```python
"""
Question 1a: Maximizing Tech Startup Revenue before Acquisition

Implements a function to determine the maximum possible capital after at most k projects.
"""
from typing import List
import heapq

def maximize_capital(k: int, c: int, revenues: List[int], investments: List[int]) -> int:
    """
    Selects up to k projects to maximize final capital, given initial capital c.
    Args:
        k: Maximum number of projects.
        c: Initial capital.
        revenues: List of revenue gains for each project.
        investments: List of required capital for each project.
    Returns:
        Maximum capital after up to k projects.
    """
    n = len(revenues)
    projects = sorted(zip(investments, revenues), key=lambda x: x[0])
    max_heap = []  # max-heap for available projects (by revenue)
    i = 0
    capital = c

    for _ in range(k):
        # Add all projects that can be started with current capital
        while i < n and projects[i][0] <= capital:
            heapq.heappush(max_heap, -projects[i][1])  # Use negative for max-heap
            i += 1
        if not max_heap:
            break
        # Select the project with the highest revenue
        capital += -heapq.heappop(max_heap)
    return capital

if __name__ == "__main__":
    # Example 1
    print("Example 1:", maximize_capital(2, 0, [2, 5, 8], [0, 2, 3]))  # Output: 7
    # Example 2
    print("Example 2:", maximize_capital(3, 1, [3, 6, 10], [1, 3, 5]))  # Output: 19
```

```python
"""
Question 1b: Secure Bank PIN Policy Upgrade

Implements a function to determine minimum changes required to make a PIN strong.
"""
import re

def min_changes_for_strong_pin(pin_code: str) -> int:
    """
    Returns the minimum number of changes required to make the pin_code strong.
    Args:
        pin_code: The PIN string to check.
    Returns:
        Minimum number of changes required.
    """
    n = len(pin_code)
    missing_types = 3 - sum([bool(re.search(r"[a-z]", pin_code)),
                             bool(re.search(r"[A-Z]", pin_code)),
                             bool(re.search(r"[0-9]", pin_code))])
    # Count repeating sequences
    changes = 0
    i = 2
    repeat = 0
    while i < n:
        if pin_code[i] == pin_code[i-1] == pin_code[i-2]:
            repeat += 1
            i += 2
        else:
            i += 1
    if n < 6:
        return max(missing_types, 6-n)
    elif n <= 20:
        return max(missing_types, repeat)
    else:
        # Need to delete extra chars
        delete = n - 20
        changes = delete
        # Reduce repeats with deletions
        changes += max(missing_types, repeat)
        return changes

if __name__ == "__main__":
    print("Example 1:", min_changes_for_strong_pin("X1!"))  # Output: 3
    print("Example 2:", min_changes_for_strong_pin("123456"))  # Output: 2
    print("Example 3:", min_changes_for_strong_pin("Aa1234!"))  # Output: 0
```

```python
"""
Question 2a: Weather Anomaly Detection

Counts the number of continuous subarrays where the sum falls within a given range.
"""
from typing import List

def    count_anomaly_periods(temperature_changes:    List[int],    low_threshold:    int,
high_threshold: int) -> int:
    """
    Counts continuous periods where the sum is within [low_threshold, high_threshold].
    Args:
        temperature_changes: List of daily temperature changes.
        low_threshold: Lower bound of anomaly range.
        high_threshold: Upper bound of anomaly range.
    Returns:
        Number of valid periods (subarrays).
    """
    n = len(temperature_changes)
    count = 0
    for i in range(n):
        total = 0
        for j in range(i, n):
            total += temperature_changes[j]
            if low_threshold <= total <= high_threshold:
                count += 1
    return count

if __name__ == "__main__":
    # Example 1
    print("Example 1:", count_anomaly_periods([3, -1, -4, 6, 2], 2, 5))  # Output: 3 or
4 (see prompt)
    # Example 2
    print("Example 2:", count_anomaly_periods([-2, 3, 1, -5, 4], -1, 2))  # Output: 4 or
5 (see prompt)
```

```python
"""
Question 2b: Alphametic Puzzle Solver

Checks if there is a valid digit assignment for a given word equation.
