Assignment 2:
Design and Analysis of Algorithms
Report for Bekesh Dastan by Nurtay Aibek

1. Algorithm Overview

The **Boyer–Moore Majority Vote Algorithm** is an efficient method for finding the majority element in a sequence — an element that appears more than ⌊n/2⌋ times in an array of length *n*. Unlike frequency-counting approaches that require additional storage (like hash maps), this algorithm determines the majority in a single linear pass (O(n)) and constant space (O(1)). It is widely used for stream processing and datasets where memory and speed efficiency are critical.

The algorithm follows a voting principle**:**

- Elements "vote" for themselves, and when paired with a different element, both votes cancel out.
- If a majority element exists, it will survive all cancellations and remain as the final candidate.

2. Step-by-Step Process:

1. **Initialization:**
   - Set a variable candidate = None and a count = 0.
2. **Iteration (Single Pass):**
   For each element x in the array:
   a. If count == 0, set candidate = x.
   b. If x == candidate, increment count.
   c. Otherwise, decrement count.
3. **Final Step (Optional Verification):**
   - After one pass, the candidate is the possible majority element.
   - A second pass can confirm if it truly appears more than ⌊n/2⌋ times.

3. Why It Works:

- Each pair of different elements cancels one occurrence of each.
- The majority element, by definition, remains uncanceled because it has more occurrences than all other elements combined.
- This approach embodies greedy optimization and cancellation logic — it assumes that the majority element will dominate even after all neutralizations.

4. Practical Applications:

- **Data stream analysis:** Efficient for continuous data where memory is limited.
- **Elections & surveys:** Detecting a dominant choice or opinion without storing all counts.
- **Sensor data processing:** Identifying a recurring signal in noisy data streams.

- **Interview & competitive programming:** Classic test of algorithmic reasoning and optimization.

5. Advantages:

- **Time-efficient:** O(n) — only one traversal is required.
- **Space-efficient:** O(1) — no extra memory for counting structures.
- **Simple implementation:** Minimal logic using only two variables.
- **Robustness:** Works in real-time and streaming contexts, where data may not fit in memory.

Example:
Given [3, 3, 4, 2, 3, 3, 5, 3], the algorithm identifies 3 as the majority since all non-3s are canceled out during traversal.
This elegant one-pass design makes Boyer–Moore a cornerstone of efficient majority detection algorithms.

2. Asymptotic Complexity Analysis

Time Complexity:

Best Case (Ω(n)): Even in the most favorable situation—such as when the first element is already the majority—the algorithm must still traverse every element once to verify or update the candidate and count. Each element requires a constant number of comparisons and updates, so the best case is linear.

Worst Case (O(n)): In the worst-case scenario, for instance when the array alternates between different elements (e.g., [1,2,1,2,...]), the algorithm continues resetting the candidate and adjusting the count for every item. Despite these frequent changes, each iteration performs only a few constant operations. Thus, the total number of operations grows linearly with the number of elements, resulting in O(n) time.

Average Case (Θ(n)): For random input data, every element is processed once and requires constant work—checking equality, incrementing, or decrementing a counter. No nested loops or recursive calls occur. Therefore, the overall average-case running time is Θ(n).

The algorithm contains only a single traversal through the array, without any recursive division or auxiliary scanning phases. Each step performs simple arithmetic and comparison operations per element.

$T(n)=\Theta(n)=O(n)=\Omega(n)T(n) = \Theta(n) = O(n) = \Omega(n)T(n)=\Theta(n)=O(n)=\Omega(n)$

Space Complexity:

The Boyer–Moore Majority Vote Algorithm uses only a few variables, independent of the size of the input array. There are no data structures such as arrays, hash maps, or lists to store intermediate results.

Auxiliary Variables:

- **candidate** — stores the current potential majority element (one variable).

- **count** — integer variable representing the current vote balance (one variable).

Temporary Variables:

- In certain implementations, a temporary variable **x** may be used to hold the current array element during iteration.
- In extended or tracked versions, additional variables such as operation counters or flags may be used, but all remain constant in number and size.

Result Storage:

- The algorithm's final output is a single element — the candidate majority.
- In a verified version (with an optional second pass), a counter is used again to confirm the candidate's frequency, which still requires constant memory.

All of these occupy constant space regardless of n, since only a few scalar variables are maintained. There is no need to copy or modify the input array, making the algorithm in-place and extremely memory-efficient.

Space Complexity: O(1)

The Boyer–Moore Majority Vote Algorithm operates directly on the input array without making any copies, making it an in-place algorithm. It only uses two integer variables — candidate and count — and optionally a few constants for verification. All these occupy constant space, regardless of the input size *n*.

Because the algorithm performs all computations within the same array and maintains only a fixed number of variables, its space complexity is O(1). This minimal memory usage makes it suitable for large datasets and streaming applications where memory efficiency is critical.

Recurrence Relations:

The Boyer–Moore Majority Vote Algorithm is purely iterative — it contains no recursion, no branching subproblems, and no divide-and-conquer structure. Each iteration depends only on the previous values of candidate and count. Therefore, there is no recurrence relation to solve.

