Please write a Python script to solve the following problem. It should read the input from a file input.txt, which has the same format as the example:

Today, The Historians take you up to the hot springs on Gear Island! Very suspiciously, absolutely nothing goes wrong as they begin their careful search of the vast field of helixes.

Could this finally be your chance to visit the onsen next door? Only one way to find out.

After a brief conversation with the reception staff at the onsen front desk, you discover that you don't have the right kind of money to pay the admission fee. However, before you can leave, the staff get your attention. Apparently, they've heard about how you helped at the hot springs, and they're willing to make a deal: if you can simply help them arrange their towels, they'll let you in for free!

Every towel at this onsen is marked with a pattern of colored stripes. There are only a few patterns, but for any particular pattern, the staff can get you as many towels with that pattern as you need. Each stripe can be white (w), blue (u), black (b), red (r), or green (g). So, a towel with the pattern ggr would have a green stripe, a green stripe, and then a red stripe, in that order. (You can't reverse a pattern by flipping a towel upside-down, as that would cause the onsen logo to face the wrong way.)

The Official Onsen Branding Expert has produced a list of designs - each a long sequence of stripe colors - that they would like to be able to display. You can use any towels you want, but all of the towels' stripes must exactly match the desired design. So, to display the design rgrgr, you could use two rg towels and then an r towel, an rgr towel and then a gr towel, or even a single massive rgrgr towel (assuming such towel patterns were actually available).

To start, collect together all of the available towel patterns and the list of desired designs (your puzzle input). For example:

r, wr, b, g, bwu, rb, gb, br

brwrr
bggr
gbbr
rrbgbr
ubwu
bwurrg
brgr
bbrgwb
The first line indicates the available towel patterns; in this example, the

onsen has unlimited towels with a single red stripe (r), unlimited towels with a white stripe and then a red stripe (wr), and so on.

After the blank line, the remaining lines each describe a design the onsen would like to be able to display. In this example, the first design (brwrr) indicates that the onsen would like to be able to display a black stripe, a red stripe, a white stripe, and then two red stripes, in that order.

Not all designs will be possible with the available towels. In the above example, the designs are possible or impossible as follows:

brwrr can be made with a br towel, then a wr towel, and then finally an r towel.
bggr can be made with a b towel, two g towels, and then an r towel.
gbbr can be made with a gb towel and then a br towel.
rrbgbr can be made with r, rb, g, and br.
ubwu is impossible.
bwurrg can be made with bwu, r, r, and g.
brgr can be made with br, g, and r.
bbrgwb is impossible.
In this example, 6 of the eight designs are possible with the available towel patterns.

To get into the onsen as soon as possible, consult your list of towel patterns and desired designs carefully. How many designs are possible?

Below is the Python script to solve the problem as described. This script reads the input from a file named `input.txt` and determines how many designs are possible with the given towel patterns.

Towel Design Solver

```python
from collections import Counter
from itertools import permutations

def can_form_design(towel_patterns, design):
    """
    Check if a design can be formed using the given towel patterns.
    """
    # Using Counter to count characters in the design
    design_counter = Counter(design)

    for pattern in towel_patterns:
        # Deduct the occurrences of each pattern in the design
        pattern_counter = Counter(pattern)

```

This script does the following:

1. **Reads Input**:
    - The first line specifies available towel patterns.
    - The subsequent lines (after a blank line) list the desired designs.

2. **Checks Feasibility**:
    - For each design, it tries to match it using available towel patterns.
    - Uses a `Counter` to track the required characters and reduces the counts as patterns are matched.
3. **Counts Possible Designs**:
    - For each design that can be formed using the available towel patterns, the counter for possible designs is incremented.
4. **Outputs the Result**:
    - Prints the number of designs that can be formed.

Save the script to a file, ensure the `input.txt` file is formatted correctly, and then execute the script to find the solution.

For the example:

r, wr, b, g, bwu, rb, gb, br

brwrr
bggr
gbbr
rrbgbr
ubwu
bwurrg
brgr
bbrgwb

6 of the patterns are possible but the script returns an answer of 4.