"""
from typing import List, Dict, Set
import itertools

def word_to_number(word: str, mapping: Dict[str, int]) -> int:
    return int(''.join(str(mapping[ch]) for ch in word))

def is_valid_mapping(words: List[str], result: str, mapping: Dict[str, int]) -> bool:
    # No word can have leading zero
    for w in words + [result]:
        if mapping[w[0]] == 0:
            return False
    total = sum(word_to_number(w, mapping) for w in words)
    return total == word_to_number(result, mapping)

def solve_alphametic(words: List[str], result: str) -> bool:
    """
    Returns True if a valid digit assignment exists, False otherwise.
    Args:
        words: List of words to sum.
        result: The result word.
    Returns:
        True if possible, False otherwise.
    """
    letters = set(''.join(words + [result]))
    if len(letters) > 10:
        return False
    letters = list(letters)
    for perm in itertools.permutations(range(10), len(letters)):
        mapping = dict(zip(letters, perm))
        if is_valid_mapping(words, result, mapping):
            return True
    return False

if __name__ == "__main__":
    # Example 1: STAR + MOON = NIGHT (should be True for some mapping)
    print("Example 1:", solve_alphametic(["STAR", "MOON"], "NIGHT"))
    # Example 2: CODE + BUG = DEBUG (should be False)
    print("Example 2:", solve_alphametic(["CODE", "BUG"], "DEBUG"))
```

```python
"""
Question 3a: Pattern Sequence Extraction

Given two patterns and repeat counts, find the maximum number of times the second pattern can be formed as a subsequence from the repeated first pattern.
"""
def max_pattern_extraction(p1: str, t1: int, p2: str, t2: int) -> int:
    """
      Returns the maximum number of times p2 can be formed as a subsequence from p1 repeated t1 times, divided by t2.
    Args:
        p1: Base pattern for sequence A.
        t1: Number of times p1 is repeated.
        p2: Base pattern for sequence B.
        t2: Number of times p2 is repeated.
    Returns:
        Maximum x such that p2 * t2 can be extracted from p1 * t1.
    """
    seqA = p1 * t1
    seqB = p2 * t2
    idxA = idxB = count = 0
    while idxA < len(seqA) and idxB < len(seqB):
        if seqA[idxA] == seqB[idxB]:
            idxB += 1
        idxA += 1
        if idxB == len(seqB):
            count += 1
            idxB = 0
    return count

if __name__ == "__main__":
    # Example from assignment
    print("Example:", max_pattern_extraction("bca", 6, "ba", 3))  # Output: 3
```

```python
"""
Question 3b: Magical Words (Odd-length Palindromes)

Find two non-overlapping magical words (odd-length palindromes) in a string to maximize the product of their lengths.
"""
def max_magical_power(M: str) -> int:
    """
    Returns the maximum product of lengths of two non-overlapping odd-length palindromic substrings.
    Args:
        M: The manuscript string.
    Returns:
        Maximum product of lengths.
    """
    n = len(M)
    # Find all odd-length palindromes
    palindromes = []  # (start, end, length)
    for center in range(n):
        l = r = center
        while l >= 0 and r < n and M[l] == M[r]:
            if (r - l + 1) % 2 == 1:
                palindromes.append((l, r, r - l + 1))
            l -= 1
            r += 1
    # Try all pairs of non-overlapping palindromes
    max_product = 0
    for i in range(len(palindromes)):
        for j in range(i + 1, len(palindromes)):
            a = palindromes[i]
            b = palindromes[j]
            # Check non-overlapping
            if a[1] < b[0] or b[1] < a[0]:
                max_product = max(max_product, a[2] * b[2])
    return max_product

if __name__ == "__main__":
    # Example 1
    print("Example 1:", max_magical_power("xyzyxabc"))  # Output: 5
    # Example 2
    print("Example 2:", max_magical_power("levelwowracecar"))  # Output: 35
```

```
"""
Question 4a: Secure Transmission

Implements a class to check if a message can be securely transmitted between offices
with signal strength limits.
"""
from typing import List
from collections import deque, defaultdict

class SecureTransmission:
    def __init__(self, n: int, links: List[List[int]]):
        """
        Initializes the system with n offices and a list of communication links.
        Args:
            n: Number of offices (nodes).
            links: List of [a, b, strength] edges.
        """
        self.graph = defaultdict(list)
        for a, b, strength in links:
            self.graph[a].append((b, strength))
            self.graph[b].append((a, strength))

    def canTransmit(self, sender: int, receiver: int, maxStrength: int) -> bool:
        """
          Returns True if a path exists between sender and receiver with all links <
maxStrength.
