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):
    if not arr:
        raise ValueError("The array should not be empty")

    n = len(arr)
    if n == 1:
        return arr[0], arr[0]

    if n % 2 == 0:
        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:
        min_elem = max_elem = arr[0]
        i = 1

    while i < n - 1:
        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

    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}")
```

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 neighbors' 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):
    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))

    pq = [(0, main_server)]
    t_transmision = {computer: float('inf') for computer in computers}
    t_transmision[main_server] = 0
    visited = set()

    while pq:
        current_transmission_time, current_computer = heapq.heappop(pq)
        if current_computer in visited:
            continue
        visited.add(current_computer)

        for neighbor, delay in graph[current_computer]:
            new_transmission_time = current_transmission_time + delay
            if new_transmission_time < t_transmision[neighbor]:
                t_transmision[neighbor] = new_transmission_time
                heapq.heappush(pq, (new_transmission_time, neighbor))

    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}
```