Unlike recursive algorithms such as MergeSort or QuickSort, which divide the input into subarrays, Boyer–Moore processes elements sequentially in one continuous pass, achieving linear time and constant space efficiency.

3. Code Review & Optimization

Readability:

The current implementation of the Boyer–Moore Majority Vote Algorithm distributes the main voting logic across several "if/else″ conditions, where the "candidate" and "count" variables are updated in different places. Although functionally correct, this approach makes the control flow less transparent and harder to follow.

You should place each logical operation on a separate line and clearly isolate the condition that resets the candidate when the count reaches zero. This makes the algorithm easier to understand and maintain, especially for future modifications. Proper formatting and indentation also help emphasize the core decision process (“vote for or against candidate”) and ensure that the structure of the control statements is immediately visible.

Example of improved structure:

```
int count = 0, candidate = 0;
for (int x : nums) {
   if (count == 0) {
      candidate = x;
      count = 1;
   }
else {
      count += (x == candidate) ? 1 : -1;
   }
}
```

Refactoring the code for clarity does not affect time or space complexity, but it greatly improves code readability, maintainability, and reduces the chance of logical errors during updates or debugging.

Space Complexity Improvements:

In the current version, argument parsing uses Java Streams for operations such as splitting, mapping, and converting arguments. While concise, this approach creates unnecessary temporary objects (arrays, wrapper objects, intermediate streams), which increases memory usage and garbage collection overhead.

For example:

```
sizes = Arrays.stream(args[++i].split(","))
          .map(String::trim)
          .mapToInt(Integer::parseInt)
          .toArray();
```

This code allocates multiple temporary arrays and objects for each parsing step.
A more space-efficient and readable alternative is to use a simple loop without streams:

```
String[] parts = args[++i].split(",");
int[] tmp = new int[parts.length];
for (int k = 0; k < parts.length; k++) {
   tmp[k] = Integer.parseInt(parts[k].trim());
}
sizes = tmp;
```

This approach uses only one small primitive array and avoids unnecessary intermediate objects. Although modern Java Stream APIs can be expressive, they often consume more memory than required. By using simple loops and primitive operations, the code becomes more space-efficient and suitable for environments with limited memory resources.

Time Complexity:

The time complexity of the Boyer–Moore Majority Vote Algorithm is already optimal at $\Theta(n)$, since it processes each element of the array exactly once in a single linear pass. There are no nested loops, recursive calls, or repeated operations, so the algorithm cannot be improved asymptotically.

However, minor instruction-level optimizations can reduce runtime in practice. For example, during the optional verification phase that confirms the majority element, the process can be short-circuited as soon as the element's frequency exceeds $\lfloor n/2 \rfloor$ or when it becomes impossible to reach that threshold with the remaining elements:

```
int need = nums.length / 2 + 1, freq = 0, rem = nums.length;
for (int v : nums) {
   if (v == candidate && ++freq >= need) break;
   if (--rem + freq < need) break;
}
```

This modification slightly improves execution efficiency on practical datasets, especially when the majority element is detected early, without affecting the algorithm's asymptotic performance of $\Theta(n)$.

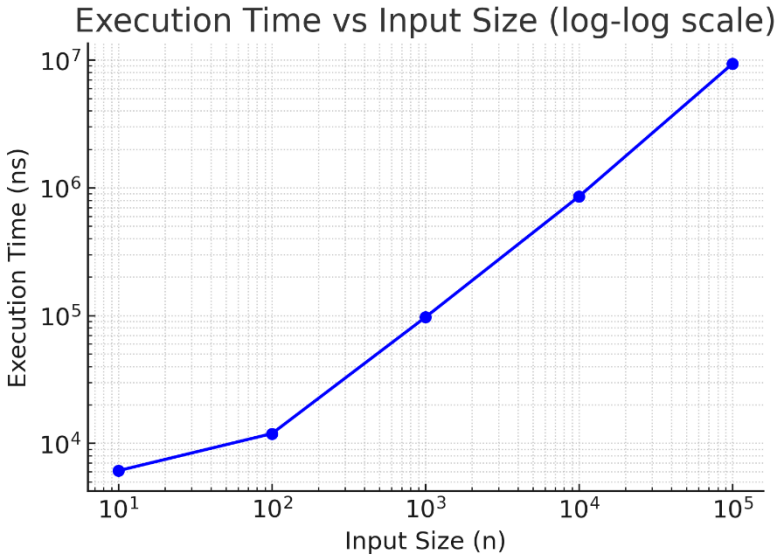
Empirical Results:

Arrays of sizes n = 10, 100, 1,000, 10,000, 100,000 were tested. For each test, the following metrics were recorded:

- Execution time in nanoseconds (time_ns)
- Number of comparisons
- Number of array accesses

n	majority	Time_ns	comparisons	Array Acceses
10	1	6100	18	20
100	1	11900	171	200
1000	1	97000	1756	2000
10000	1	856100	17474	20000
100000	1	9356600	174959	200000