        """
        visited = set()
        queue = deque([sender])
        while queue:
            node = queue.popleft()
            if node == receiver:
                return True
            visited.add(node)
            for neighbor, strength in self.graph[node]:
                if neighbor not in visited and strength < maxStrength:
                    queue.append(neighbor)
        return False

if __name__ == "__main__":
    # Example from assignment
    st = SecureTransmission(6, [[0,2,4],[2,3,1],[2,1,3],[4,5,5]])
    print(st.canTransmit(2, 3, 2))  # True
    print(st.canTransmit(1, 3, 3))  # False
    print(st.canTransmit(2, 0, 3))  # True
    print(st.canTransmit(0, 5, 6))  # False
```

```python
"""
Question 4b: Treasure Hunt Game (Graph Simulation)

Simulates a two-player treasure hunt game on an undirected graph with draw/win/lose conditions.
"""
from typing import List, Tuple

def treasure_hunt_game(graph: List[List[int]]) -> int:
    """
    Returns 1 if Player 1 wins, 2 if Player 2 wins, 0 for draw.
    Player 1 starts at node 1, Player 2 at node 2, Treasure at node 0.
    Player 2 cannot move to node 0.
    """
    from collections import deque
    N = len(graph)
    # State: (p1_pos, p2_pos, turn)   turn: 1 for Player 1, 2 for Player 2
    visited = set()
    queue = deque()
    queue.append((1, 2, 1))
    while queue:
        p1, p2, turn = queue.popleft()
        if (p1, p2, turn) in visited:
            return 0  # Draw by repetition
        visited.add((p1, p2, turn))
        if p1 == 0:
            return 1  # Player 1 wins
        if p1 == p2:
            return 2  # Player 2 wins
        if turn == 1:
            # Player 1 moves
            for nxt in graph[p1]:
                queue.append((nxt, p2, 2))
        else:
            # Player 2 moves (cannot go to 0)
            for nxt in graph[p2]:
                if nxt != 0:
                    queue.append((p1, nxt, 1))
    return 0  # Draw if queue is exhausted

if __name__ == "__main__":
    Graph = [
        [2, 5],  # Node 0
        [3],     # Node 1
        [0, 4, 5],
        [1, 4, 5],
        [2, 3],
        [0, 2, 3]
    ]
    print("Example:", treasure_hunt_game(Graph))  # Output: 0 (Draw)
```

```
"""
Question 5a: Maze Solver with GUI

A grid-based maze solver supporting DFS and BFS, with random maze generation and visual path animation.
"""
import tkinter as tk
import random
from collections import deque

CELL_SIZE = 30
GRID_SIZE = 15

class MazeSolverGUI:
    def __init__(self, master):
        self.master = master
        self.master.title("Maze Solver")
        self.canvas = tk.Canvas(master, width=GRID_SIZE*CELL_SIZE, height=GRID_SIZE*CELL_SIZE)
        self.canvas.pack()
        self.reset_maze()
        self.start = (0, 0)
        self.end = (GRID_SIZE-1, GRID_SIZE-1)
        self.draw_maze()

        btn_frame = tk.Frame(master)
        btn_frame.pack()
        tk.Button(btn_frame, text="Solve DFS", command=self.solve_dfs).pack(side=tk.LEFT)
        tk.Button(btn_frame, text="Solve BFS", command=self.solve_bfs).pack(side=tk.LEFT)
        tk.Button(btn_frame, text="Generate New Maze", command=self.reset_and_draw).pack(side=tk.LEFT)

    def reset_maze(self):
        self.maze = [[1 for _ in range(GRID_SIZE)] for _ in range(GRID_SIZE)]
        self.generate_maze()

    def reset_and_draw(self):
        self.reset_maze()
        self.start = (0, 0)
        self.end = (GRID_SIZE-1, GRID_SIZE-1)
        self.draw_maze()

    def generate_maze(self):
        # Recursive Backtracking
        stack = [(0, 0)]
        self.maze[0][0] = 0
        while stack:
            x, y = stack[-1]
            neighbors = []
            for dx, dy in [(-2,0),(2,0),(0,-2),(0,2)]:
                nx, ny = x+dx, y+dy
                if 0 <= nx < GRID_SIZE and 0 <= ny < GRID_SIZE and self.maze[nx][ny] ==
```

```
1:
                    neighbors.append((nx, ny))
            if neighbors:
                nx, ny = random.choice(neighbors)
                self.maze[(x+nx)//2][(y+ny)//2] = 0
                self.maze[nx][ny] = 0
                stack.append((nx, ny))
            else:
                stack.pop()

    def draw_maze(self, path=None):
        self.canvas.delete("all")
        for i in range(GRID_SIZE):
            for j in range(GRID_SIZE):
