1) Make sure you understand the code
2) You should be able to answer all the green questions
3) Create a flow chart of how the code executes
4) Review computational thinking below

```
txt = "Please input an integer: "
while True: #(How do we get out of this while loop?)
    try:    #(Why use a try block?)
        counter = int(input(txt))
    except ValueError: #(what triggers the ValueError?
        txt = "You must enter an integer: "
        continue #(Where does the execution go from here?)
    else:        #(When does the else block execute?)
        if counter <= 0:
            txt = "You must enter a positive number: "
            continue #(Where does the execution go from here?)

        # valid positive integer
        for i in range(counter):
            if i == 2:
                continue #(Where does the execution go from here?)

            print(f"The square of {i} is {i*i}")
        break            #(Which loop is the break in?)
```

Computational Thinking

1. Problem Decomposition

The overall task is:

- Ask the user for a number.
- Make sure it is a **positive integer**.
- Then print squares of numbers from 0 up to that integer, skipping one case.

The program was broken into smaller pieces:

1. **Input validation** → make sure the user gives an integer.
2. **Constraint check** → make sure it's positive.
3. **Core logic** → loop through numbers and print squares, with a skip condition.

---

2. Handling Input Robustly (Error Checking)

The try / except around int(input(txt)) is an example of **defensive thinking**:

- Anticipate user errors (typing letters instead of numbers).
- Handle them gracefully without crashing.

This is **pattern recognition** → programmers know user input is unreliable, so they generalize by wrapping input conversion in try/except.

---

3. Iterative Refinement of Prompts

Notice how txt changes depending on what went wrong:

- If input wasn't an integer → "You must enter an integer: "
- If input ≤ 0 → "You must enter a positive number: "

This reflects **abstraction and state management**: the program keeps track of what went wrong and adjusts its behavior. Instead of restarting the program, it updates the prompt message and keeps looping.

---

4. Loop Control & Algorithm Design

The outer while True with continue and break is an **algorithmic strategy**:

- Keep asking until a valid input is found.
- Exit only when conditions are satisfied.

This prevents infinite incorrect inputs from breaking the program.

---

5. Selective Iteration (For Loop with continue)

In the inner loop:

```
for i in range(counter):
   if i == 2:
      continue
   print(f"The square of {i} is {i*i}")
```

This demonstrates:

- **Iteration** → systematically process numbers 0 to counter - 1.
- **Conditional logic** → skip index 2 using continue.

This is **algorithmic thinking** → rules are applied step by step to produce predictable outcomes.

---

6. Abstraction & Generalization

Instead of writing out "print the square of 0", "print the square of 1", etc., the programmer uses:

- A **for loop** to generalize the repeated action.
- A **format string** (f"The square of {i} is {i*i}") to generalize the message.

This is **abstraction** → one piece of code handles all cases.

---

7. Putting It Together

- **Decomposition**: Break down task into input → validation → iteration → output.
- **Pattern recognition**: Anticipate invalid input, repeated checking, and repeated square calculation.
- **Abstraction**: Use loops and variables instead of hardcoding cases.
- **Algorithms**: Design a clear step-by-step process with while, try/except, and for.