General Instructions

- This is just a summary of the full completion and submission requirements from the course Syllabus: `https://q.utoronto.ca/courses/314442#problem-sets`.
- **Worth**: 2%. **Please read the completion and submission requirements carefully**.
- **Due**: By 21:00 on Tuesday 31 October 2023, on MarkUs. Submissions will be accepted up to one week late with a penalty of roughly −10% for each day late—see the Syllabus for full details.
- Problem sets are to be submitted individually. You are free to discuss the problems and their solutions with others, but **you must write and submit your own individual answers**. Please read the Academic Integrity section on the Syllabus for full details of exactly what is allowed and what is not.
- Submissions may be typeset or handwritten *legibly*, but must be made in *one document per question, in an accepted format, using the filenames specified on MarkUs*—**documents that cannot be displayed *directly on MarkUs* will not be marked**, even if MarkUs allows you to submit them.
- Remember that your submission must meet specific conditions to receive credit. **Please read the Syllabus** for full details of how to benefit the most from this problem set, and how to submit your work.

Problems

Challenge

"Challenge" problems require some insight or creativity to apply the course material in a different way.

1. Consider the following algorithm.

```
0. # Precondition: $x \in \mathbb{R}$, $y \in \mathbb{N}$.
1. # Postcondition: return $x^y$ (return 1 if $x = y = 0$).
2. def Pow(x, y):
3.     z = 1
4.     while y > 0:
5.         if y % 2 == 1:  z = z * x
6.         x = x * x
7.         y = y // 2
8.     return z
```

Warmup:

- Trace the function with a good set of test cases. In each case, write out the table of values of x_i, y_i, z_i for all iterations of the loop.

  HINT: notice that the control-flow is dependent only on the value of y, so you might be able to leave x and/or z "abstract" and still get a good enough understanding from a good set of ys. Such a set would include instances of "extreme" behaviour (when the `if` condition is always false, and when it is always true) and at least two more "generic" cases (where the truth of the condition varies during execution).

(a) State a *useful* Loop Invariant for algorithm Pow. (Recall that "useful" means your loop invariant can be used to prove the partial correctness of the algorithm.)

Then use your loop invariant to prove the Partial Correctness of Pow.

Reminder: Your loop invariant *LI* should **NOT** depend directly on the iteration number; it must be a statement of **only** the program variables. (The instantiation of *LI* to iteration number k, denoted LI_k or $LI(k)$, will contain k only as a subscript on the program variables.)

HINT: Remember that you can always use x_0 and y_0 to refer to the initial values of variables x, y. If you find this difficult, try to work on Question 2b first.

(b) Write a *detailed* proof that your loop invariant is correct.

Use the approach and methods described in class to write your proof. Be clear and specific about all your assumptions and about exactly what you are proving in each part of your answer,

(c) State a Loop Variant for algorithm Pow. Prove that your loop variant is correct, then use it to prove termination of the algorithm.

Use the approach and methods described in class to write your proof. Be clear and specific about all your assumptions and about exactly what you are proving in each part of your answer,

(d) Finally, bring together all these steps to prove that algorithm Pow is correct.

2. (a) Re-implement function Pow (from the previous question) by filling in the implementation of Pow1 below. The body of the helper function `r` must be recursive, and calling `r` with the appropriate initial values must then execute in the exact same way as the loop in Pow (same calculations, in the same order, with the parameters taking on the same values as the loop variables in Pow).

Although the definition of `r` is inside the body of Pow1, and thus inside the scope of x and y, do not use x and y in the body of `r`. However, your precondition for `r` should mimic your invariant for Pow, and it should refer to x and y.

```
# Precondition: $x \in \mathbb{R}$, $y \in \mathbb{N}$.
# Postcondition: return $x^y$ (return 1 if $x = y = 0$).
def Pow1(x, y):

    # Precondition: [... fill this in ...]
    # Postcondition: return $x^y$ (return 1 if $x = y = 0$).
    def r(xi, yi, zi):
        # [Fill in the body of r.]

    # [Fill in the initial arguments.]
    return r(__, __, __)
```

(b) Re-implement Pow by filling in the implementation of Pow2 and `r` below, in exactly the same way as in the previous part, except that the precondition and postcondition for `r` cannot refer to x and y and must only refer to the parameters of `r` (because in this version, `r` is defined **outside** Pow2, so `r` does not have direct access to the parameter values for Pow2).