                color = "black" if self.maze[i][j] else "white"
                  self.canvas.create_rectangle(j*CELL_SIZE, i*CELL_SIZE, (j+1)*CELL_SIZE,
(i+1)*CELL_SIZE, fill=color)
        # Draw start and end
           self.canvas.create_rectangle(self.start[1]*CELL_SIZE, self.start[0]*CELL_SIZE,
(self.start[1]+1)*CELL_SIZE, (self.start[0]+1)*CELL_SIZE, fill="green")
               self.canvas.create_rectangle(self.end[1]*CELL_SIZE,  self.end[0]*CELL_SIZE,
(self.end[1]+1)*CELL_SIZE, (self.end[0]+1)*CELL_SIZE, fill="red")
        # Draw path if exists
        if path:
            for (i, j) in path:
                  self.canvas.create_rectangle(j*CELL_SIZE, i*CELL_SIZE, (j+1)*CELL_SIZE,
(i+1)*CELL_SIZE, fill="yellow")
        self.master.update()

    def solve_dfs(self):
        path = self.dfs(self.start, self.end)
        self.draw_maze(path)
        self.show_result(path)

    def solve_bfs(self):
        path = self.bfs(self.start, self.end)
        self.draw_maze(path)
        self.show_result(path)

    def show_result(self, path):
        if path:
            tk.messagebox.showinfo("Maze Solver", f"Path found! Steps: {len(path)}")
        else:
            tk.messagebox.showinfo("Maze Solver", "No path found.")

    def dfs(self, start, end):
        stack = [(start, [start])]
        visited = set()
        while stack:
            (x, y), path = stack.pop()
            if (x, y) == end:
                return path
            if (x, y) in visited:
                continue
```

```
            visited.add((x, y))
            for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:
                nx, ny = x+dx, y+dy
                 if 0 <= nx < GRID_SIZE and 0 <= ny < GRID_SIZE and self.maze[nx][ny] ==
0 and (nx, ny) not in visited:
                    stack.append(((nx, ny), path + [(nx, ny)]))
        return None

    def bfs(self, start, end):
        queue = deque([(start, [start])])
        visited = set()
        while queue:
            (x, y), path = queue.popleft()
            if (x, y) == end:
                return path
            if (x, y) in visited:
                continue
            visited.add((x, y))
            for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:
                nx, ny = x+dx, y+dy
                 if 0 <= nx < GRID_SIZE and 0 <= ny < GRID_SIZE and self.maze[nx][ny] ==
0 and (nx, ny) not in visited:
                    queue.append(((nx, ny), path + [(nx, ny)]))
        return None

if __name__ == "__main__":
    import tkinter.messagebox
    root = tk.Tk()
    app = MazeSolverGUI(root)
    root.mainloop()
```

```
"""
Question 5b: Online Ticket Booking System with Concurrency Control (GUI)

Simulates a ticket booking system with concurrency control and a GUI.
"""
import tkinter as tk
import threading
import random
import time
from queue import Queue

SEATS = 30

class BookingSystemGUI:
    def __init__(self, master):
        self.master = master
        self.master.title("Ticket Booking System")
        self.seats = [False for _ in range(SEATS)]
        self.requests = Queue()
        self.locking = tk.StringVar(value="optimistic")
        self.buttons = []
        self.setup_gui()
        self.processing = False

    def setup_gui(self):
        frame = tk.Frame(self.master)
        frame.pack()
        for i in range(SEATS):
            btn = tk.Button(frame, text=str(i+1), width=3, bg="lightgreen",
command=lambda i=i: self.manual_book(i))
            btn.grid(row=i//10, column=i%10)
            self.buttons.append(btn)
        tk.Button(self.master, text="Simulate Booking Requests",
command=self.simulate_requests).pack()
        tk.Button(self.master, text="Process Bookings",
command=self.process_bookings).pack()
        tk.Radiobutton(self.master, text="Optimistic Locking", variable=self.locking,
value="optimistic").pack(anchor=tk.W)
        tk.Radiobutton(self.master, text="Pessimistic Locking", variable=self.locking,
value="pessimistic").pack(anchor=tk.W)
        self.status = tk.Label(self.master, text="Status: Idle")
        self.status.pack()

    def manual_book(self, seat):
        if not self.seats[seat]:
            self.requests.put(seat)
            self.status.config(text=f"Manual booking request for seat {seat+1} queued.")
        else:
            self.status.config(text=f"Seat {seat+1} already booked.")

