There are multiple ways to find duplicate elements in an array in Java and I have implemented three of them.

**DeDup1**

The first solution is very simple. All I am doing here is looping over array and comparing each element to every element in the new unique array. If the element is not already present in the new unique array, then I add it to the unique array. For doing this, I am using two loops, inner loop and outer loop. In the worst case scenario, if the array has no duplicates, then the second element of the array will be compared to one element of unique array, third element of array is compared to two elements of unique array and so on.

The time complexity will be like

O(2\*1 + 3\*2+4\*3…………………+n\*(n-1)) for the loops

+ O(n) for creating the unique array

The space complexity will be O(n) for the newly created unique array.

Where n is the number of elements in the array.

**Positives**

1. This solution is easy to understand and implement.

2. It does not use any other Collection objects.

3. It is implemented by using only primitive types.

4. The new array maintains the order of the array in the same order it is inserted into the

original array.

**Negatives**

1. This solution has high time complexity

**Use Case**

This solution can be used mainly for academic and demonstration purpose. It is not ideal for use in real world.

**DeDup 2**

Standard way to find duplicate elements from array is by using HashSet data structure. The **Set** type does not allow duplicates. This method takes advantage of this property to eliminate duplicate elements.

If we use the **HashSet** data structure, then the order of the elements in the Set will be unpredictable. So, in order to preserve the ordering of the elements in the Set, I have used **LinkedHashSet.**

This solution has time complexity of

O(n) to iterate over the array once

+ O(n) to add the elements to the Set.

In the worst case, if the array has no duplicates, the space complexity will be

O(n) for storing the Set

Where n is the number of elements in the array.

**Positives**

1. Simple solution with fewer lines of code.
2. It uses one Collection object (Set) to provide the solution.
3. The LinkedHashSet maintains the order of the elements in the order in which they were inserted.
4. The solution has a better time complexity than the DeDup1 solution.

**Negatives**

1. Performance might be little bit lagging due to the use of LinkedHashSet.

LinkedHashSet is implemented as a hash table with a linked list running through it.

Even though the time complexity of basic methods is O(1), it might be bit slower than HashSet.

If ordering of the elements is not a concern, then HashSet can be used for better performance.

**Use Case**

This solution is very effective and can be used in real-time applications. For example, if we have an array of employee IDs who have logged in to a particular application on a given day, and we want to get a distinct list of employees who have logged in to that system on that day. Each employee might have logged in to the system multiple times on the same day.

**DeDup 3**

This solution makes use of the **ArrayList** Collecion object. An ArrayList object is created using the array elements. Then the collection is sorted using Collections.sort in its natural sorting order (ascending order of numbers).

Then looping through the elements in the ArrayList, till the last but one element, each element is compared to its next element to see if its same (identify duplicates). If not, then the element is added to a new unique array. Finally , the last element of the list is added to the unique array.

This solution has time complexity of

O(n) to add the elements to the List.

+ n\*log(n) for Collections.sort

+O(n) for looping through the list for comparing

+ O(n) for creating the new unique array

In the worst case, if the array has no duplicates, the space complexity will be

O(n) for storing the List

O(n) for storing the unique array

Where n is the number of elements in the array.

**Positives**.

1. It uses one Collection object (ArrayList) and one primitive type(int) array to provide the solution.
2. It makes use of the Collections.sort method to sort the elements, which would be simpler than implementing sorting directly on the array.
3. The results are returned in a sorted order.
4. The solution has better time complexity than DeDup1, especially for the looping because it avoids double looping (inner and outer loops)

**Negatives**

1. The solution does not maintain the order of the elements in the order in which they were inserted.
2. The solution has more time complexity than the DeDup2 solution, as it has extra time complexity for sorting, and creating the two structures ArrayList and array. Whereas, DeDup 1 and DeDup2 use one data structure each.
3. This solution has more space complexity for storing the two structures, ArrayList and array.

**Use Case**

This solution is effective and can be used in real-time applications. For example, if we have an array of student Ids who have visited the library on a given day, and we want to get a distinct list of student Ids, in sorted order, who have visited the library on that day . Each student might have visited the library multiple times during the same day.

This solution has more time and space complexity than DeDup2, we can use this solution for applications where space is not a concern. Also, in cases where a sorted result is needed, we can compromise for the added time complexity.