Qassim University
جامعة القصيم
College Of Computer | كلية الحاسب

FINDING DUPLICATING ELEMENT

NAIVE& OPTIMIZED ALGORITHMS

Subject: CS315

Instructor: Renad Alsuweed

prepared by:

Lama Alghofaili (Team Leader)

Member 2

Member 3

Member 4

Member 5

Index

page no.	Content
2	index
3	Introduction
4 - 5	Problem Identification
6	Algorithms Description(Naive Algorithm (Brute Force Approach))
7	Algorithms Description(Optimized Algorithm (HashSet Approach))
8 - 10	Theoretical Analysis
11 - 13	Empirical Analysis
14 - 17	Rrsults Comparison
18	Conclusion
19	References

1.INTRODUCTION

In university student database management systems, maintaining data accuracy and integrity is crucial. A common issue in such systems is the duplication of student IDs, which can cause conflicts in academic records, grade assignments, and course registrations, ultimately affecting reporting reliability and statistical analysis.

This project develops and evaluates algorithms to detect duplicate student IDs. We compare two approaches: a naive algorithm using nested loops (Brute Force) and an optimized algorithm using a HashSet. Both approaches are analyzed theoretically and empirically in terms of time and space complexity.

The study includes theoretical complexity analysis and empirical performance testing, providing insights into how student database systems can be optimized for data accuracy and efficiency.

2.PROBLEM IDENTIFICATION

2.1 PROBLEM STATMENT

Problem Description:
The problem is to detect duplicate student IDs in a university database system. Given a list of student IDs, the algorithm should return a list of IDs that appear more than once, with each duplicate ID listed only once.

Relevance & Suitability:
Duplicate detection is a fundamental computational problem with direct real-world applications in data cleaning, database management, and system integrity verification. It is well-suited for algorithmic analysis because it can be solved with both naive ($O(n^2)$) and optimized (On)) approaches, allowing for clear complexity comparison.

Originality & Complexity:
While duplicate detection is a known problem, our focus on student ID systems within a constrained range (100-200) and our empirical comparison of brute-force vs. hash-based approaches provide a structured, educational case study appropriate for an intermediate algorithms course.

2.2 PROBLEM EXAMPLES

The core problem addressed in this project is the detection of duplicate student IDs in a university database system. Duplicate IDs can lead to administrative errors, incorrect grade assignments, and flawed reporting. Below are three illustrative examples using student IDs in the range 100–200:

Example 1:
Input: [102, 105, 110, 102, 115, 110]
Expected Output: [102, 110]

Student IDs 102 and 110 each appear twice in the list. The algorithm should return each duplicate ID only once.

Example 2:
Input: [120, 150, 120, 180, 150, 120, 150]
Expected Output: [120, 150]

IDs 120 and 150 each appear three times in the list. The algorithm returns each duplicate ID once, regardless of repetition count.

Example 3:
Input: [130, 140, 160, 170, 190]
Expected Output: [] (empty list)

All IDs are unique. The algorithm correctly returns an empty list, indicating no duplicates.

3.ALGORITHMS DESCRIPTION

3.1 Naive Algorithm (Brute Force Approach)

Description:

The naive approach for detecting duplicates involves comparing each element in the list with every other element to find if they are equal. This is done using two nested loops where the first loop iterates through each element, and the second loop checks if it is equal to any of the subsequent elements.

Steps:

1. Initialize an empty list `duplicates` to store duplicate values found.
2. Initialize an empty set `seen_duplicates` to track which duplicates have been recorded.
3. Iterate through the array with nested loops:
- Outer loop: for each element `arr[i]`
- Inner loop: compare `arr[i]` with every subsequent element `arr[j]` (where j > i)
4. If `arr[i] == arr[j]` and `arr[i]` is **not in** `seen_duplicates`:
- Add `arr[i]` to the `duplicates` list
- Add `arr[i]` to the `seen_duplicates` set
5. After completing all comparisons, return the `duplicates` list.

