Name: ___________________

Week 3: Algorithms

What is an algorithm?

We've all probably heard that companies use algorithms to solve problems, but what are they really? Algorithms are just a special way of explaining how to solve a problem by breaking the solution down into a series of discrete steps. Although we usually hear about algorithms in the context of computing and math, the problem they present a solution to can be anything! You can have an algorithm for calculating modulo, but you can also have an algorithm for baking a cake, or for driving a car! The best, canonical definition probably comes from Donald Knuth in his book *The Art of Computer Programming*: "An algorithm is a finite, definite, effective procedure, with some output."

What does this mean? Let's break it down.

- An algorithm is a procedure: An algorithm is a series of steps that are followed, in order, for a specific purpose. Any purpose works, not just mathematical ones! The algorithm may also specify deviations from the order, such as saying to skip a step if something is true, or to repeat some steps a certain number of times.
- Finite: An algorithm has to terminate eventually. It can't go on forever!
- Definite: An algorithm has to be well-defined, in the sense that each step has to be clear enough that it can be followed by someone with no outside context. Anyone should be able to pick up your algorithm and follow it, without ever being unsure of what to do next.
- Effective: This is not technically a requirement, but in practice, we always want to make sure that our algorithms are as quick and efficient as possible.
- "with some output": The algorithm must have a result, and show that result to the user! There's no point in doing all the work for an algorithm if we don't get a result, or if we don't show that result to the user! Note that "output" is pretty broadly defined - anything the user can observe that is a result of the algorithm counts. For instance, if we wrote an algorithm specifying how to make a PB&J, the sandwich would be our "output".

What would this look like in practice? Well, as an example, let's look at modulo! If you remember, we had a procedure for solving modulos of the form `x % y`, where x is the **dividend** and y is the **divisor**:

1) Find the first number **smaller** than or equal to the **dividend** (the number on the left of the '%') that is evenly divisible by the **divisor** (the number on the right of the '%'). Call that number `N`.
2) Calculate `x - N`. That's the result!

That seems pretty simple - does that qualify as an algorithm? Let's go through our list and check!

- Is it a procedure? Well, it's a series of steps, and it has the purpose of finding the modulo, so yes!
- Is it finite? Yes! There's nothing here that's impossible.
- Is it definite? Or, can anyone follow these directions? Well, the procedure for finding `N` in step 1 is kinda vague, how do we find the first number smaller than or equal to the dividend that is evenly divisible by the divisor? Let's go back and clarify.
- Is it effective? We can't tell, because we don't know how we're going to find `N` in step 1! This is another reason it's important to be definite - otherwise it will be very hard to analyze your algorithm.
- Does it have output? No! We calculate `x - N`, but we never do anything with it. We should fix that too!

Our new algorithm would look like this:

1) Check if our dividend (`x`) is negative.
2) If `y` is negative, flip its sign.
3) If `x` is positive or 0, start with 0 and repeatedly add multiples of our divisor (`y`) until adding another `y` would result in a number greater than `x`. Call this number `N`.
4) If `x` is negative, start with 0 and repeatedly subtract multiples of our divisor (`y`) until we get a number less than `x`. Call this number `N`.
5) Calculate and print `x - N`. That's the result!

Let's check this against our conditions:

- Is it a procedure? Well, it's a series of steps, and it has the purpose of finding the modulo, so yes!
- Is it finite? Yes! There's nothing here that's impossible.
- Is it definite? Or, can anyone follow these directions? Now that we've clarified how to find `N`, these steps can be followed easily by anyone, so yes.
- Is it effective? Well, as was mentioned up above, this isn't technically required, and the definition of "efficient" can be complicated, so we'll come back to this later.
- Does this have output? Yes, our output is `x - N`, or the result of the modulo!

Example Problems. For each problem, identify which requirement of algorithms it fails to meet: (Some of them may fail more than one!)

To Use Shampoo	Why is this not an algorithm? (Which

1) Wet hair in shower 2) Put shampoo in hair 3) Rinse hair 4) Repeat	requirement does it fail to meet?)
To Fly a Plane: 1) Take off 2) Fly to destination 3) Land	Why is this not an algorithm? (Which requirement does it fail to meet?)
1) Choose a number, call it x. 2) Multiply x by 4. 3) Subtract 3 from x. 4) Divide x by 5. 5) Print x.	Why is this not an algorithm? (Which requirement does it fail to meet?)
To Calculate Age: 1) Get the person's birthday 2) Compare the day they were born on with the current date. 3) If they were born before today's date, subtract the year they were born in from the current year. 4) If they were born after today's date, subtract the year they were born in from the current year, then subtract one.	Why is this not an algorithm? (Which requirement does it fail to meet?)

How Do We Use Algorithms?

Now that we have an algorithm, what can we do with it? Well, once we've figured out the logical steps by which to solve a problem, we'd like to make the computer do it. This is where all the Python you've been learning comes in! After you've defined a solution to your problem in algorithm form, figure out how to translate the steps in your algorithm into operations we have available in Python, such as making and using variables (like `x` or `N`), mathematical operations, if statements, while loops, and others that we will encounter later on.

