Here are the solutions for all the problems:

**Problem 1: Sudoku Class**

class Sudoku:

def \_\_init\_\_(self, s):

self.board = [list(map(int, s[i:i + 9])) for i in range(0, len(s), 9)]

def get\_row(self, n):

return self.board[n]

def get\_col(self, n):

return [self.board[i][n] for i in range(9)]

def get\_sqr(self, n, m=None):

if m is None:

row, col = (n // 3) \* 3, (n % 3) \* 3

return [self.board[i][j] for i in range(row, row + 3) for j in range(col, col + 3)]

else:

row, col = (n // 3) \* 3, (m // 3) \* 3

return [self.board[i][j] for i in range(row, row + 3) for j in range(col, col + 3)]

# Example usage

game = Sudoku("417950030000000700060007000050009106800600000000003400900005000000430000200701580")

print(game.get\_row(0)) # [4, 1, 7, 9, 5, 0, 0, 3, 0]

print(game.get\_col(8)) # [0, 0, 0, 6, 0, 0, 0, 0, 0]

print(game.get\_sqr(1)) # [9, 5, 0, 0, 0, 0, 0, 0, 7]

print(game.get\_sqr(1, 8)) # [0, 3, 0, 7, 0, 0, 0, 0, 0]

print(game.get\_sqr(8, 3)) # [0, 0, 5, 4, 3, 0, 7, 0, 1]

**Problem 2: Add Two Numbers Using Linked List**

class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

def add\_data(self, values):

current = self

for value in values:

current.next = ListNode(value)

current = current.next

def get\_data(self):

data = []

current = self

while current:

data.append(current.val)

current = current.next

return data

def add\_two\_numbers(l1, l2):

dummy\_head = ListNode()

current = dummy\_head

carry = 0

while l1 or l2 or carry:

val1 = l1.val if l1 else 0

val2 = l2.val if l2 else 0

total = val1 + val2 + carry

carry = total // 10

current.next = ListNode(total % 10)

current = current.next

if l1:

l1 = l1.next

if l2:

l2 = l2.next

return dummy\_head.next

# Example usage

lt1 = ListNode(2)

lt1.add\_data([4, 3])

lt2 = ListNode(5)

lt2.add\_data([6, 4])

print(add\_two\_numbers(lt1, lt2).get\_data()) # [7, 0, 8]

**Problem 3: CoffeeShop Class**

class CoffeeShop:

def \_\_init\_\_(self, name, menu):

self.name = name

self.menu = menu

self.orders = []

def add\_order(self, item):

if any(i['name'] == item for i in self.menu):

self.orders.append(item)

return "Order added!"

else:

return "This item is currently unavailable!"

def fulfill\_order(self):

if self.orders:

item = self.orders.pop(0)

return f"The {item} is ready!"

else:

return "All orders have been fulfilled!"

def list\_orders(self):

return self.orders

def due\_amount(self):

total = 0

for order in self.orders:

for item in self.menu:

if item['name'] == order:

total += item['price']

return total

def cheapest\_item(self):

return min(self.menu, key=lambda x: x['price'])['name']

def drinks\_only(self):

return [item['name'] for item in self.menu if item['type'] == 'drink']

def food\_only(self):

return [item['name'] for item in self.menu if item['type'] == 'food']

# Example usage

menu = [

{"name": "iced tea", "type": "drink", "price": 1.5},

{"name": "cinnamon roll", "type": "food", "price": 2.0},

{"name": "iced coffee", "type": "drink", "price": 2.5}

]

tcs = CoffeeShop("Tesha's Coffee Shop", menu)

print(tcs.add\_order("cinnamon roll")) # "Order added!"

print(tcs.due\_amount()) # 4.5

**Problem 4: Loneliest Number**

def is\_prime(n):

if n <= 1:

return False

for i in range(2, int(n \*\* 0.5) + 1):

if n % i == 0:

return False

return True

def closest\_prime(n):

lower, upper = n - 1, n + 1

while True:

if is\_prime(lower):

return lower

if is\_prime(upper):

return upper

lower -= 1

upper += 1

def loneliest\_number(lo, hi):

max\_distance = -1

result = {}

for num in range(lo, hi + 1):

closest = closest\_prime(num)

distance = abs(num - closest)

if distance > max\_distance:

max\_distance = distance

result = {'number': num, 'distance': distance, 'closest': closest}

return result

# Example usage

print(loneliest\_number(0, 22)) # {'number': 0, 'distance': 2, 'closest': 2}

**Problem 5: Selfie Class**

class Selfie:

def \_\_init\_\_(self):

self.state\_history = []

self.save\_state()

def save\_state(self):

self.state\_history.append(self.\_\_dict\_\_.copy())

def recover\_state(self, index):

if 0 <= index < len(self.state\_history):

state = Selfie()

state.\_\_dict\_\_ = self.state\_history[index]

state.state\_history = self.state\_history[:index + 1]

return state

return self

def n\_states(self):

return len(self.state\_history)

# Example usage

p = Selfie()

p.x = 2

p.save\_state()

p.x = 5

p = p.recover\_state(0)

print(p.x) # 2