1.Iniatilize Candidates Time and Space Complexity

```
public void initializeCandidates(LinkedList<String> candidates) {
   int size = candidates.size();
   Space O(N), N = size of candidates list
   - In the loop below, every node in candidates is put into the hash map
    this.votingMap = new HashMap<>();


   Space O(N), N = size of candidates list
   this.candidateMaxHeap = new String[size];

   Time: O(N), N = size of candidates list
   for(int i = 0; i < size; i++) {\
       votingMap.put(candidates.get(i), 0);

       Time: O(N), N = size of candidates list
       - Searching a linked list node by node
       candidateMaxHeap[i] = candidates.get(i);
   }

}
```

Time: O(N^2) where N is the size of the candidate's list
Space: O(N) where N is the size of the candidate's list

2. Cast Vote Time and Space Complexity

```
public void castVote(String candidate) {
   Space O(1): We are not changing the number of items in the hashmap
   votingMap.put(candidate, votingMap.get(candidate) + 1);

   // Time: O(M), M = length of array candidateMaxHeap
   for(int i = candidateMaxHeap.length/2; i >= 0 ; i--) {
       // Time: log(M), M = length of array candidateMaxHeap
       heapify(candidateMaxHeap, i);
   }
}
```

Time: O(M*log(M)) where M is the length of array candidateMaxHeap, but M is equal to the length of the original candidate list (N). So, it can also be written as O(N*log(N)).
Space: O(1)

3. Cast Random Vote

```
public void castRandomVote(){
   // Time of generating next random int => O(1)
   // Space => O(1)
   https://docs.oracle.com/javase/1.5.0/docs/api/java/util/Random.html
   synchronized protected int next(int bits) {
      seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
      return (int)(seed >>> (48 - bits));
   }
   int rand_int = gen.nextInt(candidateMaxHeap.length);
   String candidate = candidateMaxHeap[rand_int];

   // Time O(M*log(M)) where M is the length of array candidateMaxHeap
   // Space(1)
   -Refer to cast vote space and time complexity above
   castVote(candidate);
}
```

Time: O(M*log(M)) where M is the length of array candidateMaxHeap, but M is equal to the length of the original candidate list (N). So, it can also be written as O(N*log(N)).
Space: O(1)

4. Rig Election

```
public void rigElection(String candidate) {
   // Space O(1)
   int totalVotes = 0;

   // Space O(1)
   //  Time O(P) where P total amount of key-value pairs stored in votingMap
   for(int votes : votingMap.values()) {
       totalVotes += votes;
   }

   //  Time O(P) where P total amount of key-value pairs stored in votingMap
   for(String key : votingMap.keySet()) {
       votingMap.put(key, 0);
   }

   // Time O(1)
   votingMap.put(candidate, totalVotes);

}
```

Time: O(P) where P is the amount of key-value pairs stored in votingMap, but P is equal to the length of the original candidate list (N). So, it can also be written as O(N).
Space: O(1)

5. Get Top K Candidates

```
public String[] getTopKCandidates ( int k){
    // Space: O(k), we have to create an array of size k
    String[] topk = new String[k];
    int heapLen = candidateMaxHeap.length;

    // Remove root of heap K times
    // Time: O(k)
    for (int i = 0; i < k && i < heapLen; i++) {
        topk[i] = candidateMaxHeap[0];
        candidateMaxHeap[0] = candidateMaxHeap[heapLen - 1 - i];
        candidateMaxHeap[heapLen - 1 - i] = null;

        // Time: O(M), M = length of array candidateMaxHeap
        for (int j = candidateMaxHeap.length / 2; j >= 0; j--) {
            // Time: log(M), M = length of array candidateMaxHeap
            heapify(candidateMaxHeap, j);
        }
    }

    // Add removed nodes back into heap
    // Time: O(k)
    for (int i = 0; i < k && i < heapLen; i++) {
        candidateMaxHeap[heapLen - 1 - i] = topk[i];
    }

    //reheapify
    // Time: O(M), M = length of array candidateMaxHeap
    for (int j = candidateMaxHeap.length / 2; j >= 0; j--) {
        // Time: log(M), M = length of array candidateMaxHeap
        heapify(candidateMaxHeap, j);
    }

    return topk;
}
```

Time: O(k*M*log(M)) where M is the length of the candidateMaxHeap and k is the int parameter for the function. Because M is equal to N, we can also rewrite this as O(k*N*log(N)), where N is the length of the original candidate list.

Space: O(k), where k is the int variable named k.

6. Audit Election

```
public void auditElection () {
   // Time O(k*M*log(M), where M is the length of array candidateMaxHeap, but
   since k = M, the length of the array candidateMaxHeap,
   Time =>  O(M^2*log(M))

   // Space O(k) k = M, the length of the array candidateMaxHeap, Time => O(M)
   String[] candidates = getTopKCandidates(candidateMaxHeap.length);

   // Time: O(M) loop iterates the length of candidateMaxHeap(M)
   // Space: O(1)
   for (int i = 0; i < candidateMaxHeap.length; i++) {
       // O(S) where is the number of characters in the candidate's name with
       the most characters. String concatenation requires a new String to be
       created and copied character by character.
       System.out.println(candidates[i] + " - " + votingMap.get(candidates[i]));
   }
}
```

Time: O(M^2*log(M)) + O(M*S), where M is the length of the array candidateMaxHeap, and S is the number of characters in the candidate's name with the most characters. Realistically, O(M^2*log(M)) will grow faster, so O(M*S) can be excluded. Also, because M equals N, the length of the original candidate list, the time complexity can be rewritten as O(N^2*log(N)).
Space: O(M), where M is the length of the array candidateMaxHeap.