Let's give this a shot with our example of modulo. Now, we obviously could use the modulo operation '%' to do this, but what we will be making is mathematical modulo, which is slightly different from Python's modulo.

What Python code would correspond to our first step, “Check if our dividend (x) is positive or negative.”? Well, we’re checking if something is true, and doing different things based on the result, so that sounds a lot like an if statement to me! So we know we’ll have an if statement, and we can start a rough outline of what the code might look like: (We’ll also include code to get x and y from the user. Later on we’ll see a better way of doing this with **functions**.)

```
int_x = int(input())
int_y = int(input())

if (int_x >= 0):

else:
```

Wait a minute, you might be saying, that’s not valid Python! You can’t have an if or an else without any statements after it! That’s entirely true - but we don’t need our Python to be valid just yet. We’re still in the middle of converting our algorithm into code, and it doesn’t need to be valid Python until we’re done and ready to run it. Odds are even after we’ve gone through all of the steps in our algorithm, we might still have to fix up a few things in order to make the program work, but that’s okay. What we care about is getting the **logic** correctly transferred from the algorithm we came up with into Python.

Let’s look at the next step: “If y is negative, flip its sign.” This is pretty easy:

```
int_x = int(input())
int_y = int(input())

if (int_y < 0):
    int_y = int_y * (-1)

if (int_x >= 0):

else:
```

The next step is more substantive: “If x is positive, start with 0 and repeatedly add multiples of our divisor (y) until adding another y would result in a number greater than x. Call this number N.”

This also contains an if statement, but this time it’s telling us what to do in one of our if branches. We want to do something “until” something else, which sounds like a loop. We’ve also got a value N that’s pretty clearly going to be a variable, and since it’s “starting with 0” it’s probably going to be repeatedly added to in our loop. Let’s try to convert this into Python: (Remember that to do something “until” a condition, we do it while not that condition.)

```
int_x = int(input())
int_y = int(input())
int_N = 0
```

```
if (int_y < 0):
    int_y = int_y * (-1)

if (int_x >= 0):
    while (not (int_N + int_y > int_x)): # Would adding int_y to int_N be greater 
that int_x?
        int_N += int_y
else:
```

The next step is “If x is negative, start with 0 and repeatedly subtract multiples of our divisor (y) until we get a number less than x. Call this number N.” This is really just the negative version of what we did above, with the exception that we want to go until N is **less** than x, which is different from above, where we wanted to stop when the **next** addition would make N greater than x.

```
int_x = int(input())
int_y = int(input())
int_N = 0

if (int_y < 0):
    int_y = int_y * (-1)

if (int_x >= 0):
    while (not (int_N + int_y > int_x)): # Would adding int_y to int_N be greater 
that int_x?
        int_N += int_y
else:
    while (not (int_N < int_x)):
        int_N -= int_y
```

This is looking good! It’s starting to look like a serious program here, and it’s all from the rules we worked out earlier. Let’s do our last step now: “Calculate and print x - N. That’s the result!”

```
int_x = int(input())
int_y = int(input())
int_N = 0

if (int_y < 0):
    int_y = int_y * (-1)

if (int_x >= 0):
    while (not (int_N + int_y > int_x)): # Would adding int_y to int_N be greater 
that int_x?
        int_N += int_y
else:
    while (not (int_N < int_x)):
        int_N -= int_y

print(str(int_x - int_N))
```

And there we have it! You can run this code, and it will successfully calculate mathematical modulo. Even though the code is somewhat complicated (we've got `while` loops inside of `if` statements!) we got here through a very straightforward process - we figured out how to solve the problem, then we took our vague idea of how to solve it and refined it until it qualified as an algorithm. Then, we took our algorithm, and transformed it from words into Python.

This is how programmers approach almost any problem! It will be hard sometimes, figuring out how to solve a problem can be difficult, as can making it qualify as an algorithm, and converting it into code can be also. That said, it's much, much easier than trying to figure out how to solve it by immediately attempting to think up Python code! Over time, your mental idea of "what is an algorithm" will likely converge with "what can I do in my favorite programming language", and the steps for converting from an algorithm to code will get easier, so don't worry if that bit is tricky for now!

Example Problems:

Convert this algorithm to code: To double int_x: 1) Start with the result equal to int_x 2) Multiply the result by 2. 3) Print out the result.	
Convert this algorithm to code: To perform a login: 1) Ask the user for a password. 2) Check if that password is correct. 3) If it is, print "Access granted!" 4) If it isn't, print "Access denied."	
Convert this algorithm to code: To multiply int_x by int_y: 1) Start with the result set to 0. 2) If int_y or int_x is 0, go to step 6. 3) Add int_x to the result. 4) Subtract one from int_y. 5) Repeat steps 3 & 4 until int_y is 0. 6) Print the result.	

Convert this algorithm to code: Calculating exponents with addition: 1) Ask the user for a base. 2) Ask the user for an exponent. 3) Set a variable `N` equal to 1. 4) If exponent is equal to 0, go to step 8. 5) Set a variable called `M` equal to `N`. 6) Add `M` to `N` a number of times equal to the value of base-1. 7) Subtract 1 from the exponent. 8) Repeat steps 5-7 until the exponent is 0. 9) Print the result.	