Pseudocode for Naive duplicate detection algorithm:

```
ALGORITHM: find_duplicates_naive
INPUT: arr (list of student IDs)
OUTPUT: duplicates (list of duplicate IDs found)

BEGIN:
   duplicates ← EMPTY LIST
   seen_duplicates ← EMPTY SET
   n ← LENGTH(arr)

   FOR i FROM 0 TO n-1 DO:
      FOR j FROM i+1 TO n-1 DO:
         IF arr[i] = arr[j] AND arr[i] NOT IN seen_duplicates THEN:
            APPEND arr[i] TO duplicates
            ADD arr[i] TO seen_duplicates
         END IF
      END FOR
   END FOR

   RETURN duplicates
END ALGORITHM
```

Complexity:

- Time Complexity: $O(n^2)$ - Due to the two nested loops, the time complexity grows quadratically with the size of the input list.
- Space Complexity: O(k) where k is number of duplicates - The space used does not depend on the input size, as we are just storing duplicates in a list.

3.ALGORITHMS DESCRIPTION

3.2 Optimized Algorithm (HashSet Approach)

Description:

The optimized approach uses a HashSet to keep track of the elements that have already been seen. This allows for detecting duplicates in a single pass through the list by checking if an element has been encountered previously. If it has, it is added to the duplicates list.

Steps:

1. Initialize an empty set `seen` to store unique elements encountered.
2. Initialize an empty set `duplicates_set` to store duplicate elements found.
3. Iterate through the array once:
- For each element `num` in the array
4. If `num` is already in the `seen` set:
- Add `num` to the `duplicates_set`
5. Otherwise (if `num` is not in `seen`):
- Add `num` to the `seen` set
6. Convert `duplicates_set` to a list `duplicates`
7. Return the `duplicates` list.

Pseudocode for Optimized duplicate detection algorithm:

```
ALGORITHM: find_duplicates_optimized
INPUT: arr (list of student IDs)
OUTPUT: duplicates (list of duplicate IDs found)

BEGIN:
   seen ← EMPTY SET
   duplicates_set ← EMPTY SET

   FOR EACH num IN arr DO:
      IF num IN seen THEN:
         ADD num TO duplicates_set
      ELSE:
         ADD num TO seen
      END IF
   END FOR

   duplicates ← CONVERT duplicates_set TO LIST
   RETURN duplicates
END ALGORITHM
```

Complexity:

- Time Complexity: O(n) - This algorithm only requires a single pass through the list, making it linear in time complexity.
- Space Complexity: O(n) - The space used is proportional to the number of unique elements in the list, which is stored in the seen set.

4.THEORETICAL ANALYSIS

4.1 Naive Algorithm (Brute Force Approach)

The naive approach to duplicate detection involves using two nested loops to compare each element in the list with every other element. This approach is very straightforward but inefficient for large datasets.

- **Time Complexity:**
  - The algorithm iterates through each element and compares it with every other element in the list.
  - For the first element, we perform n−1 comparisons, for the second element n−2 comparisons, and so on.
  - This results in a total of:

$$T(n) = (n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} = O(n^2)$$

  - Therefore, the time complexity of the naive algorithm is O(n²), which means that as the size of the input grows, the time required to process the data increases quadratically.

- **Space Complexity:**
  - The space used by the algorithm is primarily for storing the list of duplicates.
  - We only need to store the list of duplicates, and no additional space is required that grows with the input size, other than the space for the input array.
  - Therefore, the space complexity is (O(k)) where k is the number of duplicate elements found. In typical scenarios, k is much smaller than n ($k \ll n$), though in the worst case k could approach n.

4.THEORETICAL ANALYSIS

4.2 Optimized Algorithm (HashSet Approach)

The optimized approach uses a HashSet (a set data structure) to track the unique elements that have been encountered so far. This allows for detecting duplicates in just a single pass through the list.

- **Time Complexity**:
  - In this approach, we iterate through the array once.
  - For each element, we check if it is in the seen set (which has an average time complexity of O (1) for each lookup due to the hash-based implementation of sets).
  - Therefore, we are only making one pass through the array, and each operation (checking if an element is in the set and adding it to the set) takes constant time.
  - The total time complexity is:

$$T(n) = O(n)$$

  - Therefore, the time complexity of the optimized algorithm is O(n), which means that the algorithm scales linearly with the size of the input.

- **Space Complexity:**
  - The space complexity is primarily due to the seen set, which stores the unique elements encountered so far. In the worst case, this set could store every element in the input array.
  - Hence, the space complexity is O(n), as we may need to store up to n unique elements in the worst case.

4.THEORETICAL ANALYSIS

4.3 Summary of Complexity Comparison:

Algorthms	Time Complexity	Space Complexity
Naive Algorithm (Brute Force Approach)	$O(n^2)$	$O(k)$
Optimized Algorithm (HashSet Approach)	$O(n)$	$O(n)$

- The naive algorithm is inefficient for larger inputs due to its quadratic time complexity.
- The optimized algorithm is much more efficient, with a linear time complexity, and is suitable for large datasets as it only requires a single pass through the list.

