```
4    def test_jump_db1():
5        print("Expect shift of 1, the result: ", test_debug1.caesar("z", "a"))
6        print("Expect shift of 1, the result: ", test_debug1.caesar("a", "b"))
7        print("Expect shift of 2, the result: ", test_debug1.caesar("z", "b"))
8        print("Expect shift of 2, the result: ", test_debug1.caesar("a", "c"))
```

```
14   if __name__ == '__main__':
15       test_jump_db1()
```

```
Expect shift of 1, the result:  -25
Expect shift of 1, the result:  1
Expect shift of 2, the result:  -24
Expect shift of 2, the result:  2
```

When these tests are run its clear that the valid cypher checker does not account for wrap arounds. To confirm this suspicion we can check the debugger while running the cipher function using the following params: “za”,”ab”. We see 2 flawed yet correct examples when the code is run because there is no wrap around in those 2 tests.

```
7        print("Expect shift of 1, the result: ", test_debug1.caesar("za", "ab"))
```

```
Expect shift of 1, the result:  -1
```

As seen above this confirms my suspicions.

```
81       shift = ord(codeword[0]) - ord(original[0])   shift: -25
82
83       for idx in range(len(codeword)):   idx: 1
84           if ord(codeword[idx]) - ord(original[idx]) != shift:
85               return -1
86
87       return shift
88
```

```
codeword = {str} 'ab'
idx = {int} 1
original = {str} 'za'
shift = {int} -25
```

Its clear that as the first shift is a shift of 1 but the program registers it as -25 due to it not taking wrap around into consideration. The following code fixes this issue.

```
        shift = ord(codeword[0]) - ord(original[0])
        if shift < 0:
            shift += 26
        for idx in range(len(codeword)):
            if ord(codeword[idx]) - ord(original[idx]) < 0:
                temp = ord(codeword[idx]) - ord(original[idx]) + 26
                if temp != shift:
                    return -1
            elif ord(codeword[idx]) - ord(original[idx]) != shift:
                return -1

        return shift
```

```
    print("Expect shift of 1, the result: ", test_debug1.caesar("z", "a"))
    print("Expect shift of 1, the result: ", test_debug1.caesar("a", "b"))
    print("Expect shift of 1, the result: ", test_debug1.caesar("za", "ab"))
    print("Expect shift of 2, the result: ", test_debug1.caesar("z", "b"))
    print("Expect shift of 2, the result: ", test_debug1.caesar("a", "c"))
```

As we can see the wrap-around test is passed.

```
Expect shift of 1, the result:  1
Expect shift of 1, the result:  1
Expect shift of 1, the result:  1
Expect shift of 2, the result:  2
Expect shift of 2, the result:  2
```

```
        print("Expect shift of 25, the result: ", test_debug1.caesar("zack",
                                                                      "yzbj"))
        print("Expect shift of 5, the result: ", test_debug1.caesar("very",
                                                                     "ajwd"))
```

```
Expect shift of 25, the result:  25
Expect shift of 5, the result:  5
```

For the first test we notice that the function skips the last letter in each string.

```
61          best_length = 0  best_length: 0
62          # for all possible string1 start points
63          for idx1 in range(len(string1)-1):  idx1: 1
64              # for all possible string2 start points
65              for idx2 in range(len(string2)-1):  idx2: 7
66                  # check if these characters match
67                  if string1[idx1] == string2[idx2]:
68                      this_match_count = 1
69                      # see how long the match continues
70                      while string1[idx1 + this_match_count] == \
71                              string2[idx2 + this_match_count]:
72                          this_match_count += 1
73
74                      # compare to best so far
75                      if this_match_count > best_length:
76                          best_length = this_match_count
77
78          # now return the result
79          return best_length
```

To fix this we can remove -1. This is a logical error where the developer thought the range function is inclusive.

Besides that, in the above screenshot we also notice that the arguments in the while loop can go out of range as they have no loop control condition. To fix this we can introduce a while loop that ensures the check stays within bounds of the shortest string and then another conditional can be used to break out of the loop when the matching pattern ends.

```
60          best_length = 0
61          # for all possible string1 start points
62          for idx1 in range(len(string1)):
63              # for all possible string2 start points
64              for idx2 in range(len(string2)):
65                  # check if these characters match
66                  if string1[idx1] == string2[idx2]:
67                      this_match_count = 1
68                      # see how long the match continues
69                      while not (idx1 + this_match_count > len(string1)-1 or idx2
70                                 + this_match_count > len(string2)-1):
71                          if string1[idx1 + this_match_count] == string2[idx2
72                                                                         + this_match_count]:
73                              this_match_count += 1
74                          else:
75                              break
76
77                      # compare to best so far
78                      if this_match_count > best_length:
79                          best_length = this_match_count
80
81          # now return the result
82          return best_length
```

To break out of the loop we can simply use an else statement and break since no match is the only other possibility.

```
23  def test_db2():
24      print("expected 1, got: ", test_debug2.match("a", "a"))
25      print("expected 3, got: ", test_debug2.match("established", "ballistic"))
26      print("expected 3, got: ", test_debug2.match("abc", "abcabc"))
27      print("expected 1, got: ", test_debug2.match("abc", "cba"))
```

```
expected 1, got:  1
expected 3, got:  3
expected 3, got:  3
expected 1, got:  1
```

And as we can see the following fixes allow the code to pass our texts.