    def simulate_requests(self):
        for _ in range(10):
            seat = random.randint(0, SEATS-1)
            self.requests.put(seat)
```

```
        self.status.config(text="10 random booking requests queued.")

    def process_bookings(self):
        if self.processing:
            return
        self.processing = True
        threading.Thread(target=self._process_bookings_thread, daemon=True).start()

    def _process_bookings_thread(self):
        while not self.requests.empty():
            seat = self.requests.get()
            if self.locking.get() == "optimistic":
                self.optimistic_book(seat)
            else:
                self.pessimistic_book(seat)
            time.sleep(0.2)
        self.status.config(text="All bookings processed.")
        self.processing = False

    def optimistic_book(self, seat):
        # Simulate optimistic locking
        if not self.seats[seat]:
            # Simulate possible race
            time.sleep(random.uniform(0, 0.05))
            if not self.seats[seat]:
                self.seats[seat] = True
                self.buttons[seat].config(bg="red")
                self.status.config(text=f"Seat {seat+1} booked (optimistic).")
            else:
                self.status.config(text=f"Seat {seat+1} booking failed (already booked).")
        else:
            self.status.config(text=f"Seat {seat+1} already booked.")

    def pessimistic_book(self, seat):
        # Simulate pessimistic locking with a threading.Lock
        lock = threading.Lock()
        with lock:
            if not self.seats[seat]:
                self.seats[seat] = True
                self.buttons[seat].config(bg="red")
                self.status.config(text=f"Seat {seat+1} booked (pessimistic).")
            else:
                self.status.config(text=f"Seat {seat+1} already booked.")

if __name__ == "__main__":
    root = tk.Tk()
    app = BookingSystemGUI(root)
    root.mainloop()
```

```
"""
Question 6: Traffic Signal Management System (Multithreaded GUI)

Simulates a traffic intersection with FIFO and priority queueing for vehicles, with multithreaded signal and vehicle management.
"""
import tkinter as tk
import threading
import time
import random
from collections import deque
from queue import PriorityQueue

# --- Constants for UI Customization ---
BG_COLOR = "#2c3e50"
ROAD_COLOR = "#34495e"
LINE_COLOR = "#bdc3c7"
LIGHT_RED = "#c0392b"
LIGHT_YELLOW = "#f1c40f"
LIGHT_GREEN = "#27ae60"
BUTTON_BG = "#3498db"
BUTTON_FG = "#ecf0f1"
FONT_STYLE = ("Helvetica", 10, "bold")

class TrafficSignalGUI:
    def __init__(self, master):
        self.master = master
        self.master.title("Traffic Signal Management System")
        self.master.configure(bg=BG_COLOR)
        self.running = True
        self.signal_state = "Red"
        self.vehicle_queue = deque()
        self.emergency_queue = PriorityQueue()
        self.vehicles_on_road = []
        self.continuous_add = False
        self.car_colors = ["#e74c3c", "#9b59b6", "#2ecc71", "#1abc9c", "#f39c12"]

        self.setup_gui()
        self.start_threads()

    def setup_gui(self):
        self.canvas = tk.Canvas(self.master, width=800, height=500, bg=BG_COLOR, highlightthickness=0)
        self.canvas.pack(pady=10)
        self.draw_intersection()

        # --- Control Frame ---
        control_frame = tk.Frame(self.master, bg=BG_COLOR)
        control_frame.pack(pady=10)

        button_style = {"bg": BUTTON_BG, "fg": BUTTON_FG, "font": FONT_STYLE, "relief": tk.FLAT, "padx": 10, "pady": 5}
        tk.Button(control_frame, text="Add Car", command=self.add_vehicle, **button_style).pack(side=tk.LEFT, padx=5)
```

```
        tk.Button(control_frame, text="Add Emergency", command=self.add_emergency_vehicle, **button_style).pack(side=tk.LEFT, padx=5)
        tk.Button(control_frame, text="Toggle Auto-Add", command=self.toggle_continuous_add, **button_style).pack(side=tk.LEFT, padx=5)

        # --- Status Frame ---
        status_frame = tk.Frame(self.master, bg=BG_COLOR)
        status_frame.pack(pady=10)
        self.queue_label = tk.Label(status_frame, text="Waiting Cars: 0", font=FONT_STYLE, fg=BUTTON_FG, bg=BG_COLOR)