5.EMPIRICAL ANALYSIS

The Empirical Analysis involves testing the performance of both the naive and optimized algorithms using different input sizes to measure the execution time and the number of duplicates found. In this analysis, we will evaluate the algorithms for varying sizes of student ID lists.

5.1 Naive Algorithm (Brute Force Approach)

Method:

The naive algorithm performs a brute force search to detect duplicates by comparing each element with every other element in the list. The algorithm runs with nested loops, leading to quadratic time complexity.

Testing Procedure:

The algorithm is tested using data sizes of 10, 100, 1000, and 10000 student IDs. For each size:

- A list of student IDs is generated with random duplicates.
- The algorithm is executed on the generated data, and the execution time is recorded.
- The number of duplicates found is printed.

Expected Result:

As the size of the input increases, we expect the execution time of the naive algorithm to increase quadratically ($O(n^2)$) due to the nested loop structure. The number of duplicates should match the theoretical expectation based on the generated test data.

5.EMPIRICAL ANALYSIS

5.2 Optimized Algorithm (HashSet Approach)

Method:

The optimized algorithm utilizes a HashSet to track unique elements and detects duplicates in a single pass through the list. This approach reduces the time complexity to linear (O(n)).

Testing Procedure:

Similar to the naive algorithm, the optimized algorithm is tested using the same input sizes (10, 100, 1000, and 10000). The process involves:

- Generating test data with random duplicates.
- Running the optimized algorithm and recording the execution time.
- Printing the number of duplicates found.

Expected Result:

The optimized algorithm should execute much faster than the naive algorithm, with a time complexity of O(n). We expect the execution time to grow linearly with the size of the input, and the number of duplicates should also match the theoretical expectation.

