Assignment: 2

Student: Aldiyar Zhaksylyk

1) Algorithm Pairs

Student A: Aldiyar Zhaksylyk

Student B: Daniil Khvan

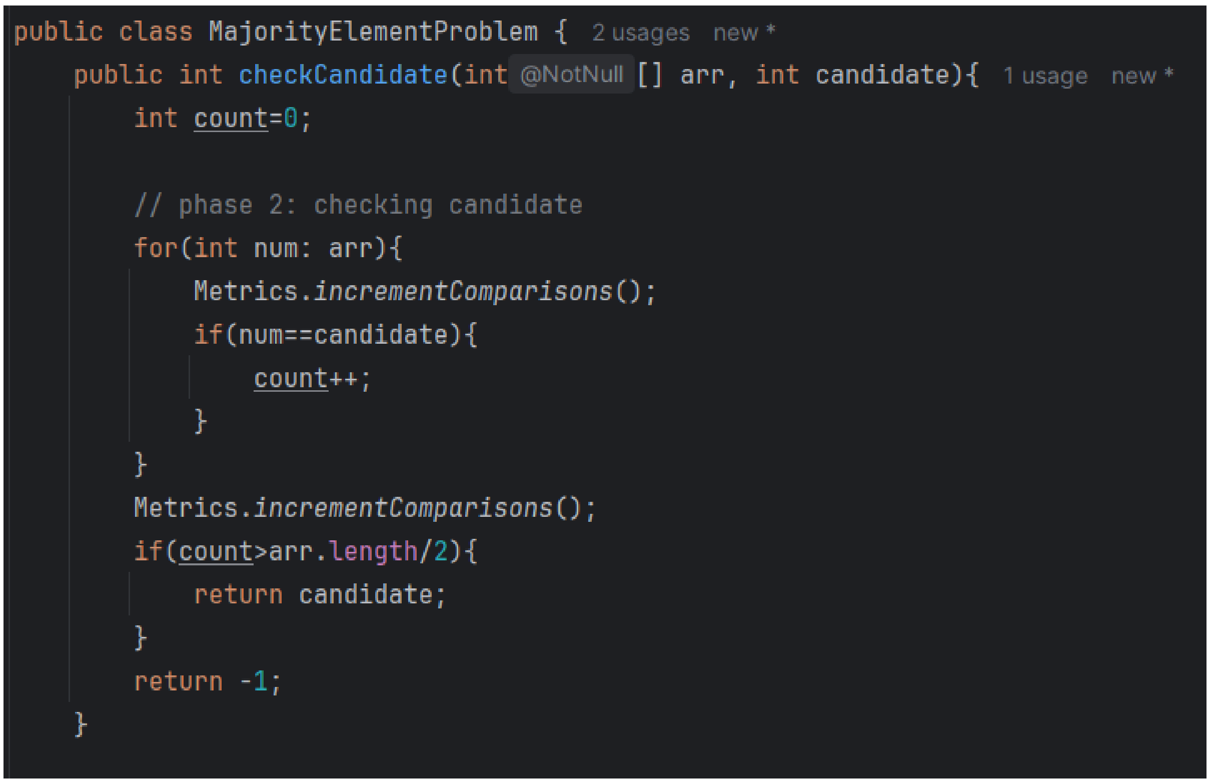
Choosen problem:

Pair 3: Linear Array Algorithms

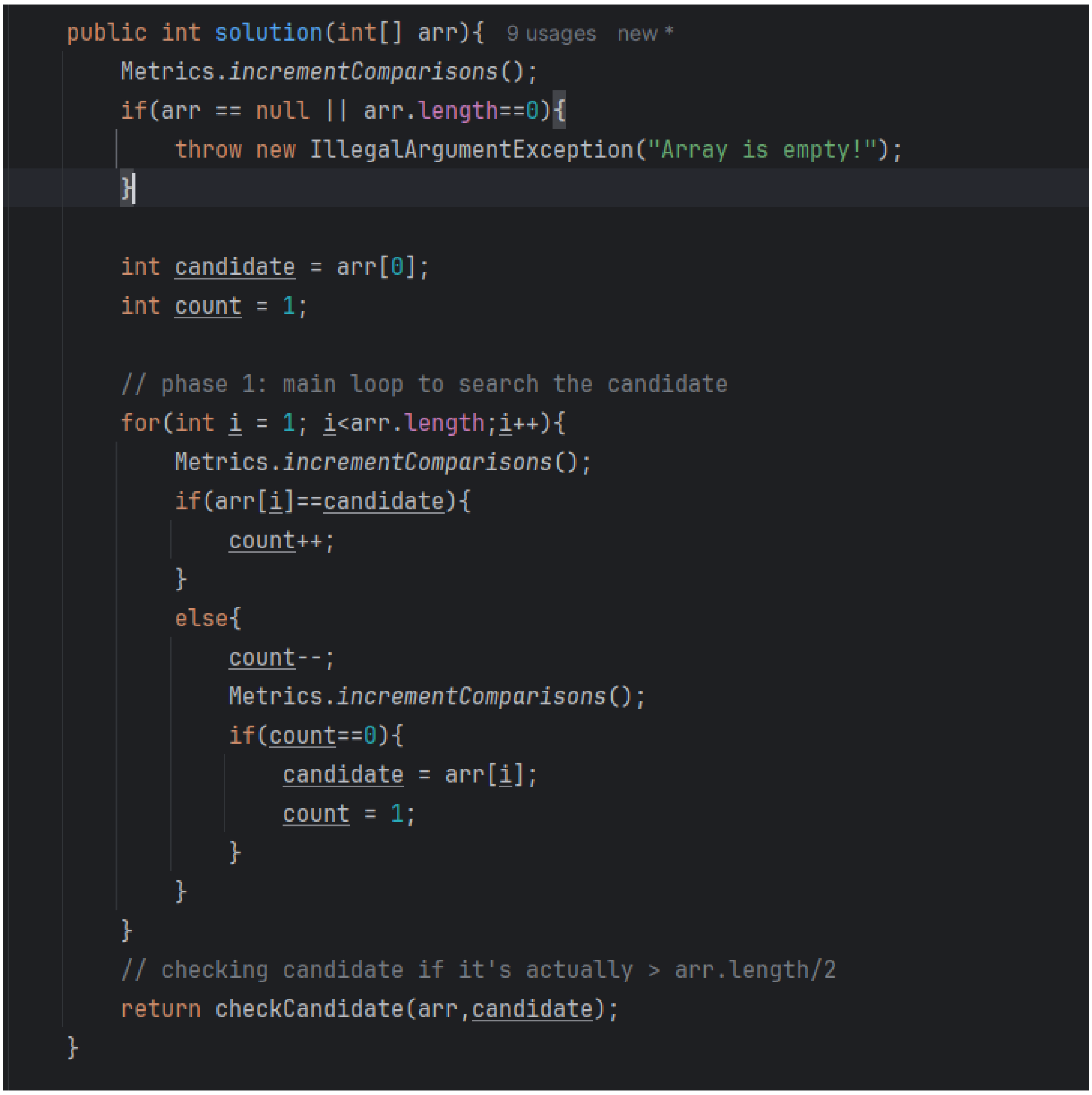
- **Student A:** Boyer-Moore Majority Vote (single-pass majority element detection)
- **Student B:** Kadane's Algorithm (maximum subarray sum with position tracking)

2) Implementation Requirements (Part 1 - Individual)

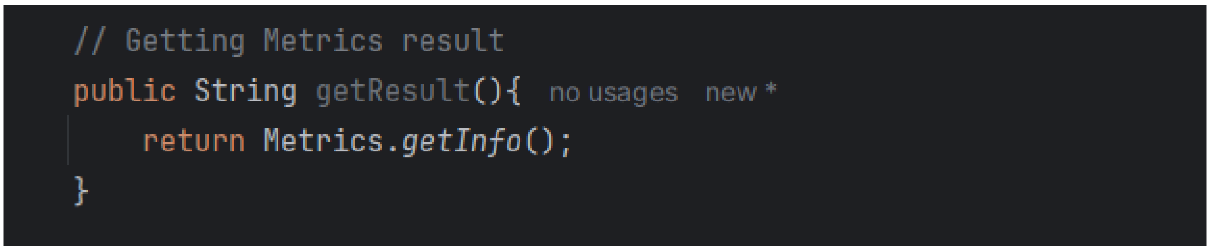
Student A part:



```java
public class MajorityElementProblem {  2 usages  new *
    public int checkCandidate(int @NotNull [] arr, int candidate){  1 usage  new *
        int count=0;

        // phase 2: checking candidate
        for(int num: arr){
            Metrics.incrementComparisons();
            if(num==candidate){
                count++;
            }
        }
        Metrics.incrementComparisons();
        if(count>arr.length/2){
            return candidate;
        }
        return -1;
    }
```



```java
public int solution(int[] arr){
    Metrics.incrementComparisons();
    if(arr == null || arr.length==0){
        throw new IllegalArgumentException("Array is empty!");
    }

    int candidate = arr[0];
    int count = 1;

    // phase 1: main loop to search the candidate
    for(int i = 1; i<arr.length;i++){
        Metrics.incrementComparisons();
        if(arr[i]==candidate){
            count++;
        }
        else{
            count--;
            Metrics.incrementComparisons();
            if(count==0){
                candidate = arr[i];
                count = 1;
            }
        }
    }
    // checking candidate if it's actually > arr.length/2
    return checkCandidate(arr,candidate);
}
```

