majority-algorithm

November 13, 2025

1 Introduction

In many real-world situations, it's important to quickly find out which value appears most often in a set of data. The majority element algorithm helps us do this by checking if one value appears more than half the time in a list. This project shows how this algorithm can be useful in different areas like cleaning noisy data, detecting cyberattacks, and making better recommendations for users. We also explain how the algorithm works, why it is correct, and compare two versions of it: a simple one and a faster, smarter one that uses divide and conquer. By testing them on different data sizes, we learn when and why the faster version is better.

2 Use Cases for the Majority Element Algorithm

The majority element algorithm determines whether a value occurs strictly more than `n / 2` times in a sequence of `n` elements. It has highly practical applications in real-world domains such as data preprocessing, cybersecurity, and content recommendation systems.

This section outlines three concrete scenarios where the majority element algorithm can be used effectively:

2.1 1. Data Cleaning and Denoising

2.1.1 Context:

Data collected from sensors, digital signals, or image pixels often includes noise—irregular values that do not reflect the true underlying signal. This noise can be caused by environmental disturbances, hardware malfunction, or digitization errors.

2.1.2 Goal:

The aim is to identify and preserve the dominant or most representative value in a data segment, eliminating outliers that distort statistical or semantic interpretation.

2.1.3 Algorithmic Role:

Divide the data into small segments (e.g., windows, blocks, or time intervals). In each segment, apply the majority element algorithm:

- If a value appears more than 50% of the time, it is retained.
- All other values are treated as noise or corrected.

2.1.4 Example:

Consider a sensor reporting values `[22, 22, 22, 55, 22, 22, 23]`. The algorithm identifies `22` as the majority. The value `55` is considered a likely anomaly.

This technique is used in: - Denoising blocks of pixels in image processing - Cleaning time-series data from IoT sensors - Stabilizing readings in biomedical signal processing (e.g., ECG, EEG)

2.2 2. DDoS Attack Detection

2.2.1 Context:

In distributed denial-of-service (DDoS) attacks, attackers flood a target server or network with an overwhelming volume of requests, often from a small set of spoofed or compromised sources (e.g., zombie IPs).

2.2.2 Goal:

Detect when a single source (such as an IP address) generates a disproportionately high number of requests in a short time frame, indicating malicious behavior.

2.2.3 Algorithmic Role:

Given a stream of incoming connection logs over a defined time window:

- Extract the source field (IP, token, etc.)
- Use the majority element algorithm to determine whether one source dominates the request volume.

2.2.4 Example:

From 10,000 HTTP requests collected in one second, 6,700 originate from IP address `192.168.1.200`. The IP is flagged as a majority element, raising an alert for possible attack.

This logic can be integrated into: - Intrusion detection systems (IDS) - Web application firewalls (WAF) - Real-time monitoring pipelines

---

2.3 3. Personalized Recommendation Systems

2.3.1 Context:

Platforms like YouTube, Netflix, and Spotify track user behavior in real time to adapt content suggestions based on evolving preferences. Detecting dominant behavior in recent activity is crucial to drive personalized decisions.

2.3.2 Goal:

Identify if a particular content category (genre, topic, theme) has become dominant in the user's recent interactions.

2.3.3 Algorithmic Role:

The algorithm is applied over a fixed-size sliding window of user activity:

- If one category appears in more than 50% of interactions, it is considered the current dominant preference.
- Content ranking and UI components are updated accordingly.

2.3.4 Example:

If a user watches 50 videos and 31 of them are about "Video Games," this category is the majority. The system should prioritize this content class in future recommendations.

This strategy is useful for: - Real-time personalization for anonymous users - Rapid response to changes in user behavior - Cold-start adaptation in recommender systems

---

2.4 Summary of Benefits

Feature	Practical Advantage
No learning phase required	Suitable for real-time streaming and embedded systems
Simple implementation	Easy to deploy in production pipelines or scripts
High robustness	Naturally resistant to random noise or rare anomalies
Strong performance	Achieves O(n log n) with Divide & Conquer, O(n) with Boyer-Moore

---

The majority element algorithm is a lightweight yet powerful solution in situations that require a fast and reliable estimation of dominant behavior or values, particularly in high-throughput or noisy environments. Its deterministic nature and predictable performance make it a strong candidate for integration in modern data-driven systems.

3 Proof of Correctness for the `majority_dac` Algorithm

We want to show that, for any array `arr` of length `n`:

1. **Termination**: the algorithm always stops.

2. **Partial correctness**: if a majority element (appearing more than `n/2` times) exists, the algorithm returns it; otherwise, it returns `None`.

