#2-5: Algorithmic Thinking

CS SCHOLARS – PROGRAMMING

Learning Goals

Identify whether a problem can be solved by **following an algorithm**, **applying a pattern**, or **problem solving**

Solving Familiar Problems

Designing Algorithms

Finding the right Python syntax to write a program isn't really the hardest part of programming. Designing an algorithm to solve a problem is where the process really gets tricky.

Luckily, we don't always need to design brand new algorithms. In many situations you can either use a **pre-existing algorithm** or an **algorithmic pattern** you've used before.

Pre-Existing Algorithms

Algorithms can be represented in many forms other than code!

Sometimes an algorithm might be provided to you in the text of the problem statement itself.

For example, consider the prompt to the right. We can use this prompt to find the **input** (a number), the **output** (a Boolean, whether it's prime), and a bit of the **algorithm** (check that only 1 and the number itself are factors).

Algorithms can also be provided as flow charts or formulas.

Write a function to determine whether a given number is prime.

A number is prime if it is only divisible by two numbers – itself and 1.

No other number between 1 and the prime should be a factor.

Note that primes must be larger than 1.

Pseudocode

If we wanted to break this into abstracted algorithmic steps (also called **pseudocode**), it might look like this:

Input: int (the number, x)

Output: bool (whether it's prime)

1) Verify that x is bigger than 1
2) Check all the numbers between 1 and x (non-inclusive)
    a) Verify that the number does not evenly divide x
3) Output whether x passed all verifications

It's often easier to code an algorithm if you write out the pseudocode first!

Algorithmic Patterns

There will also be many cases where an algorithm is not directly available for a program you need to implement, but you still don't need to invent a brand-new algorithm on your own. These programs will often follow common **algorithmic patterns**.

If you can recognize when a problem is similar to a pattern you've seen before, you can make the problem-solving process much more straightforward.

Example Algorithmic Pattern

For example, consider the prompt on the right.

The prompt isn't giving us the exact way to solve the problem. But it feels similar to a problem we've seen before- it's checking for numbers that divide the given number, just like `isPrime`!

We can use the structure of `isPrime` (looping over possible factors and checking something for each) as a starting place.

Write a function that checks whether a number is **powerful**.

A positive integer x is powerful if, for every prime y that divides x, y^2 also divides x.

isPowerful

```
def isPowerful(num):
    for factor in range(2, num+1):
        if isPrime(factor) and num % factor == 0:
            # factor**2 also needs to divide num
            if num % (factor ** 2) != 0:
                return False
    return True
```

Solving New Problems

Inventing New Algorithms

There will be times when you need to write an algorithm that is unlike any problem you've worked with before. In this situation, you can't rely on a pre-built algorithm or adapt an algorithmic pattern; you need to build a new algorithm on your own.

This is one of the hardest parts of the programming process, because you can't follow instructions to get the right answer; every problem is different. This just takes practice to learn.

However, there are a few strategies you can use that might make the problem-solving process easier.

Strategy 1: Human Computer

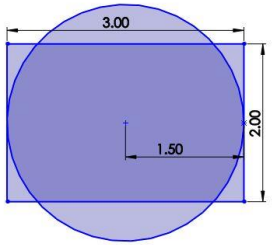
First, you can use the **human computer** strategy to look for natural approaches towards solving the problem.

Try to solve an example input to the problem yourself, as a human being. You can then extrapolate from your own approach to come up with an algorithm.

The human computer approach works best when you make your own approach as systematic and detailed as possible. Pay attention to every step you take, and make sure not to skip steps. Think about how the computer would see the problem- would the computer see it differently from you?

Human Computer Example

Example: Write the function `rectangularPegRoundHole(r, w, h)`, which returns `True` if a rectangular peg with width `w` and height `h` can pass through a round hole with radius `r`, and `False` otherwise.

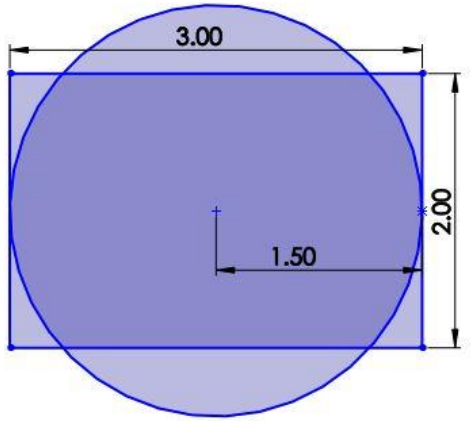


You Do: Imagine trying to fit a rectangular object into a round hole. How can you tell if the peg will be able to fit or not?