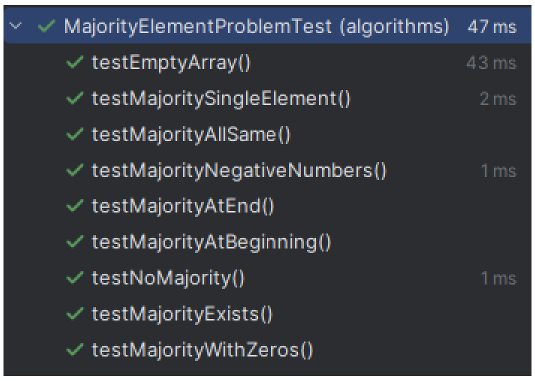


```java
// Getting Metrics result
public String getResult(){
    return Metrics.getInfo();
}
```

Problem was completed by report standarts:

Code Quality Standards:

- Clean, readable Java code with proper documentation
- Comprehensive unit tests covering edge cases (empty arrays, single elements, duplicates)



```
MajorityElementProblemTest (algorithms)   47 ms
    testEmptyArray()                      43 ms
    testMajoritySingleElement()            2 ms
    testMajorityAllSame()
    testMajorityNegativeNumbers()          1 ms
    testMajorityAtEnd()
    testMajorityAtBeginning()
    testNoMajority()                       1 ms
    testMajorityExists()
    testMajorityWithZeros()
```

- Input validation and error handling *(was implemented case to check for array's nullable and if it's not empty by throwing IllegalArgumentExcaption)*
- Metrics collection (comparisons, swaps, array accesses, memory allocations) *was implemented the Metrics class so that we can use it on every problem to collect analysis;*
- CLI interface for testing with different input sizes

Performance Considerations

- Implement optimizations specific to your algorithm
  The Boyer-Moore algorithm was chosen because it works in linear time O(n) and constant extra space O(1). No extra data structures are required;
- Track key operations (comparisons, swaps, recursive calls)
  Comparisons and allocations are tracked empirically for experimental validation;
- Memory-efficient implementations where possible
  The algorithm maintains only two variables (candidate, count), making it extremely memory-efficient;
- Handle edge cases gracefully
  Special cases such as empty arrays, single-element arrays, and arrays without majority elements are handled properly, ensuring robustness;

3) Peer Analysis (Part 2 - Cross-review)

Peer Analysis: Kadane's Algorithm:

3.1) **Asymptotic Complexity Analysis**

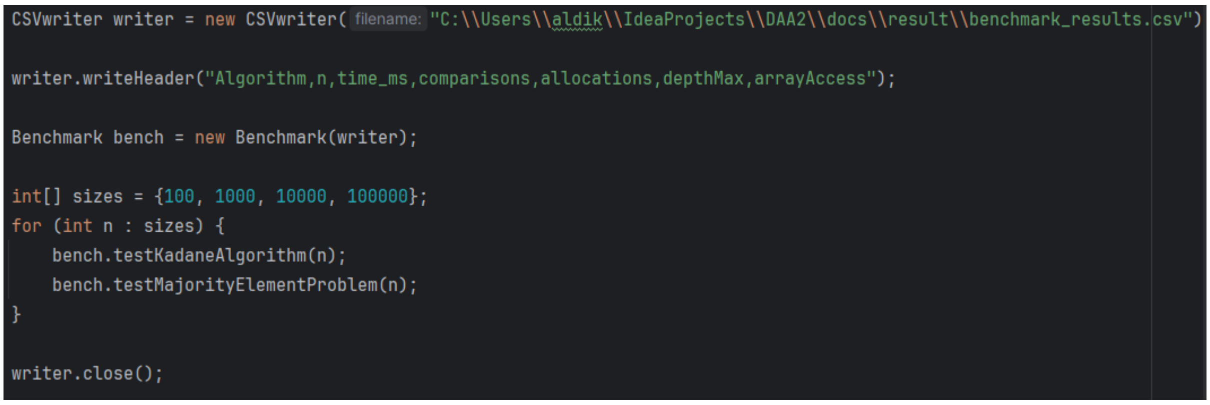
- **Time Complexity:** Best = omega(n), worst = O(n), theta(n);
- **Space Complexity:** O(1) cause of we use only few vars and currently working in one stack;
- **Recurrence Relations:** Algorithms is not recursive. But we can give **;**