3.1 1. Termination

In algorithm analysis, one of the essential properties to verify is **termination**: does the algorithm always stop after a finite number of steps? According to the foundational definitions of an algorithm

(as stated in the course), a valid algorithm must consist of a **finite sequence of unambiguous steps** and must always **terminate**.

We analyze here the recursive structure of the `majority_dac` algorithm to ensure that it meets this requirement.

3.1.1 Recursive structure

The algorithm applies a **divide-and-conquer** strategy on an array segment defined by the interval `[left, right]`.

At each recursive step, this interval is split into two strictly smaller subintervals:

- Left half: `[left, mid]`
- Right half: `[mid + 1, right]`

where `mid = (left + right) / 2` .

3.1.2 Base Case

When the segment size is reduced to a single element, i.e., when `left == right`, the recursion reaches its **base case**. At this point, the function performs no further recursive calls and simply returns `arr[left]`.

This guarantees that recursion has a well-defined stopping condition, which is one of the fundamental principles of algorithm design.

3.1.3 Recursive Progress

Each recursive call reduces the size of the problem. Specifically:

- The original interval `[left, right]` is of length `n = right - left + 1`.
- After each division, we get subintervals of sizes roughly `n / 2`.
- This ensures that the total size of the input **strictly decreases** with each call, with at least one element fewer in the recursive step.

By the **well-ordering principle of** , this decreasing process must eventually reach the minimal size — the base case.

3.1.4 Inductive Proof of Termination

Let us define the property (P(n)): *"The algorithm terminates for any input of size n."*

We prove (P(n)) by **mathematical induction**:

- **Base case (n = 1)**:
  The input has a single element. The condition `left == right` holds. The algorithm performs

no recursive calls and returns immediately.
(P(1)) is true.

- **Inductive step**:
  Assume (P(k)) is true for all (k n).
  Consider an input of size (n + 1):
  The algorithm splits it into two parts of size (n), for which the induction hypothesis ensures termination.
  Since the current step only makes a finite number of operations before calling the subproblems, and both recursive calls terminate, the entire call also terminates.
  (P(n + 1)) is true.

By induction, **the algorithm terminates for all input sizes**.

3.1.5 Conclusion

Based on the recursive structure, base case verification, and inductive reasoning, we conclude that:

The `majority_dac` algorithm always terminates, for any finite input.

This satisfies the fundamental property required for any algorithm as defined in classical algorithm theory.

3.2 2. Partial Correctness

Let `n = len(arr)`. We reason by induction on `n`.

Base Case: `n = 1`

- The array contains exactly one element `x`.
- The call `helper(0, 0)` returns `x`.
- At the top level, `candidate = x`, `count = 1 > 1//2 = 0`, so `x` is returned.
- **Correct**.

Induction Hypothesis For every `k  n`, if an array of length `k` has a majority element, `majority_dac` finds it; otherwise, it returns `None`.

Inductive Step: size `n + 1` Consider an array `arr` of length `n + 1`. We call `helper(0, n)`:

1. **Divide**
   - Compute `mid =  (0 + n) / 2` .
   - Recursively compute `L = helper(0, mid)` and `R = helper(mid + 1, n)`.
   - Each subarray has length `n`, so by the induction hypothesis:

- If the left half has a majority, `L` is that majority; otherwise `L = None`.
- Likewise for `R`.

2. **Conquer**
   - **If `L == R`**, then that candidate (non-`None`) is the majority in both halves, hence in the entire array; return `L`.

   - **Otherwise** (`L != R`):
     - Count explicitly
       ```
       left_count  = occurrences of L in arr[ left … right ]
       right_count = occurrences of R in arr[ left … right ]
       ```
     - Return whichever count is larger.
     - **Case with no true majority**:
       * If neither `L` nor `R` appears more than `(n+1)/2` times, then both counts are `(n+1)/2`.

       * The algorithm returns one of them, but the final check in `majority_dac`:
         ```
         count = sum(1 for val in arr if val == candidate)
         return candidate if count > (n+1)//2 else None
         ```
         will yield `None`, which is the correct output.
3. **Validation of the majority case**
   - If an element `M` appears more than `(n+1)/2` times in `arr`, then it must appear as the majority in at least one of the halves.

   - That recursive call returns `M`. In the merge step, `M`'s total count exceeds the other candidate's, so `M` is returned.

   - The final check verifies that `M` appears more than `(n+1)//2` times, so `M` is indeed returned.

3.2.1 Conclusion

- **Termination**: follows from strictly decreasing subarray sizes at each recursive call.