Human Computer Example

The longest part of the rectangle is its **diagonal**. If the diagonal fits, the rest will too; if the diagonal doesn't fit, then the whole thing doesn't work.

Now you just need to calculate the length of that diagonal in code and compare it to the diameter.



Human Computer Code

```
def rectangularPegRoundHole(r, w, h):
    # calculate diagonal
    diagonal = (w**2 + h**2)**0.5
    # calculate diameter
    diameter = r * 2
    # compare
    return diagonal <= diameter
```

Strategy 2: Test-Driven Design

Another strategy that can help make problem solving easier is **test-driven design**. This is an approach where you start by generating test cases instead of by jumping right into the problem.

Test-driven design can be useful because it helps you think through all the requirements of the code, which can help you notice patterns and edge cases in advance. This is better than realizing you've made a logical error only after you've written all the code.

Test-Driven Design Example

Example: Write the function `nearestBusStop(street)` that takes a non-negative integer street number and returns the nearest bus stop to the given street. Buses stop on every 8th street, including street 0, and ties go to the lower street.

You Do: what are some test cases we could use for this function that would inform us about how it works?

Test-Driven Design Example

Normal case: the nearest bus stop to 6^{th} street would be 8^{th} street

Edge case: where is there a change in results, maybe from 8^{th} street to 16^{th} street? 12^{th} street goes to 8^{th} street, but 13^{th} street goes to 16^{th} street

Special case: do we need to deal with negative or float street numbers? No, the prompt says non-negative integer.

The test functions show that this is like a **step** function. We can use conditionals and the mod operator to make this work.

Test-Driven Design Code

```
def nearestBusStop(street):
    # get distance from prev street
    belowDistance = street % 8
    if belowDistance <= 4: # edge case specifies this
        return street - belowDistance # lower street
    else:
        return street + (8 - belowDistance) # upper
    # OR
    offset = street + 3
    return offset - (offset % 8)
```

Strategy 3: Simplify and Solve

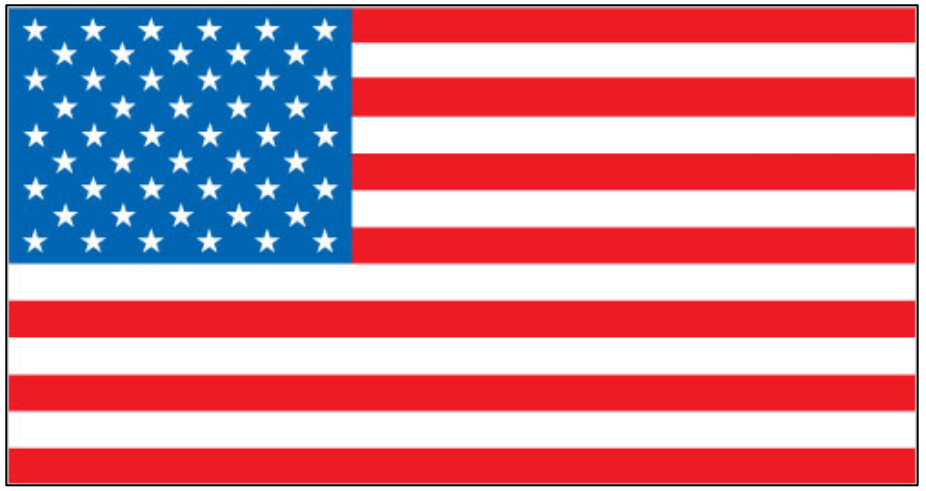
The third strategy is called **simplify and solve**. The main idea is that it's sometimes easier to solve a problem if you make that problem simpler first.

Solve the smaller problem, then add back in the more complex details once the core problem is done.

Simplify and Solve Example

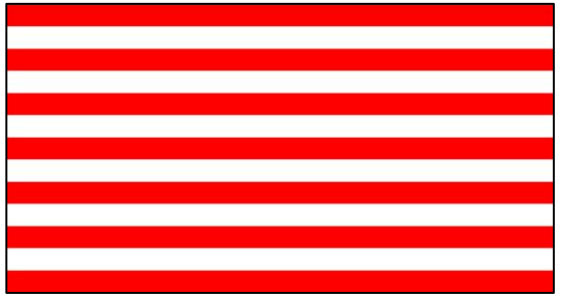
Example: we want to draw the flag of the United States using tkinter graphics.