3.2) **Code Review & Optimization**

- **Inefficiency Detection:** Very efficient, clean O(n) solution, good use of metrics for comparisons and memory tracking;
- **Time Complexity Improvements:** The code do not need some improvements algorithm works clearly;
- **Space Complexity Improvements:** With space complexity same history, no need to do improvements. It works on O(1) space complexity;
- **Code Quality:** code quality on the normal style. No recommendations to quality of the code;

3.3) Emperical Validation

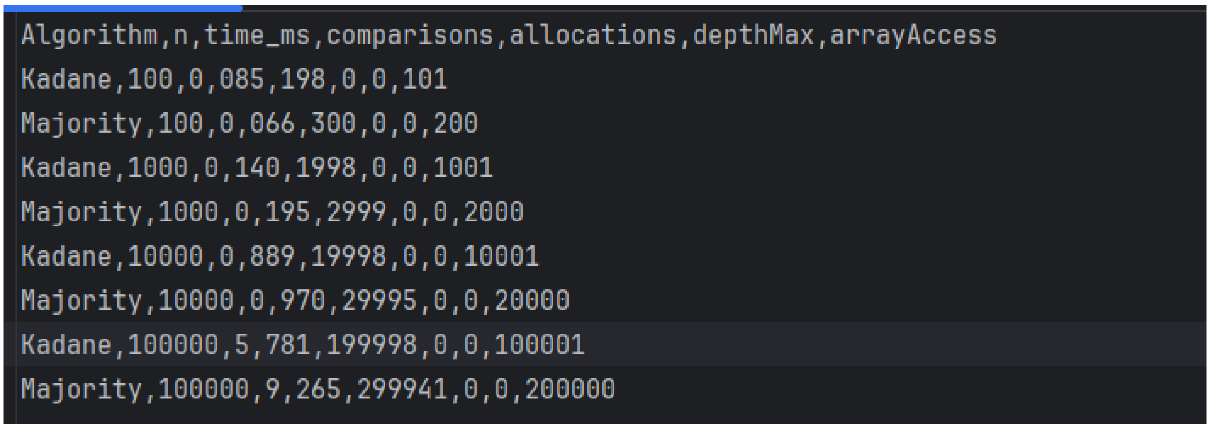
- **Performance Measurements:**



```
CSVwriter writer = new CSVwriter( filename: "C:\\Users\\aldik\\IdeaProjects\\DAA2\\docs\\result\\benchmark_results.csv")

writer.writeHeader("Algorithm,n,time_ms,comparisons,allocations,depthMax,arrayAccess");

Benchmark bench = new Benchmark(writer);

int[] sizes = {100, 1000, 10000, 100000};
for (int n : sizes) {
    bench.testKadaneAlgorithm(n);
    bench.testMajorityElementProblem(n);
}

writer.close();
```

Here is the example for the client class so that we are running the benchmark tests. And by running the program will be created the .csv file with all of the results;