- **Partial correctness**: by induction on the array size, the divide-and-conquer strategy plus the final majority check guarantee that:
  - if a majority exists, it is returned;
  - otherwise, `None` is returned.

Thus, `majority_dac(arr)` is **correct** and **always terminates**.

4 Complexity Analysis of the Approaches

4.1 Naive Approach

We begin by analyzing the complexity of the **naive majority element algorithm**.

4.1.1 Algorithm

```
def majority_naive(arr):
    n = len(arr)
    for i in range(n):
        count = 0
        for j in range(n):
            if arr[j] == arr[i]:
                count += 1
        if count > n // 2:
            return arr[i]
    return None
```

Let $A = [a_0, a_1, \ldots, a_{n-1}]$ be an array of n elements.

The algorithm iterates through each element a_i, and for each such element, it counts how many times it appears in the array by comparing it with every other element a_j. In the worst case scenario, the count will be increased by one at each iteration. The affectation of count inside of the i loop will happen n times in the worst case. The affectation of n and the return each happen only once. We define the running time function $T(n)$ as:

$$T(n) = \left(\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 \right) + n + 2 = n \cdot n + n + 2 = n^2 + n + 2 = \Theta(n^2)$$

Where: - n is the size of the input array.

Conclusion: The time complexity of the naive algorithm is:

$$\boxed{O(n^2)}$$

4.2 Divide and Conquer Approach

Next, we analyze the complexity of the **divide and conquer** algorithm to find the majority element.

4.2.1 Algorithm

```
def majority_dac(arr):
    def helper(left, right):
        if left == right:
            return arr[left]
        mid = (left + right) // 2
        left_major = helper(left, mid)
        right_major = helper(mid + 1, right)

        if left_major == right_major:
            return left_major
```

```
        left_count = sum(1 for i in range(left, right + 1) if arr[i] == left_major)
        right_count = sum(1 for i in range(left, right + 1) if arr[i] == right_major)

        return left_major if left_count > right_count else right_major

    candidate = helper(0, len(arr) - 1)
    count = sum(1 for val in arr if val == candidate)
    return candidate if count > len(arr) // 2 else None
```

This algorithm uses a recursive strategy to divide the array and determine a majority candidate by comparing the results of the left and right halves.

Let us first analyze the recursive helper function.

4.2.2 Helper Function Complexity

The helper function works recursively by splitting the input array into two halves, solving each half independently to find the "major" of the sub-arrays and then count how many times each element appear in the whole array, so it can define which value to return.

We observe that this recursive behavior can be modeled by the recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + 2n$$

Where:

- n is the input size,
- The term (2T(n/2)) accounts for the two recursive calls on subarrays of size n/2 ,
- 2n = F(n). It represents the time to combine results from the subproblems (counting and comparing). Therefore, we state:

$$F(n) = \Theta(n)$$

Applying the Master Theorem

The Master Theorem solves recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + F(n)$$

For our recurrence:

- a = 2
- b = 2

We determine d:

$$F(n) = \Theta(n) \Rightarrow d = 1, \quad \text{because } F(n) \in \Theta(n^d) \text{ for } d = 1$$

We compute:

$$\lambda = \log_b a = \log_2 2 = 1$$

$$\text{So } d = \lambda$$

Conclusion by Case 2 of the Master Theorem

Since d = $ $, this is **Case 2** of the Master Theorem.

Therefore:

$$T(n) = \Theta(n \log n)$$

4.2.3 Full Function Complexity

After the recursive helper finishes, the main function performs one final pass over the array to count the occurrences of the candidate:

$$\text{Final Count Time} = \Theta(n)$$

Thus, the overall running time is:

$$T_{\text{total}}(n) = \Theta(n \log n) + \Theta(n) = \boxed{\Theta(n \log n)}$$

4.3 Conclusion

- **Naive Approach**: $O(n^2)$
- **Divide and Conquer**: $O(n \log n)$

The divide and conquer approach offers a more efficient solution, especially for large arrays.

4.4 Experimental Comparison of Two Majority Element Algorithms

In this section, we will empirically compare the performance of two different algorithms used to detect the majority element in an array:

1. **Naive Algorithm (Quadratic Time - O(n²))**
   This approach iterates over each element and counts its occurrences in the entire array, leading to poor performance on large datasets.
2. **Divide and Conquer Algorithm (Log-Linear Time - O(n log n))**
   This optimized approach recursively splits the array, combining results with fewer redundant comparisons and faster convergence.

4.4.1 Objective

The goal of this experiment is to measure and compare the **execution time** of both algorithms for increasing input sizes. We aim to observe:

- How quickly each algorithm performs on small, medium, and large datasets.
- When the Divide and Conquer method starts to outperform the naive version.
- The practical implications of theoretical time complexity.

