5. For problem 3:
Time and Space Complexity Analysis

1. <u>Constructor: Election()</u>
Time Complexity: O(1)
Space Complexity: O(1)
Explanation: This constructor simply initializes a HashMap and a PriorityQueue, both of which are empty at the start. These operations take constant time and space, as no input or dynamic data processing is involved during construction.

2. <u>initializeCandidates(List<String> candidates)</u>
Time Complexity: O(n)
Space Complexity: O(n)
Explanation: This method begins by clearing any existing data from the HashMap and PriorityQueue, which takes constant time since both are reset. It then iterates over the list of n candidate names. Each candidate is added to the HashMap in O(1) time and to the heap in O(log n) time. Since there are n insertions into the heap, the total time is O(n log n). Some heap implementations optimize bulk additions and achieve O(n) via heapify, but under general assumptions, we consider the worst-case complexity to be O(n log n). The space complexity is linear because both the HashMap and PriorityQueue store n candidate entries.

3. <u>setTotalVotes(int p)</u>
Time Complexity: O(1)
Space Complexity: O(1)
Explanation: Simple assignment operation - performs a simple variable assignment to store the total number of votes, which is a constant-time and constant-space operation.

4. <u>castVote(String candidate)</u>
Time Complexity: O(log n)
Space Complexity: O(1)
Explanation: Casting a vote involves updating the candidate's vote count in the HashMap, which takes O(1) time. To maintain the correct order in the PriorityQueue, the candidate is removed and reinserted or the heap is re-balanced by offering a new instance, which typically takes O(log n). The space usage remains constant, as no additional storage is created per vote beyond temporary objects or reused references.

5. <u>castRandomVote()</u>
Time Complexity: O(n) for worst case, O(1) average case
Space Complexity: O(1)
Explanation: It converts the set of candidate names from the HashMap into a list to facilitate random access. This conversion step takes O(n) time in the worst case. However, if the list is cached or reused, it could be considered O(1) on average. Selecting a random index and accessing it are both constant-time operations. Finally, it calls castVote, which takes O(log n),

but the dominating factor here remains the initial conversion to a list. Since no permanent data structures are created or expanded, the space complexity is constant.

6. rigElection(String candidate)
Time Complexity: O(n log n)
Space Complexity: O(1)
Explanation: This method resets all vote counts to zero by iterating over the HashMap, which takes O(n) time. It clears the heap entirely (O(n)), and then re-inserts the rigged candidate and a few others with manipulated vote counts. Each re-insertion into the PriorityQueue takes O(log n), and while the number of such insertions may be small (constant), the overall complexity is still dominated by the O(n log n) nature of heap operations across all entries. No additional persistent storage is created, so space usage remains constant beyond the original structures.

7. getTopKCandidates(int k)
Time Complexity: O(n log n)
Space Complexity: O(n)
Explanation: To retrieve the top k candidates, this method leverages Java streams to sort all candidates based on vote count. Sorting the n candidates takes O(n log n) time using TimSort. The result is then limited to the top k, which is a constant or bounded operation, and mapped into a new collection, which takes O(n) in the worst case. The sorting step is the dominant factor, and a new list is generated, hence the space complexity is linear.

8. auditElection()
Time Complexity: O(n log n)
Space Complexity: O(n)
Explanation: This method is functionally similar to getTopKCandidates, except it prints the top candidates instead of returning them. It still requires sorting the entire candidate list, which takes O(n log n), and builds intermediate stream structures, contributing to an O(n) space requirement. As with the previous method, the complexity is driven by the sorting operation.