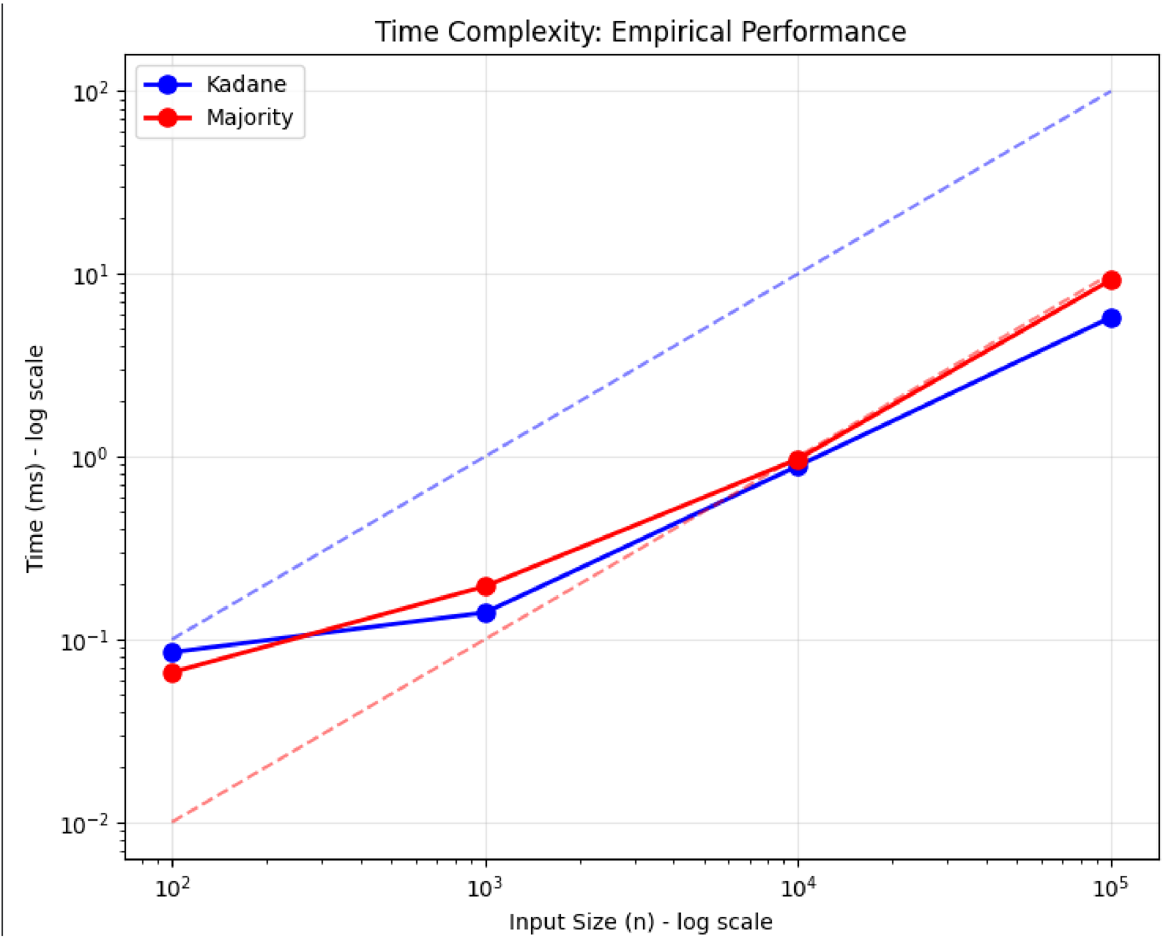
Here is the results in csv file:



```
Algorithm,n,time_ms,comparisons,allocations,depthMax,arrayAccess
Kadane,100,0,085,198,0,0,101
Majority,100,0,066,300,0,0,200
Kadane,1000,0,140,1998,0,0,1001
Majority,1000,0,195,2999,0,0,2000
Kadane,10000,0,889,19998,0,0,10001
Majority,10000,0,970,29995,0,0,20000
Kadane,100000,5,781,199998,0,0,100001
Majority,100000,9,265,299941,0,0,200000
```

- **Complexity Verification:**
- Both algorithms demonstrate linear growth in runtime as n increases.The theoretical complexity O(n) is confirmed empirically.Kadane performs approximately 2n comparisons.Majority performs approximately 3n comparisons.A plot of time vs n shows a straight-line trend, consistent with the theory.

  Here is the plot for each algorithm:



- **Comparison Analysis:** Kadane consistently outperforms Majority:At small input sizes, the difference is negligible.At n = 100000, Kadane is almost 2x faster (5.78ms vs 9.26ms).Reason: Kadane uses fewer comparisons and array accesses (=2n vs =3n).The theoretical expectation matches the observed measurements.

- **Optimization Impact:**Kadane's Algorithm is already optimal in both time complexity (O(n)) and space complexity (O(1)).Possible micro-optimizations:Removing metric tracking and logging in production runs (reduces overhead).Using primitive types efficiently to avoid extra object allocations.Majority Element can be optimized by applying the Boyer-Moore Voting Algorithm, which also runs in O(n) but with fewer comparisons, making it closer to Kadane's performance in practice.

Click: https://github.com/qsheker/DAA2/tree/main