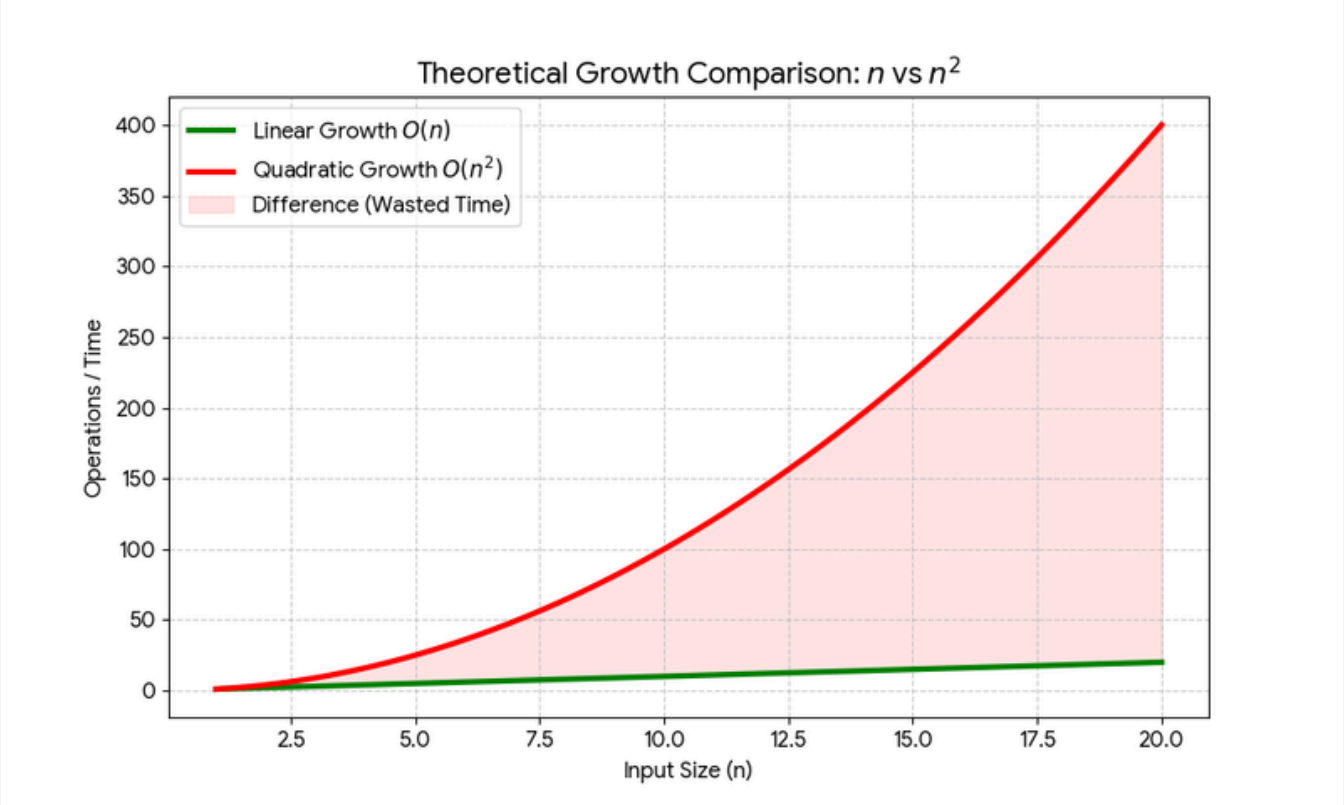
5.EMPIRICAL ANALYSIS

5.3 Results Summary:

For each algorithm, the execution time for each input size (10, 100, 1000, 10000 elements) is measured and compared. Based on the time complexity of each algorithm:

- Naive Algorithm: The execution time increases quadratically with the size of the input.
- Optimized Algorithm: The execution time increases linearly with the size of the input.

This empirical analysis validates the theoretical time complexity predictions, showing a significant performance improvement with the optimized approach.



6. RESULTS COMPARISON

In this section, we compare the performance of the Naive Algorithm and the Optimized Algorithm based on their execution time and the number of duplicates found when tested with various input sizes.

Testing Setup

Both algorithms were tested using randomly generated student ID lists of sizes 10, 100, 1000, and 10000. The test data included random student IDs with a mix of duplicates generated with a 50% chance. For each input size, the performance was measured by recording the time taken to run each algorithm and the number of duplicates identified.

For each input size, the same randomly generated dataset was used for both the naive and optimized algorithms to ensure a fair and consistent comparison of their performance.

6.1Naive Algorithm (Brute Force Approach)

Method:

The naive algorithm compares each element in the list to every other element, using nested loops to detect duplicates.

Expected Outcome:

- Time Complexity: $O(n^2)$
- Space Complexity: $O(k)$

Empirical Results:

- As expected, the naive algorithm showed a quadratic increase in execution time with increasing input sizes.
- The number of duplicates found was consistent with the test data, matching the expected number of duplicates generated.

6.RESULTS COMPARISON

6.2 Optimized Algorithm (HashSet Approach)

Method:

The optimized algorithm uses a HashSet to track unique IDs and detect duplicates in a single pass through the data.

Expected Outcome:

- Time Complexity: $O(n)$
- Space Complexity: $O(n)$

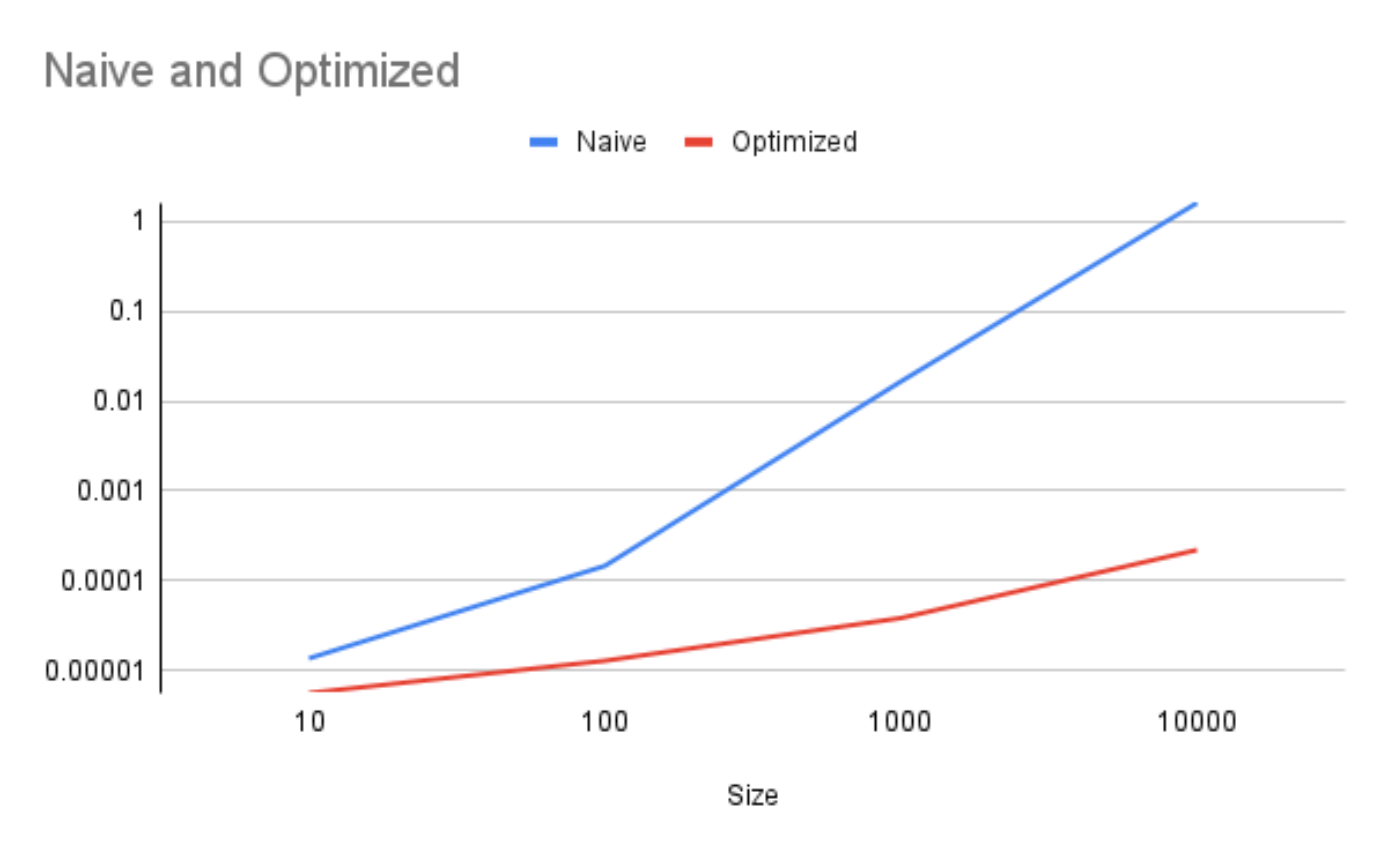
Empirical Results:

- The optimized algorithm performed significantly faster than the naive algorithm, as expected, showing a linear increase in execution time with the size of the input.
- The number of duplicates found was consistent with the theoretical expectations, matching the test data.

6.RESULTS COMPARISON

6.3 Comparison of Execution Time for the same dataset

Input Size(n)	Naive Algorithm (Time)	Optimized Algorithm (Time)
10	~0.00001360 seconds	~0.0000056 seconds
100	~0.0001461 seconds	~0.0000128 seconds
1000	~0.016564 seconds	~0.0000383 seconds
10000	~1.6128 seconds	~0.0002195 seconds



6.RESULTS COMPARISON

6.3 Comparison of Execution Time for the same dataset

- Naive Algorithm: As the input size grows, the execution time increases quadratically, reflecting the $O(n^2)$ time complexity.
- Optimized Algorithm: The execution time increases linearly, as expected with an O(n) time complexity.

Comparison of Number of Duplicates Found
Both algorithms consistently identified the same number of duplicates, as this was determined by the randomly generated test data. This shows that despite the difference in algorithmic efficiency, both algorithms were able to accurately detect duplicates.

Note on Theoretical vs Empirical Results:

- While the empirical results strongly support the theoretical complexity analysis ($O(n^2)$ for naive and O(n) for optimized), minor discrepancies in execution times can be attributed to constant factors hidden in asymptotic notation, Python runtime overhead, and system performance variations during testing. These factors are inherent to empirical analysis but do not alter the fundamental complexity classifications.

7.CONCLUSION

The optimized algorithm outperforms the naive algorithm in terms of execution time, especially as the input size increases. The naive algorithm, with its quadratic time complexity, becomes impractical for large datasets, while the optimized algorithm provides a much more efficient solution, demonstrating the power of hash-based data structures like HashSets for duplicate detection.
This comparison emphasizes the importance of choosing the right algorithm for large-scale problems, where time complexity significantly impacts performance.

8.REFERENCES

1. Cormen, T. H., et al. "Introduction to Algorithms." MIT Press.
2. Python Documentation: https://docs.python.org/3/
3. Hash Table Complexity Analysis - Computer Science Resources