Thank you for pointing that out! It seems the current approach in the script doesn't correctly account for patterns that can appear multiple times or in various permutations in a design. I'll fix the logic to ensure all possible combinations of towel patterns are considered.

Here's the corrected script:

Corrected Python Script

python Copy code

```python
from collections import Counter

def can_form_design(towel_patterns, design):
    """
    Check if a design can be formed using the given towel patterns.
    """
    def helper(remaining_design):
```

```
        if not remaining_design:
            return True  # Base case: If no design remains, it was successfully formed

        for pattern in towel_patterns:
            if remaining_design.startswith(pattern):
                # Try using the current pattern and check the rest of the design
                if helper(remaining_design[len(pattern):]):
                    return True

        return False  # No patterns matched

    return helper(design)

def main():
    with open('input.txt', 'r') as file:
        lines = file.read().strip().split('\n')

    # First line is the towel patterns
    towel_patterns = lines[0].split(', ')

    # Designs are listed after a blank line
    designs = lines[2:]

    # Count the number of possible designs
    possible_count = 0

    for design in designs:
        if can_form_design(towel_patterns, design):
            possible_count += 1

    print(f"Number of possible designs: {possible_count}")

if __name__ == "__main__":
    main()
```

Key Updates

1. **Recursive Helper Function**:
   - The `helper` function recursively checks if the design can be built by using one pattern at a time and reducing the remaining design.
2. **Startswith Matching**:
   - This ensures the function can build the design from the start, considering the sequence of patterns.
3. **Base Case**:
   - If the design is empty, it means the design has been completely matched.

Why This Fix Works

This approach ensures all patterns are tried in every possible sequence, accurately determining whether the design can be constructed.

Example Run for the Provided Input

Using this script with your example input:

```css
r, wr, b, g, bwu, rb, gb, br

brwrr
bggr
gbbr
rrbgbr
ubwu
bwurrg
brgr
bbrgwb
```

Expected Output:

```javascript
Number of possible designs: 6
```

This should now match the correct answer. Let me know if you need further clarifications!

The script is now very slow on the actual input, which is much longer than the example.

For larger inputs, the recursive solution may indeed become inefficient because it explores too many possibilities. Let's optimize the solution using **dynamic programming (DP)** to reduce redundant computations and improve efficiency.

Optimized Solution with DP

We'll use a bottom-up approach, where we build a `dp` array to check if each prefix of the design can be formed using the given towel patterns.

Here is the updated and optimized Python script:

```python
def can_form_design(towel_patterns, design):
    """
    Check if a design can be formed using the given towel patterns using DP.
    """
```

```
    n = len(design)
    dp = [False] * (n + 1)  # dp[i] indicates if the first i characters of the design
can be formed
    dp[0] = True  # Base case: empty string can always be formed

    for i in range(1, n + 1):
        for pattern in towel_patterns:
            if i >= len(pattern) and design[i - len(pattern):i] == pattern:
                if dp[i - len(pattern)]:
                    dp[i] = True
                    break  # No need to check further if we've already formed the
prefix

    return dp[n]

def main():
    with open('input.txt', 'r') as file:
        lines = file.read().strip().split('\n')

    # First line is the towel patterns
    towel_patterns = lines[0].split(', ')

    # Designs are listed after a blank line
    designs = lines[2:]

    # Count the number of possible designs
    possible_count = 0

    for design in designs:
        if can_form_design(towel_patterns, design):
            possible_count += 1

    print(f"Number of possible designs: {possible_count}")

if __name__ == "__main__":
    main()
```

How It Works

1. **Dynamic Programming Array**:
   - `dp[i]` stores whether the first `i` characters of the design can be formed using the given towel patterns.
2. **Pattern Matching**:
   - For each position `i` in the design, we check all towel patterns to see if they can form a substring ending at `i`.
3. **Efficient Termination**:
   - As soon as we find a pattern that matches and forms a valid prefix, we break out of the loop to save time.

Advantages

- **Reduced Redundancy**: The DP table ensures each prefix of the design is processed only once.
- **Scalability**: This approach handles larger inputs efficiently.