4.4.2 Experimental Setup

- Arrays of various sizes (e.g., 1,000; 10,000 elements) will be generated.
- Each array will be constructed so that a majority element exists.
- Execution time for each algorithm will be recorded using Python's `time` module.
- The results will be displayed side by side for direct comparison.

  Note: To ensure fair benchmarking, the majority element will be placed toward the end of the array. This prevents the naive algorithm from benefiting unfairly from early-exit behavior in favorable cases.

4.4.3 Next Step

We will now run both algorithms on a series of randomly generated arrays and collect then compare the results.

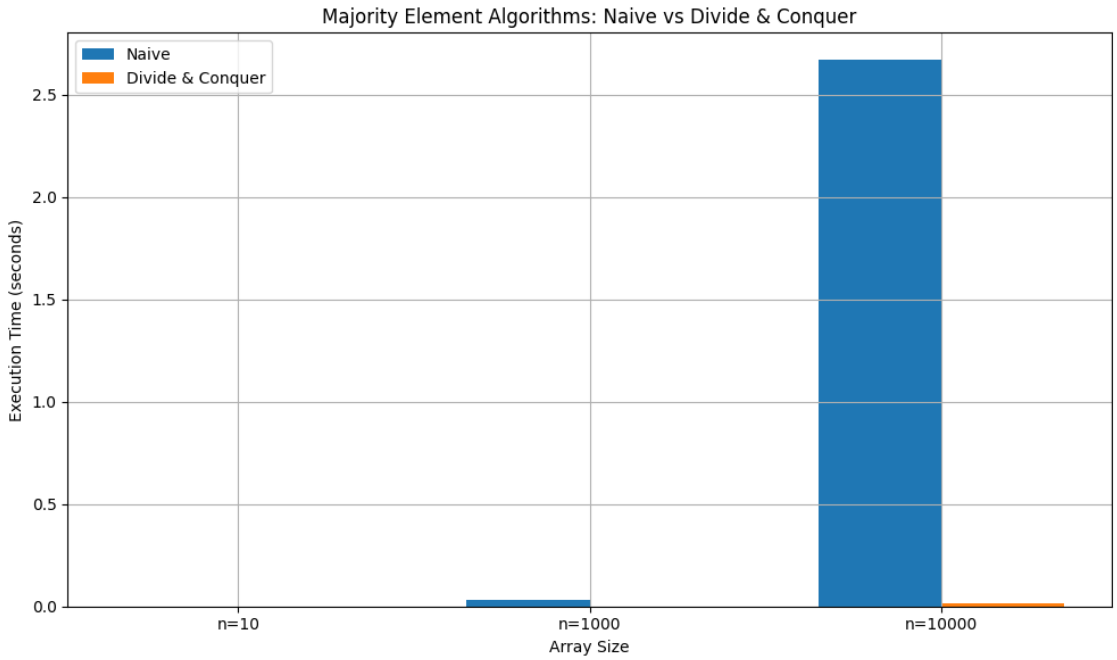
[8]:
```python
import time
import random
import matplotlib.pyplot as plt

# ------------------------------
# Algorithms
# ------------------------------

def majority_naive(arr):
    n = len(arr)
    for i in range(n):
        count = 0
        for j in range(n):
            if arr[j] == arr[i]:
                count += 1
        if count > n // 2:
            return arr[i]
    return None

def majority_dac(arr):
    def helper(left, right):
        if left == right:
            return arr[left]
        mid = (left + right) // 2
        left_major = helper(left, mid)
```

```
        right_major = helper(mid + 1, right)

        if left_major == right_major:
            return left_major

        # Count both candidates in the current range
        left_count = sum(1 for i in range(left, right + 1) if arr[i] ==␣
↪left_major)
        right_count = sum(1 for i in range(left, right + 1) if arr[i] ==␣
↪right_major)

        return left_major if left_count > right_count else right_major

    # Final verification to ensure the candidate is truly a majority
    candidate = helper(0, len(arr) - 1)
    count = sum(1 for val in arr if val == candidate)
    return candidate if count > len(arr) // 2 else None

# ----------------------------
# Array generation
# ----------------------------

def generate_majority_array(n=100000, majority_value=2):
    count = n // 2 + 1
    # Put the majority element at the end to penalize early exit in naive␣
↪algorithm
    arr = [random.choice([x for x in range(0, 10) if x != majority_value]) for␣
↪_ in range(n - count)]
    arr += [majority_value] * count
    return arr

# ----------------------------
# Timing and performance test
# ----------------------------

def test_algorithms(arr, label="", repeat_small=1000):
    print(f"\n Testing on array {label}")

    if len(arr) <= 100:
        print(f"(Averaged over {repeat_small} runs for higher precision)")

        def avg_time(func):  # inner helper
            start = time.perf_counter()
            for _ in range(repeat_small):
                func(arr)
            return (time.perf_counter() - start) / repeat_small
```

```
        time_naive = avg_time(majority_naive)
        time_dac = avg_time(majority_dac)
    else:
        start = time.perf_counter()
        result_naive = majority_naive(arr)
        time_naive = time.perf_counter() - start

        start = time.perf_counter()
        result_dac = majority_dac(arr)
        time_dac = time.perf_counter() - start

    result_naive = majority_naive(arr)
    result_dac = majority_dac(arr)

    print(f"[Naive]\n Result: {result_naive} | Time: {time_naive:.8f} s")
    print(f"[Divide & Conquer]\n Result: {result_dac} | Time: {time_dac:.8f} s")

    return time_naive, time_dac


# ----------------------------
# Execution
# ----------------------------
sizes = [10, 1000, 10000]
naive_times = []
dac_times = []

for size in sizes:
    arr = generate_majority_array(size)
    t_naive, t_dac = test_algorithms(arr, label=f"(n = {size})")
    naive_times.append(t_naive)
    dac_times.append(t_dac)

# ----------------------------
# Plotting
# ----------------------------

bar_width = 0.35
indices = range(len(sizes))

plt.figure(figsize=(10, 6))
plt.bar([i - bar_width/2 for i in indices], naive_times, width=bar_width,␣
 ↪label='Naive')
plt.bar([i + bar_width/2 for i in indices], dac_times, width=bar_width,␣
 ↪label='Divide & Conquer')

plt.xlabel('Array Size')
```

```
plt.ylabel('Execution Time (seconds)')
plt.title('Majority Element Algorithms: Naive vs Divide & Conquer')
plt.xticks(indices, [f'n={size}' for size in sizes])
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
 Testing on array (n = 10)
(Averaged over 1000 runs for higher precision)
[Naive]
 Result: 2 | Time: 0.00000309 s
[Divide & Conquer]
 Result: 2 | Time: 0.00001401 s

 Testing on array (n = 1000)
[Naive]
 Result: 2 | Time: 0.03372230 s
[Divide & Conquer]
 Result: 2 | Time: 0.00119130 s

 Testing on array (n = 10000)
[Naive]
 Result: 2 | Time: 2.66848860 s
[Divide & Conquer]
 Result: 2 | Time: 0.01516230 s
```