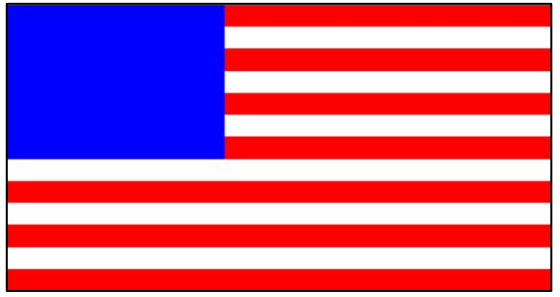
You Do: how can you break the US flag down into simpler parts?



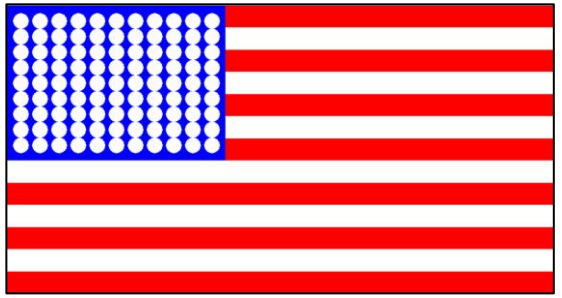
Simplify and Solve Example

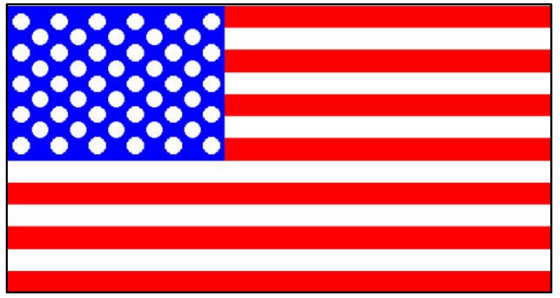
Start with just the stripes of the flag with a blue rectangle in the corner. Match proportions as you go!



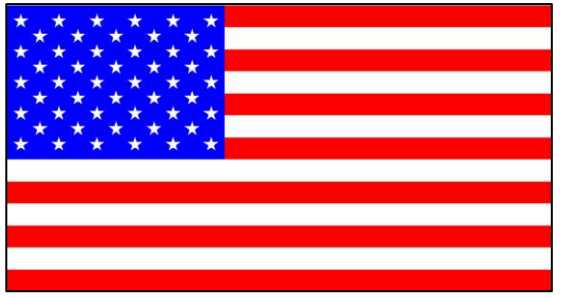


Then arrange the stars by drawing circles instead. Start with a normal grid, then alternate stars.





Finally, figure out how to draw a star instead of a circle. You can use a helper method here!



Simplify and Solve Code – Step 1

We'll need lists to draw stars properly. But we can do the rest now!

```
def drawFlag(canvas, width, height):
    numStripes = 13
    stripeHeight = height / numStripes
    for stripe in range(numStripes):
        top = stripe * stripeHeight
        if stripe % 2 == 0:
            color = "red"
        else:
            color = "white"
        canvas.create_rectangle(0, top, width, top + stripeHeight,
                                fill=color, width=0)

    squareHeight = stripeHeight * 7
    squareWidth = width * 0.4
    canvas.create_rectangle(0, 0, squareWidth, squareHeight,
                            fill="blue", width=0)
```

Simplify and Solve Code – Step 2

```
...
starRows = 9
starCols = 11
starYMargin = squareHeight / 20
starSize = (squareHeight - 2 * starYMargin) / starRows
innerXMargin = squareWidth / 60
outerXMargin = (squareWidth - starCols * starSize - innerXMargin * 10) / 2
for row in range(starRows):
    top = starYMargin + row * starSize
    for col in range(starCols):
        if (row % 2 == 0 and col % 2 == 0) or \
           (row % 2 == 1 and col % 2 == 1):
            left = outerXMargin + col * (starSize + innerXMargin)
            canvas.create_oval(left, top, left + starSize, top + starSize,
                               fill="white", width=0)
```

Learning Goals

Identify whether a problem can be solved by **following an algorithm**, **applying a pattern**, or **problem solving**