What is an algorithm?

*An **algorithm**, loosely speaking, is a sequence of computational instructions that takes an input and produces a desired output.*

It is useful to view algorithms as tools for solving problems. Understanding exactly what an algorithm is and what it does is best understood with an example task that an algorithm solves.

Task 1: Sort a list of numbers into non-decreasing order.

Input: A list of n numbers, $(a_1, a_2, a_3, \ldots., a_n)$.
Output: The list of numbers sorted $(a_1', a_2', a_3' \ldots., a_n')$ *with* $a_1' \leq a_2' \leq a_3' \leq \ldots. \leq a_n'$.

Thus, an algorithm is called ***correct*** if it produces the correct answer/ output for all possible input values.

For example, a correct algorithm when given input: $(-2, 34, -11, 63, 0, 999, 100)$ will output the list: $(-11, -2, 0, 34, 63, 100, 999)$.

What is the complexity of an algorithm?

The complexity of an algorithm can be measured in two major ways. Firstly, the amount of ***memory*** needed in order for the algorithm to complete the task. Secondly, the number of *steps* the algorithm take. Or how much ***time*** does the algorithm need in order to complete the task.

We will investigate the second of these measures, ***the time complexity,*** of an algorithm. In order to understand time complexity it is best to consider the following example.

Task 2: Determine if the number x is in a list of ordered numbers.

Input: A number x and a list of n ordered numbers, $(a_1, a_2, a_3, \ldots., a_n)$ *with* $a_1 \leq a_2 \leq a_3 \leq \ldots. \leq a_n$.
Output: True if x is in the list and False if x is not in the list.

How could we tell a computer to solve this task?

A simple solution is as follows:

```
for i =1 to n:
    if $a_i$ = x:
        return True
return False
```

Let's understand the ***pseudocode*** above. Line 1 states that i will go through the values 1 through n, where n is the number of elements in the ordered list. Indentation, as we should know, is important: *all the code indented one tab under the for loop will run as i goes from 1 to n.*

The if statement on Line 2 checks if a_i is x. If the if statement is True then the indented code under the if statement runs, if the if statement is False then the indented code does NOT run.

Line 3 only runs if the if statement is True, that is if a_i is x. The return True halts the algorithm and an output of True is given.

Line 4 is on the same indentation of the for loop and only runs if the for loop runs i all the way to n. Meaning, that none of the a_i were x, therefore a return of False is output.

This is not the *fastest* way to solve this task since the algorithm above does not take advantage of the fact that the list is *ordered.*

In the above example if x is not on the list the algorithm must take at least n steps checking each element till it reaches the end of the list.

However, we could do something different:

1. We could start by looking at the middle number in the list and compare it to x.
2. If x is greater than the middle number, disregard the entire left part of the list, if x is less than the middle number, disregard the entire right part of the list, and if x is the middle number return True.
3. Then look at the middle number of the new list.
4. Repeat this process until the list is empty where we return False, or we find x.

If we consider the two algorithms running on a list of 1024 numbers where x is NOT on the list we see that the first algorithm will take 1024 steps, at least. In contrast, the second algorithm will take 10 steps at most.

Exercises:
1. Write code in python that implements the pseudocode for the first algorithm that solves task 1.
2. Write code in python that implements the outlined second algorithm that solves task 2.