(HINT: `r` does something relatively easy to describe for a wider variety of inputs than just the ones constrained by the precondition of the previous part: try it with a good set of test cases.)

Write a detailed proof that your `r` is correct, and that your implementation of Pow2 is correct.

```
0. # Precondition: $x \in \mathbb{R}$, $y \in \mathbb{N}$.
1. # Postcondition: return $x^y$ (return 1 if $x = y = 0$).
2. def Pow2(x, y):
3.     # [Fill in the initial arguments.]
4.     return r(__, __, __)
```

```
0. # [Add a Precondition.]
1. # [Add a Postcondition.]
2. def r(xi, yi, zi):
3.     # [Fill in the body of r.]
```

(c) Since multiplication is associative, we can implement Pow recursively in a way that accumulates the value of z in the return value "on the way back/out" of the recursion. Do that, by implementing PowR below. The implementation must be recursive, without using any loop nor defining any helper function. The recursive calls must only use the arguments $x * x$ and $y//2$.

Prove your implementation of PowR is correct.

```
0. # Precondition: $x \in \mathbb{R}$, $y \in \mathbb{N}$.
1. # Postcondition: return $x^y$ (return 1 if $x = y = 0$).
2. def PowR(x, y):
3.     # [Fill in the body of PowR.]
4.     # [The recursive call(s) must only be PowR(x * x, y // 2).]
```

3. Recall that in Python, we can use *lists of lists* to represent 2-dimensional arrays (there are also various libraries that provide more efficient handling of multidimensional arrays, but we will limit ourselves to the base language for simplicity; after all, our focus is on abstract algorithms, not on lower-level implementation details).

For example, a matrix like $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ can be stored as `A = [[1, 2, 3], [4, 5, 6]]`, using what is called "row-major order". (There is also a "column-major order" convention that would use `[[1,4], [2,5], [3,6]]` to store matrix A. But *we will use row-major order in this problem.*)

For convenience, we will abuse Python's slice notation. For each 2D array A (meaning A is a list of lists where each sub-list has the same length: $\text{len}(A[0]) = \text{len}(A[1]) = \cdots = \text{len}(A[\text{len}(A)-1])$), and each $0 \le r_0 \le r_1 \le \text{len}(A)$, $0 \le c_0 \le c_1 \le \text{len}(A[0])$, we write $A[r_0 : r_1][c_0 : c_1]$ to represent the sub-array that includes rows $r_0, \ldots, r_1 - 1$ and columns $c_0, \ldots, c_1 - 1$. For example, if $A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$, then $A[0:2][1:3] = \begin{pmatrix} 2 & 3 \\ 6 & 7 \end{pmatrix}$.
Note: this is NOT valid Python! We use it only as a notational convenience.

(a) Consider the following algorithm.

```
# Precondition: v is comparable with the elements of A;
#         A is a 2D array with each row and each column sorted.
# Postcondition: returns True if A contains v; False otherwise
def IsIn(v, A):

    # Precondition: [... fill this in ...]
    # Postcondition: returns True if A[r0:r1][c0:c1] contains v; False otherwise.
    def Helper(r0, r1, c0, c1):
        # [... code goes here ...]

    # Get the search started.
    return Helper(0, len(A), 0, len(A[0]))
```

Write a suitable precondition for Helper. You should *not* repeat the preconditions from IsIn as part of your answer (even though they will hold whenever Helper is called).

(b) Now implement Helper **recursively**, by generalizing the idea of binary search: compare v against the "middle" element of $A[r0 : r1][c0 : c1]$ (you will have to figure out how to calculate this to ensure that it does something reasonable even when there is no *exact* middle), then make appropriate recursive calls to sub-arrays.

Before you write any code, take the time to think about each of the following questions, and *write your answers formally* (you will need them later). **We really mean it!** Don't just fall back into past programming habits, and try to write code *without* thinking it through first. The goal of this problem is to get you to experience how you can use the tools we have been developing to help you *write correct code on the first try*; but you cannot achieve this without putting in some thinking time first!

- What information do you gain from knowing that one element is $< v$ (or $> v$)? (Remember that each row and each column of A is sorted.)
- How do you know that each of your recursive calls is made to a strictly smaller 2D array?
- How do you combine the results of the recursive calls to ensure that your return value is always correct?

(c) Now use this information to write a complete proof that your implementation of Helper is correct. And use this fact to conclude that IsIn is also correct.