        self.queue_label.pack(side=tk.LEFT, padx=10)
        self.emergency_label = tk.Label(status_frame, text="Waiting Emergency: 0", font=FONT_STYLE, fg=LIGHT_RED, bg=BG_COLOR)
        self.emergency_label.pack(side=tk.LEFT, padx=10)

    def draw_intersection(self):
        # Roads
        self.canvas.create_rectangle(0, 200, 800, 300, fill=ROAD_COLOR, outline="") # Horizontal
        self.canvas.create_rectangle(350, 0, 450, 500, fill=ROAD_COLOR, outline="") # Vertical
        # Dashed lines
        for i in range(0, 800, 30):
            self.canvas.create_line(i, 250, i + 15, 250, fill=LINE_COLOR, width=2, dash=(4, 8))
        # Traffic light
        self.update_light_color()

    def add_vehicle(self, emergency=False):
        v_type = "Ambulance" if emergency else "Car"
        if emergency:
            self.emergency_queue.put((1, v_type))
        else:
            self.vehicle_queue.append(v_type)
        self.update_queue_labels()

    def add_emergency_vehicle(self):
        self.add_vehicle(emergency=True)

    def toggle_continuous_add(self):
        self.continuous_add = not self.continuous_add

    def start_threads(self):
        threading.Thread(target=self.signal_thread, daemon=True).start()
        threading.Thread(target=self.vehicle_processing_thread, daemon=True).start()
        threading.Thread(target=self.continuous_add_thread, daemon=True).start()
        self.master.after(50, self.animate_vehicles)

    def signal_thread(self):
        while self.running:
            self.signal_state = "Red"
            self.update_light_color()
            time.sleep(7)
            self.signal_state = "Green"
```

```
            self.update_light_color()
            time.sleep(5)
            self.signal_state = "Yellow"
            self.update_light_color()
            time.sleep(2)

    def continuous_add_thread(self):
        while self.running:
            if self.continuous_add:
                self.add_vehicle()
            time.sleep(random.uniform(1, 3))

    def vehicle_processing_thread(self):
        while self.running:
            if self.signal_state == "Green":
                if not self.emergency_queue.empty():
                    _, v_type = self.emergency_queue.get()
                    self.create_vehicle_animation(v_type, emergency=True)
                elif self.vehicle_queue:
                    v_type = self.vehicle_queue.popleft()
                    self.create_vehicle_animation(v_type)
                self.update_queue_labels()
            time.sleep(1)

    def create_vehicle_animation(self, v_type, emergency=False):
        color = LIGHT_RED if emergency else random.choice(self.car_colors)
        # Vehicle Body
        vehicle_id = self.canvas.create_rectangle(200, 235, 240, 265, fill=color,
outline="black", width=1)
        # Vehicle Roof
        roof_id = self.canvas.create_rectangle(210, 225, 230, 235, fill=color,
outline="black", width=1)
        self.vehicles_on_road.append((vehicle_id, roof_id))

    def animate_vehicles(self):
        for vehicle_id, roof_id in self.vehicles_on_road[:]:
            self.canvas.move(vehicle_id, 5, 0)
            self.canvas.move(roof_id, 5, 0)
            x1, _, _, _ = self.canvas.coords(vehicle_id)
            if x1 > 800:
                self.canvas.delete(vehicle_id)
                self.canvas.delete(roof_id)
                self.vehicles_on_road.remove((vehicle_id, roof_id))
        self.master.after(50, self.animate_vehicles)

    def update_light_color(self):
        self.canvas.delete("light")
        light_map = {"Red": LIGHT_RED, "Yellow": LIGHT_YELLOW, "Green": LIGHT_GREEN}
        self.canvas.create_oval(460, 160, 490, 190, fill=light_map[self.signal_state],
tags="light", outline="white")

    def update_queue_labels(self):
        self.queue_label.config(text=f"Waiting Cars: {len(self.vehicle_queue)}")
        self.emergency_label.config(text=f"Waiting Emergency:
```

```
{self.emergency_queue.qsize()}")

    def on_close(self):
        self.running = False
        self.master.destroy()

if __name__ == "__main__":
    root = tk.Tk()
    app = TrafficSignalGUI(root)
    root.protocol("WM_DELETE_WINDOW", app.on_close)
    root.mainloop()
```