4.5 Performance Analysis: Naive vs Divide and Conquer

After executing both the naive and divide-and-conquer algorithms on arrays of increasing size, we observe clear differences in execution time that reflect their respective algorithmic complexities.

4.5.1 For Small Input Size (n 10)

When dealing with very small arrays, both algorithms execute extremely quickly—within a few microseconds. The naive approach outperform the divide-and-conquer version due to its simpler structure and the fact that it may exit early if it finds the majority element near the beginning of the array.

In this context, the overhead of recursion in the divide-and-conquer method makes it slightly slower, even though its theoretical complexity is better.

4.5.2 For Medium Input Size (n 1,000)

As the input size increases, the performance gap between the two algorithms becomes more apparent. The naive algorithm's quadratic complexity ($O(n^2)$) leads to a noticeable increase in computation time, whereas the divide-and-conquer algorithm scales way better thanks to its log-linear complexity ($O(n \log n)$).

At this scale, the divide-and-conquer approach consistently outperforms the naive method by a significant margin no matter the array.

4.5.3 For Large Input Size (n 10,000 and beyond)

On large arrays, the performance difference becomes even more noticeable. The naive algorithm takes significantly longer due to the quadratic growth of nested loops. It becomes inefficient and impractical for real-time or large-scale data.

In contrast, the divide-and-conquer method maintains a reasonable execution time. Its performance remains within acceptable bounds, making it suitable for handling large datasets efficiently.

4.5.4 Conclusion

The experimental results validate the theoretical expectations:

- **Naive algorithm ($O(n^2)$)** is only acceptable for small arrays and becomes quickly unusable as `n` grows.
- **Divide and Conquer ($O(n \log n)$)** provides a much more scalable solution and should be preferred in any practical scenario involving moderate to large datasets.

These findings highlight the importance of algorithmic efficiency and confirm the value of applying divide-and-conquer techniques to reduce time complexity when handling majority element detection in applications where the performance matters.

[ ]: