1. You are given two strings s and t. String t is generated by randomly shuffling string s and then adding one more letter at a random position. Return the letter that was added to t.

Examples:
find_the_difference("abcd", "abcde") → "e"
find_the_difference("", "y") → "y"
find_the_difference("ae", "aea") → "a"

Ans:

In [1]:
```
def find_the_difference(s,t):
    t1 = list(t)
    for i in s:
        if i in t1:
            t1.pop(t1.index(i))
    print(f'find_the_difference({s},{t})→ {t1}')
find_the_difference("abcd", "abcde")
find_the_difference("", "y")
find_the_difference("ae", "aea")
```

```
find_the_difference(abcd,abcde)→ ['e']
find_the_difference(,y)→ ['y']
find_the_difference(ae,aea)→ ['a']
```

---

2. Given a function that accepts unlimited arguments, check and count how many data types are in those arguments. Finally return the total in a list.

List order is: [int, str, bool, list, tuple, dictionary]
Examples:
count_datatypes(1, 45, "Hi", False) → [2, 1, 1, 0, 0, 0]
count_datatypes([10, 20], ("t", "Ok"), 2, 3, 1) → [3, 0, 0, 1, 1, 0]
count_datatypes("Hello", "Bye", True, True, False, {"1": "One", "2": "Two"}, [1, 3], {"Brayan": 18}, 25, 23) → [2, 2, 3, 1, 0, 2]
count_datatypes(4, 21, ("ES", "EN"), ("a", "b"), False, [1, 2, 3], [4, 5, 6]) → [2, 0, 1, 2, 2, 0]

Ans:

In [29]:
```
from collections import OrderedDict
def count_datatypes(*args):
    output = OrderedDict({'int':0,'str':0,'bool':0,'list':0,'tuple':0,'dict':0})
    for i in args:
        output[type(i).__name__] += 1
    print(f'count_datatypes{args}  →  {list(output.values())}')

count_datatypes(1, 45, "Hi", False)
count_datatypes([10, 20], ("t", "Ok"), 2, 3, 1)
count_datatypes("Hello", "Bye", True, True, False, {"1": "One", "2": "Two"}, [1, 3], {"Brayan": 18}, 25, 23)
count_datatypes(4, 21, ("ES", "EN"), ("a", "b"), False, [1, 2, 3], [4, 5, 6])
```

```
count_datatypes(1, 45, 'Hi', False)  →  [2, 1, 1, 0, 0, 0]
count_datatypes([10, 20], ('t', 'Ok'), 2, 3, 1)  →  [3, 0, 0, 1, 1, 0]
count_datatypes('Hello', 'Bye', True, True, False, {'1': 'One', '2': 'Two'}, [1, 3], {'Brayan': 18}, 25, 23)  →  [2, 2, 3, 1,
0, 2]
count_datatypes(4, 21, ('ES', 'EN'), ('a', 'b'), False, [1, 2, 3], [4, 5, 6])  →  [2, 0, 1, 2, 2, 0]
```

3. A Fibonacci string is a precedence of the Fibonacci series. It works with any two characters of the English alphabet (as opposed to the numbers 0 and 1 in the Fibonacci series) as the initial items and concatenates them together as it progresses in a similar fashion as the Fibonacci series.

Examples:
fib_str(3, ["j", "h"]) ⟶ "j, h, hj"
fib_str(5, ["e", "a"]) ⟶ "e, a, ae, aea, aeaae"
fib_str(6, ["n", "k"]) ⟶ "n, k, kn, knk, knkkn, knkknknk"

Ans:

In [15]:
```
def fib_str(n,inlist):
    char1 = inlist[-1]
    char2 = inlist[-2]
    output =", ".join(inlist)
    string = []
    if n>2:
        for i in range(2,n):
            string = char1+char2
            output += ", "+string
            char1 = char2
            char2 = string
    print(f'fib_str{n, inlist} ⟶ {output}')
fib_str(3,["j", "h"])
fib_str(5, ["e", "a"])
fib_str(6, ["n", "k"])
```

```
fib_str(3, ['j', 'h']) ⟶ j, h, hj
fib_str(5, ['e', 'a']) ⟶ e, a, ae, eae, aeeae
fib_str(6, ['n', 'k']) ⟶ n, k, kn, nkn, knnkn, nknknnkn
```

4. Given an integer between 0 and 26, make a variable (self.answer). That variable would be assigned to a string in this format:

"nines:your answer, threes:your answer, ones:your answer"
You need to find out how many ones, threes, and nines it would at least take for the number of each to add up to the given integer when multiplied by one, three or nine (depends).
Examples:
ones_threes_nines(10) ⟶ "nines:1, threes:0, ones:1"
ones_threes_nines(15) ⟶ "nines:1, threes:2, ones:0"
ones_threes_nines(22) ⟶ "nines:2, threes:1, ones:1"

Ans:

In [1]:

```
def ones_threes_nines(n):
    n_copy = n
    answer = {'nines':0,'threes':0,'ones':0}
    if n > 0 and n < 26:
        while n_copy != 0:
            if n_copy >= 9:
                n_copy -= 9
                answer['nines'] += 1
            elif n_copy >= 3:
                n_copy -= 3
                answer['threes'] += 1
            elif n_copy >= 1:
                n_copy -=1
                answer['ones'] +=1
    print(f"ones_threes_nines({n}) → \"nines:{answer['nines']}, threes:{answer['threes']}, ones:{answer['ones']}\"")
ones_threes_nines(10)
ones_threes_nines(15)
ones_threes_nines(22)
```

```
ones_threes_nines(10) → "nines:1, threes:0, ones:1"
ones_threes_nines(15) → "nines:1, threes:2, ones:0"
ones_threes_nines(22) → "nines:2, threes:1, ones:1"
```

5. The Fibonacci sequence is a classic use case for recursive functions since the value of the sequence at a given index is dependent on the sum of the last two values. However, the recursion tree created by solving the Fibonacci sequence recursively can grow quite fast. Therefore it can be important to think about the implications of running a function recursively. Depending on the size of n needed and the capabilities of the system in question you might want to take a different approach.

Write a non-recursive function that takes an integer n and returns the Fibonacci sequence's value at index n.

Examples:

fib(6) → 8

0 + 1 = 1, 1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, 3 + 5 = 8

fib(1) → 1

fib(2) → 1

Ans:

In [3]:

```
def fib(n):
    x=0
    output = 0
    y = 1
    if n == 0:
        output = 0
    elif n == 1:
        output = 1
    else:
        for i in range(2, n+1):
            output = x+y
            x = y
            y = output
    print(f'fib({n}) → {output}')
fib(6)
fib(1)
fib(2)
```

```
fib(6) → 8
fib(1) → 1
fib(2) → 1
```