Report for the Kadane's algorithm

Assignment 2
Daniil Khvan
Aldiyar Zhaksylyk

The algorithm works in linear time. Basically, it always goes through the whole array once and checks each element. That's why the time complexity is O(n). It doesn't matter if the case is best, worst, or average — it's always the same because the loop runs the same way.

For memory, it's also simple: the algorithm only uses a few variables (for the candidate and a counter). So the space complexity is O(1). It doesn't make any extra arrays or lists. It's also not recursive, just one loop, so there's no recurrence relation here.

Problems with the Code

1. No input check. If the array is null or empty, the code breaks right away because it tries to read the first element.
2. No candidate verification. The Boyer-Moore algorithm can give back some element even if there is no real majority. Normally, you need a second pass to check if the element really appears more than n/2 times.
3. No metrics. The code doesn't track comparisons or array accesses, which are useful for analysis.
4. Bad names. The class and method names could be better. For example, instead of solution, it would be clearer to call it findMajorityElement. The class could be BoyerMooreMajorityVote.
5. No comments. Without Javadoc or normal comments, the code looks dry and harder to understand.

Optimizations

- Time: The algorithm is already optimal at O(n). You can't really make it faster, since you need to look at every element anyway. You can add the verification step, but that's another O(n) pass.
- Space: Already optimal at O(1). No extra memory is needed.

Conclusion

The algorithm is fast and memory-efficient, but the code is kind of sloppy. It should have input validation, candidate verification, better names, and some metric tracking. With these fixes, it would be both correct and easier to read.

Report for Boyer-Moore

The Boyer-Moore Majority Vote Algorithm is used to find the element that shows up more than half the time in an array. It works in one pass by keeping a candidate and a count. If the current element matches the candidate, count goes up. If it doesn't match, count goes down. If count hits zero, the candidate gets replaced. To be safe, we can do a second pass to check if the candidate really is the majority. This algorithm is often used in things like voting systems or analyzing data streams.

Complexity Analysis

- Time Complexity:
  It's always O(n). The first version only does one pass, which is O(n). If you add the verification pass, it's still O(n), just with a bit more work. Basically, it's just one comparison per element, maybe two if you verify.
- Space Complexity:
  It's O(1), since the algorithm only needs two variables (candidate and count). No extra arrays are used.
- Comparison with Kadane's Algorithm:
  Both run in O(n) time and O(1) space. The difference is Boyer-Moore with verification does almost 2n comparisons, kind of like how Kadane's does two checks for each element.

Code Review

Problems in the code:

1. No check for null or empty arrays.
2. No verification step, so it might return the wrong element.
3. Doesn't track metrics like comparisons or array accesses.
4. Bad naming — solution is not clear, better would be findMajorityElement.

Suggestions to improve:

- Add null/empty array checks.
- Add the second verification pass.
- Track comparisons and accesses for analysis.
- Use clear names for methods and classes.

Empirical Results

Tests showed that the algorithm's runtime grows linearly with n, like theory says.

- With verification, runtime roughly doubles, but it's still O(n).
- Comparisons also grow close to 2n when verification is included.
- Because the algorithm is simple, constant factors are small and it runs very efficiently.

Conclusion

The improved Boyer-Moore algorithm is fast (O(n) time) and memory-efficient (O(1) space). Adding verification makes it always correct, though it doubles the number of comparisons. In practice, it still runs fast and matches the theory. With better names, input checks, and metrics, the code becomes much more reliable.