a constant number of operations to solve the algorithm.
2. For which functions is the number of **ops** dependent on the value of the input?
   `mul` (div and power) involve loops which means that the number of operations vary with the input value. If the input is a value close to zero, then the number of operations is small but if the input is a number far from zero, then the number of operations is large.
3. Why do some functions not vary in the number of **ops** and why do some functions vary in the number **ops**?
   The logic of the function may or may not be dependent on the input. If you have a loop in the function that is dependent on an input variable, then the number of iterations through that loop and subsequently the total number of operations is determined by the input value. `add` and `sub` have no loops so they are completely decoupled from the input and the number of operations will not vary by the input. However, `mul` (div and power) contain a loop that is dependent on an input value, so they will run faster with smaller inputs and slower with larger inputs.
4. What mathematical functions approximate the number of **ops** for each of your function implementations?
   `add` → f(x) = c :: c is constant and not dependent on the input x
   `sub` → f(x) = c :: c is constant and not dependent on the input x
   `mul` → f(x) = x :: the number of operations is linearly dependent on the input x due to the loop that is subject to the input x.
   `div` → f(x) = x :: the number of operations is linearly dependent on the input x due to the loop that is subject to the input x.
   `pow` → f(x) = x :: the number of operations is linearly dependent on the input x due to the loop that is subject to the input x.

Conclusions:

- We can model how much work (work → time) our imaginary program will do (and whether it can compute an answer on specified hardware) by counting how many fundamental operations it will need. We were able to make this model even though we

do not have a physical computer nor a specific language. We have been able to predict the performance of programs on our future computer simply by developing a metric and applying it to how basic operations will be implemented on that computer. From this, we can extend our estimates to more and more complex programs built on that computer. The computer is mostly irrelevant because we can measure the algorithm itself.

- We can (and will) generalize this idea even further, so don't get hung up looking for a specific ops measure exclusively in future work. We don't even need to know the specifics of a computer or programming language to estimate the computation time of an algorithm. We simply need a metric for our estimation and an understanding of the logical structure of the algorithm itself.
- Even a computer with only a few operations or instructions can do complex operations. For example, this machine does not provide a multiply operation but the operation can be implemented by combining add and loops. The integer multiplication function on a real computer is more complex and faster than our version, but it still holds that we can build complexity from simplicity. In fact, most processors support about a hundred operations and from this comes all the complex programs that we experience today.
- Pseudocoding is a critical skill. It allows us to model programs without worrying about the specifics of a language. Good pseudocode is completely language agnostic but easy to express in any language. With pseudocode, we can estimate the computational efficiency of any program without wasting the time to learn a language, write a program in specific syntax, and debug myriad issues that can arise. We can determine if it is worth the time to implement without spending that time. It also gives us a model for our implementation to help guide us when we are implementing.

Further Information:
Instructions per second : https://en.wikipedia.org/wiki/Instructions_per_second
Pseudocode : https://en.wikipedia.org/wiki/Pseudocode