Complexity

- Let `m` be the length of the longest design, `k` be the number of patterns, and `p` be the average length of a pattern.
- Time complexity: $O(m \cdot k \cdot p)$
- Space complexity: $O(m)$

Expected Output for Your Example

When run with the given example input, the output remains:

```javascript
Number of possible designs: 6
```

Suitability for Larger Inputs

This optimized approach should handle larger inputs gracefully. Let me know if you'd like further refinements or testing!

Please update the script for part 2 of the problem:

The staff don't really like some of the towel arrangements you came up with. To avoid an endless cycle of towel rearrangement, maybe you should just give them every possible option.

Here are all of the different ways the above example's designs can be made:

brwrr can be made in two different ways: b, r, wr, r or br, wr, r.

bggr can only be made with b, g, g, and r.

gbbr can be made 4 different ways:

g, b, b, r
g, b, br
gb, b, r
gb, br
rrbgbr can be made 6 different ways:

r, r, b, g, b, r

r, r, b, g, br
r, r, b, gb, r
r, rb, g, b, r
r, rb, g, br
r, rb, gb, r
bwurrg can only be made with bwu, r, r, and g.

brgr can be made in two different ways: b, r, g, r or br, g, r.

ubwu and bbrgwb are still impossible.

Adding up all of the ways the towels in this example could be arranged into the desired designs yields 16 (2 + 1 + 4 + 6 + 1 + 2).

They'll let you into the onsen as soon as you have the list. What do you get if you add up the number of different ways you could make each design?

To solve **Part 2**, we need to compute **all possible ways** to arrange the towels into each design and sum them up. This can be done using dynamic programming to count the ways, rather than just checking if a design is possible.

Here's the updated script for **Part 2**:

python Copy code

```python
def count_ways_to_form_design(towel_patterns, design):
    """
    Count the number of ways a design can be formed using the given towel patterns.
    """
    n = len(design)
    dp = [0] * (n + 1)  # dp[i] stores the number of ways to form the first i characters of the design
    dp[0] = 1  # Base case: one way to form an empty design (use no towels)

    for i in range(1, n + 1):
        for pattern in towel_patterns:
            if i >= len(pattern) and design[i - len(pattern):i] == pattern:
                dp[i] += dp[i - len(pattern)]

    return dp[n]

def main():
    with open('input.txt', 'r') as file:
        lines = file.read().strip().split('\n')

    # First line is the towel patterns
    towel_patterns = lines[0].split(', ')

    # Designs are listed after a blank line
    designs = lines[2:]
```

```
    # Sum the number of ways for all designs
    total_ways = 0

    for design in designs:
        total_ways += count_ways_to_form_design(towel_patterns, design)

    print(f"Total number of ways: {total_ways}")

if __name__ == "__main__":
    main()
```

How It Works

1. **Dynamic Programming Array**:
   - `dp[i]` stores the number of ways to form the first `i` characters of the design using the given towel patterns.
2. **Pattern Matching**:
   - For each position `i` in the design, check all patterns to see if they can form a substring ending at `i`.
   - If a pattern matches, add the number of ways to form the prefix before the pattern (`dp[i - len(pattern)]`) to `dp[i]`.
3. **Output the Total Count**:
   - For each design, compute the total number of ways it can be formed, and sum up the results.

Complexity

- Let `m` be the length of the longest design, `k` be the number of patterns, and `p` be the average length of a pattern.
- Time complexity: $O(m \cdot k \cdot p)$ for each design.
- Space complexity: $O(m)$.

Example Input and Output

For the example:

Input:

```
r, wr, b, g, bwu, rb, gb, br

brwrr
bggr
gbbr
rrbgbr
ubwu
bwurrg
```

```
brgr
bbrgwb
```

Output:

```mathematica
Total number of ways: 16
```

Key Cases

- Designs that are **impossible** (like `ubwu`) contribute `0` ways.
- For designs that are possible, all valid arrangements are counted.

This approach is efficient and scales well for larger inputs. Let me know if you'd like to test it further or need assistance!