**Algorithm Design**

There are several methods that can be used to identify and return unique items. I’ve chosen to utilize a hashing method that makes use of the unordered\_set container from the STL library. This is the C++ code for the function:

A picture containing text, screenshot, font

Description automatically generated

The function takes in a const reference to the n by m array of book titles. It begins by initializing the 1-D container and reserving enough space for the worst-case scenario, which in this case is when all book titles in the array are unique. Next, the function iterates through each item in the array and inserts it into the unordered\_set data structure. Finally, the unordered\_set is returned to main.

The testing code is added below. The test data consisted of 12 book titles organized into a 3 by 4 array, which contained 10 unique titles and two duplicate titles.

A picture containing text, screenshot

Description automatically generated

The original test data:

A screenshot of a computer

Description automatically generated

And the results of the test:

A screen shot of a computer

Description automatically generated with medium confidence

**Algorithm Analysis**

**Time complexity: O(n \* m)**

**Space complexity: O(n \* m)**

The analysis of this algorithm is somewhat interesting. Starting with the simpler side, the space complexity is O(n \* m). This is true of both the n by m input array and is also true for the 1-D unordered\_set.

The time complexity is less straightforward. Initilization of the unordered\_set is a constant time operation. The reserve function is also constant while the set is empty. By necessity, the function must loop over all rows and all columns in order to process all of the data. The first for loop runs n times, and the second runs m times. In the worst-case, the insert function can be linear with respect to the number of elements currently in the container, but this situation is avoided by the use of the reserve function earlier. Because of this, the insert function is now a constant time operation. And finally, the return command is also constant. This leaves the overall algorithm with a time complexity of O(n \* m).

Showing that this algorithm avoids the worst-case time complexity for the insert function can be done by monitoring the number of buckets in the unordered\_set during operations inside of the for loops. This is the modified test code:

A picture containing text, screenshot, font, software

Description automatically generated

And the results:

A picture containing text, screenshot, font

Description automatically generated

The number of buckets in the set remain unchanged after the reserve command, showing that the function avoided the need to rehash.