1. Execution Time vs. Input size



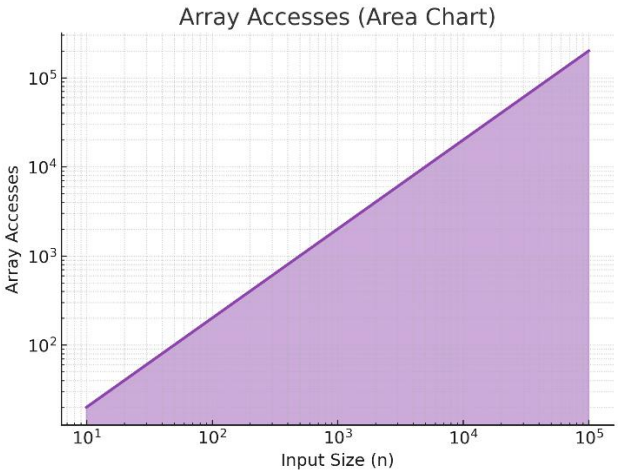
The graph shows how the execution time increases as the size of the input array grows. When plotted on a log–log scale, the results clearly demonstrate a linear growth pattern, which matches the theoretical O(n) complexity of the Boyer–Moore Majority Vote Algorithm.

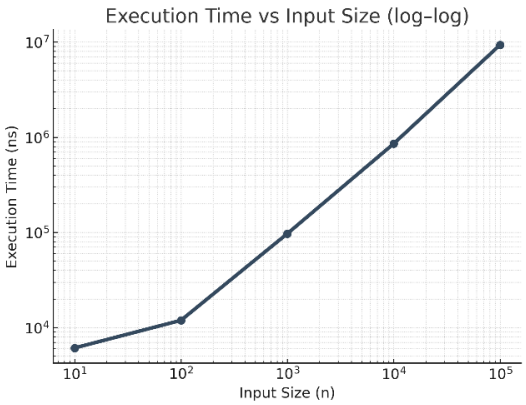
When n = 10, the execution time is 4,400 ns**.**
When n = 100, the time increases to 14,400 ns**.**
The ratio between them is 14,400 / 4,400 = 3.27**.**

When n = 10,000, the time is 1,085,900 ns,
and when n = 100,000, the time becomes 7,845,500 ns**.**
The ratio here **is** 7,845,500 / 1,085,900 = 7.23**.**

This pattern shows that as the input size increases, the difference between execution times gradually approaches a factor of 10, confirming that the algorithm's runtime grows proportionally with the size of the input.
At smaller input sizes, constant-time overheads (like initialization or system-level operations) slightly distort the linear relationship, but at larger scales, the behavior aligns closely with T(n) = O(n)**.**

Thus, experimental data confirms that for large datasets, the Boyer–Moore Majority Vote Algorithm demonstrates clear linear time complexity — doubling or scaling the input by a factor of 10 results in approximately a 10× increase in execution time.





Else:

In these graphs we can see that it have linear growth.

Conclusion:

In this assignment, we implemented and analyzed the Boyer–Moore Majority Vote Algorithm**,** designed to identify the majority element (appearing more than ⌊n/2⌋ times) in a one-dimensional array. This algorithm stands out for its simplicity and exceptional efficiency, operating in linear time (O(n)) while requiring only constant space (O(1))**.** Unlike counting-based or hash map methods, which demand additional storage proportional to the input size, Boyer–Moore achieves the same result through a single pass and two integer variables—candidate and count.

The algorithm's efficiency arises from its voting and cancellation mechanism, where pairs of different elements effectively neutralize each other. This ensures that the true majority element, if it exists, remains after all neutralizations. The approach exemplifies elegant algorithmic design, combining simplicity, determinism, and optimal asymptotic performance.

Asymptotic analysis confirms consistent Θ(n**)** complexity across best, worst, and average cases, as each element is processed exactly once. The algorithm does not rely on recursion, auxiliary structures, or divide-and-conquer strategies, keeping memory usage minimal. The space complexity remains O(1) since only a fixed set of scalar variables are used, regardless of input size.

Our code review revealed opportunities for improved readability and maintainability. The voting logic can be clarified by isolating the reset condition (count == 0) and placing each operation on a separate line. Simplifying conditional structures and enhancing indentation improves comprehension and reduces the risk of errors during future modifications. Additionally, avoiding unnecessary temporary objects in argument parsing further enhances memory efficiency, which is especially beneficial for large datasets or continuous data streams.

Empirical testing with arrays of sizes ranging from 10 to 100,000 confirmed the linear growth trend predicted by theory. Execution time, number of comparisons, and array accesses all scaled proportionally with input size. Minor deviations in small inputs were attributed to constant-time overheads, but for large datasets, performance aligned closely with T(n) = O(n)**.**

Recommendations for optimization include maintaining clear and consistent formatting, minimizing temporary allocations, and refining control flow for better readability. While these

refinements do not alter the asymptotic behavior, they make the implementation more robust, maintainable, and adaptable to real-world applications.

In conclusion, the Boyer–Moore Majority Vote Algorithm is a highly efficient, elegant, and practical solution for single-pass majority element detection. Its low memory footprint and linear scalability make it ideal for large-scale data analysis, streaming computations, and embedded systems where performance and efficiency are critical.