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:

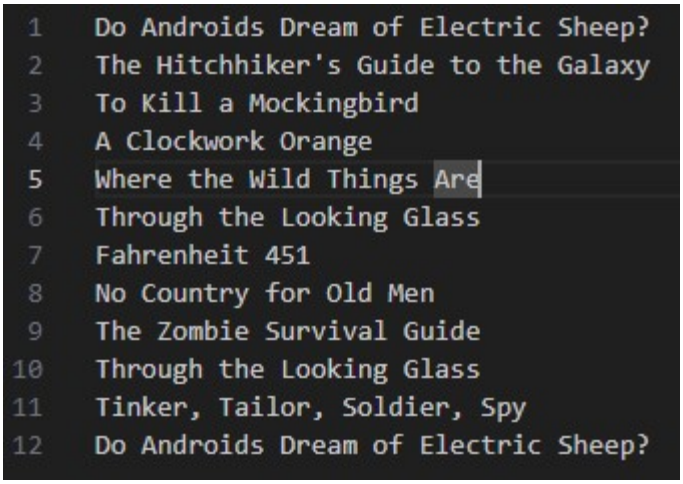
```
unordered_set<string> getUnique(const array<array<string, m>, n> &books) {
    // Initialize set and reserve space
    unordered_set<string> unique_books;
    unique_books.reserve(n * m);

    // Insert book titles into set
    for (int i{}; i < n; ++i) {
        for (int j{}; j < m; ++j) {
            unique_books.insert(books[i][j]);
        }
    }

    return unique_books;
}
```

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 in the container for the worst-case scenario, which 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. Finally, the unordered_set is returned to main.

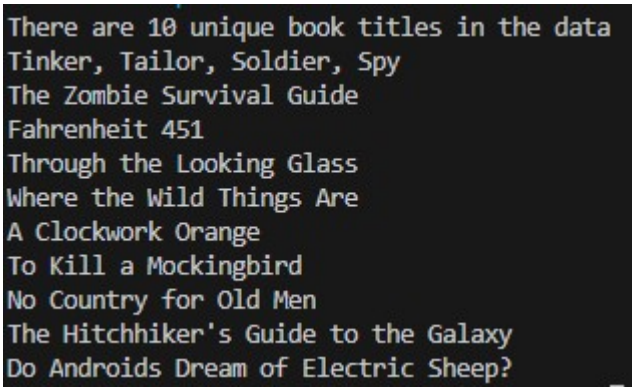
The testing code is added below. The test data consisted of 12 book titles organized into a 3 by 4 array, which contained 10 unique titles and two duplicate titles.

```
int main() {
    // Load book titles into 2D array
    array<array<string, m>, n> books;

    // Fill array with book titles
    ifstream file("p2data.txt");
    string text;
    for (int i{}; i < n; ++i) {
        for (int j{}; j < m; ++j) {
            getline(file, text);
            books[i][j] = text;
        }
    }

    // Hash book titles and return only unique
    unordered_set<string> unique_books = getUnique(books);

    // Print total and titles
    cout << "There are " << unique_books.size() << " unique book titles in the data" << endl;
    for (const auto& book: unique_books) {
        cout << book << endl;
    }
}
```

The original test data:



And the results of the test:



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:

```
unordered_set<string> getUnique(const array<array<string, m>, n> &books) {
    // Initialize set and reserve space
    unordered_set<string> unique_books;
    unique_books.reserve(n * m);

    // Insert book titles into set
    for (int i{}; i < n; ++i) {
        for (int j{}; j < m; ++j) {
            unique_books.insert(books[i][j]);
            cout << unique_books.bucket_count() << endl;
        }
    }

    return unique_books;
}
```

The program was run with a new dataset of the same size, but with no repeats. This is the result:

```
13
13
13
13
13
13
13
13
13
13
13
There are 12 unique book titles in the data
```

The number of buckets in the set remains unchanged after the reserve command, showing that the function avoided the need to rehash and ensuring that the insert function runs in constant time.