Exercise 3

Explanation:

The function *find_min_max* efficiently determines the minimum and maximum values in an array. Instead of performing a naive iteration over all elements, it optimizes the process by comparing pairs of elements, reducing the total number of comparisons. The function follows these steps:

1. If the array is empty, it raises a *ValueError*.
2. If the array has only one element, it returns that element as both minimum and maximum.
3. It initializes the minimum and maximum values based on the first one or two elements.
4. It then iterates through the array in pairs, comparing two elements at a time and updating the minimum and maximum accordingly.
5. The function returns the minimum and maximum values.

Code:

```
import random

def find_min_max(arr):
    # Step 1: Check for empty array
    if not arr:
        raise ValueError("The array should not be empty")

    n = len(arr)

    # Step 2: Initialize variables
    if n == 1:
        return arr[0], arr[0]

    # Step 3: Initial comparison
    if n % 2 == 0: # If even number of elements
        if arr[0] < arr[1]:
            min_elem, max_elem = arr[0], arr[1]
        else:
            min_elem, max_elem = arr[1], arr[0]
        i = 2
    else: # If odd number of elements
        min_elem = max_elem = arr[0]
        i = 1

    # Step 4: Pairwise comparison
    while i < n - 1:
        # Compare in pairs to reduce the number of comparisons
        # Compare the smaller element with the min_elem
        # and the larger element with the max_elem
        if arr[i] < arr[i + 1]:
            min_elem = min(min_elem, arr[i])
            max_elem = max(max_elem, arr[i + 1])
        else:
            min_elem = min(min_elem, arr[i + 1])
            max_elem = max(max_elem, arr[i])
        i += 2
```

```
    # Step 5: Return results
    return min_elem, max_elem

# Example usage
V = [3, 5, 1, 2, 4, 8]
min_val, max_val = find_min_max(V)
print(f"Minimum: {min_val}, Maximum: {max_val}")

# Generate a large array with 1,000,000 random integers
large_array = [random.randint(1, 100000000) for _ in range(1000000)]

# Find the minimum and maximum values in the large array
min_val, max_val = find_min_max(large_array)
print(f"Minimum: {min_val}, Maximum: {max_val}")
```

Test Case:

Input: *[3, 5, 1, 2, 4, 8]*

Output: *Minimum: 1, Maximum: 8*

Exercise 5

Explanation:

The function *find_fastest_routes* calculates the shortest transmission time between a main server and all other computers in a network using Dijkstra's algorithm. The algorithm follows these steps:

1. Creates a graph where each computer is a node and cables represent edges with weights (transmission time).
2. Uses a priority queue to explore the shortest transmission times efficiently.
3. Initializes all transmission times to infinity, except for the main server (which starts at 0).
4. Iteratively selects the node with the shortest transmission time and updates its neighbours times.
5. Returns the shortest transmission times from the main server to all other computers.

Code:

```
import heapq

def find_fastest_routes(computers, cables, main_server):
    # Create the graph from the list of computers and cables
    graph = {computer: [] for computer in computers}
    for cable in cables:
        computer1, computer2, transmission_time = cable
        graph[computer1].append((computer2, transmission_time))
        graph[computer2].append((computer1, transmission_time))

    # Initialize the priority queue with the main server and a transmission time of 0
    pq = [(0, main_server)]
    # Initialize the transmission times dictionary with infinity for all computers
    t_transmision = {computer: float('inf') for computer in computers}
    # Set the transmission time for the main server to 0
```

```
    t_transmision[main_server] = 0
    # Initialize the visited set to keep track of processed computers
    visited = set()

    while pq:
        # Get the computer with the smallest transmission time from the priority queue
         # Used a heap to simplify the process of getting the smallest transmission time,
could be done with a regular list and min function
        current_transmission_time, current_computer = heapq.heappop(pq)

        # If the current computer has already been visited, skip it
        if current_computer in visited:
            continue

        # Mark the current computer as visited
        visited.add(current_computer)

        # Iterate over the neighbors of the current computer
        for neighbor, delay in graph[current_computer]:
            # Calculate the new transmission time to the neighbor
            new_transmission_time = current_transmission_time + delay

            # If the new transmission time is smaller, update it
            if new_transmission_time < t_transmision[neighbor]:
                t_transmision[neighbor] = new_transmission_time
                # Push the neighbor and its transmission time to the priority queue
                heapq.heappush(pq, (new_transmission_time, neighbor))

    # Return the dictionary of transmission times from the main server to all other
computers
    return t_transmision

# Example usage
computers = ['A', 'B', 'C', 'D']
cables = [
    ('A', 'B', 5),
    ('A', 'C', 1),
    ('B', 'C', 2),
    ('B', 'D', 5),
    ('C', 'D', 1)
]
main_server = 'A'
t_transmision = find_fastest_routes(computers, cables, main_server)
print(f"Fastest routes from {main_server}: {t_transmision}")
```

Test Case:

Input:

```
computers = ['A', 'B', 'C', 'D']
cables = [
    ('A', 'B', 5),
    ('A', 'C', 1),
    ('B', 'C', 2),
    ('B', 'D', 5),
    ('C', 'D', 1)
]
```

main_server = 'A'

Output:

Fastest routes from A: {'A': 0, 'B': 3, 'C': 1, 'D': 2}

Exercise 6

Explanation:

This function calculates the minimum number of courts needed to accommodate all reservations without overlap.

1. **Sort reservations by start time** to process them in order.
2. **Use a list (***courtsInUse***) to track active courts.**
    - If a court is free (its reservation has ended), reuse it.
    - Otherwise, allocate a new court.
3. **Return the total number of courts used.**

Code:

```
def courts(reservations):
   n = len(reservations)
   reservations= sorted(reservations) #sort the reservations by start time
   courtsInUse=[] #list to keep track of the courts in use
   for i in range(n):
      if len(courtsInUse) == 0:
         courtsInUse.append(reservations[i])
      else:
         for j in range(len(courtsInUse)):
               if reservations[i][0] >= courtsInUse[j][1]:#if the reservation starts after the
end of the last one
                  courtsInUse[j] = reservations[i] #replace the last reservation with the new
one
               break
         else:
            courtsInUse.append(reservations[i]) # if no court is available, add a new one

   return len(courtsInUse)
```

Test Cases:

Test Case 2: Edge case with varying reservation durations

Input: reservations = [(1, 2), (2, 9), (3, 4), (5, 6), (9, 12), (9, 12)]

Expected Output: Minimum number of courts required: 2

Explanation:

(1,2) uses one court. (2,9) reserves that court.

(3,4) and (5,6) collide with (2,9), and take another court

One of the (9,12) goes to the first court, and the other to the second one

COURT1[(1,2),(2,9),(9,12)] COURT2[(3,4),(5,6),(9,12)]