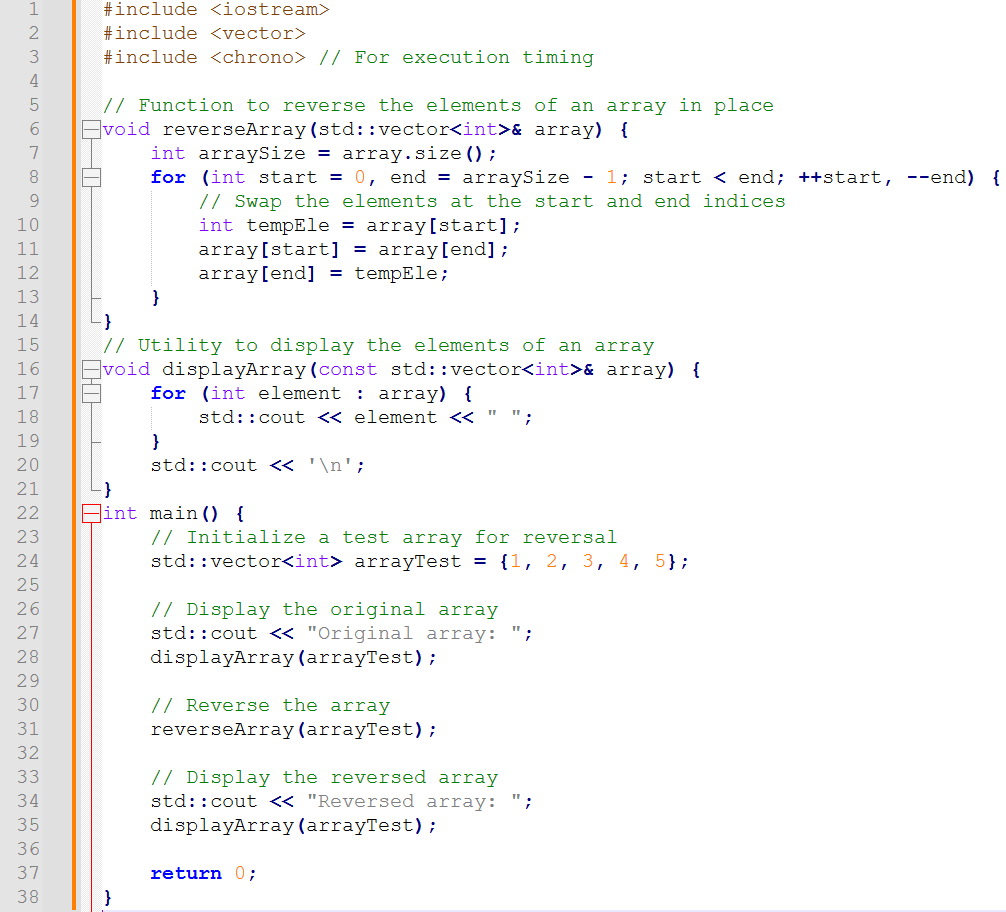
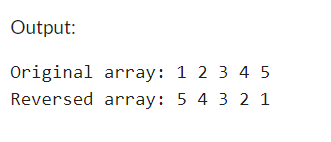
In this assignment I developed an efficient algorithm to reverse the order of elements in an array, specifically for a mobile app that plays audio files backward. The goal is to minimize memory usage by using an "in-place" algorithm, meaning the reversal happens without extra memory allocation beyond the original array. I’ll implement the algorithm, evaluate its time, and space complexity, and then measure its actual runtime on arrays of various sizes. Finally, I’ll visualize how the algorithm's performance changes as the array size increases.

**Part 1: Reverse the Array**

1. **Input**: The function takes a list of numbers as its input.
2. **Logic**: The function swaps numbers from the beginning and end of the list, gradually moving towards the center. It uses a temporary variable to hold a number during each swap.
3. **Output**: Once the function is done, the original list is reversed, and no extra memory is used to create a copy—it all happens right in the same list.





Used one complier to run the above code attaching screenshot output:

Reference: [OneCompiler - Write, run and share code online | Free online compiler with](https://onecompiler.com/) 70+ languages and databases



**Part 2**: **Complexity Analysis**

### **Time Complexity**

1. The function has a loop that runs n/2 times, where n is the size of the array.
2. During each loop iteration:
   1. Two numbers are accessed from the array (one from the start and one from the end).
   2. One temporary variable is used to hold a number temporarily for swapping.
   3. Three assignments are performed: updating the temporary variable and swapping the two elements.

Since all these actions take constant time in each iteration, the overall time is proportional to the number of iterations, which is **n/2**.

However, when analyzing time complexity, we drop constants and focus on the biggest factor affecting runtime. The time complexity simplifies to **O(n)**.

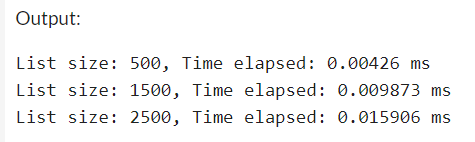
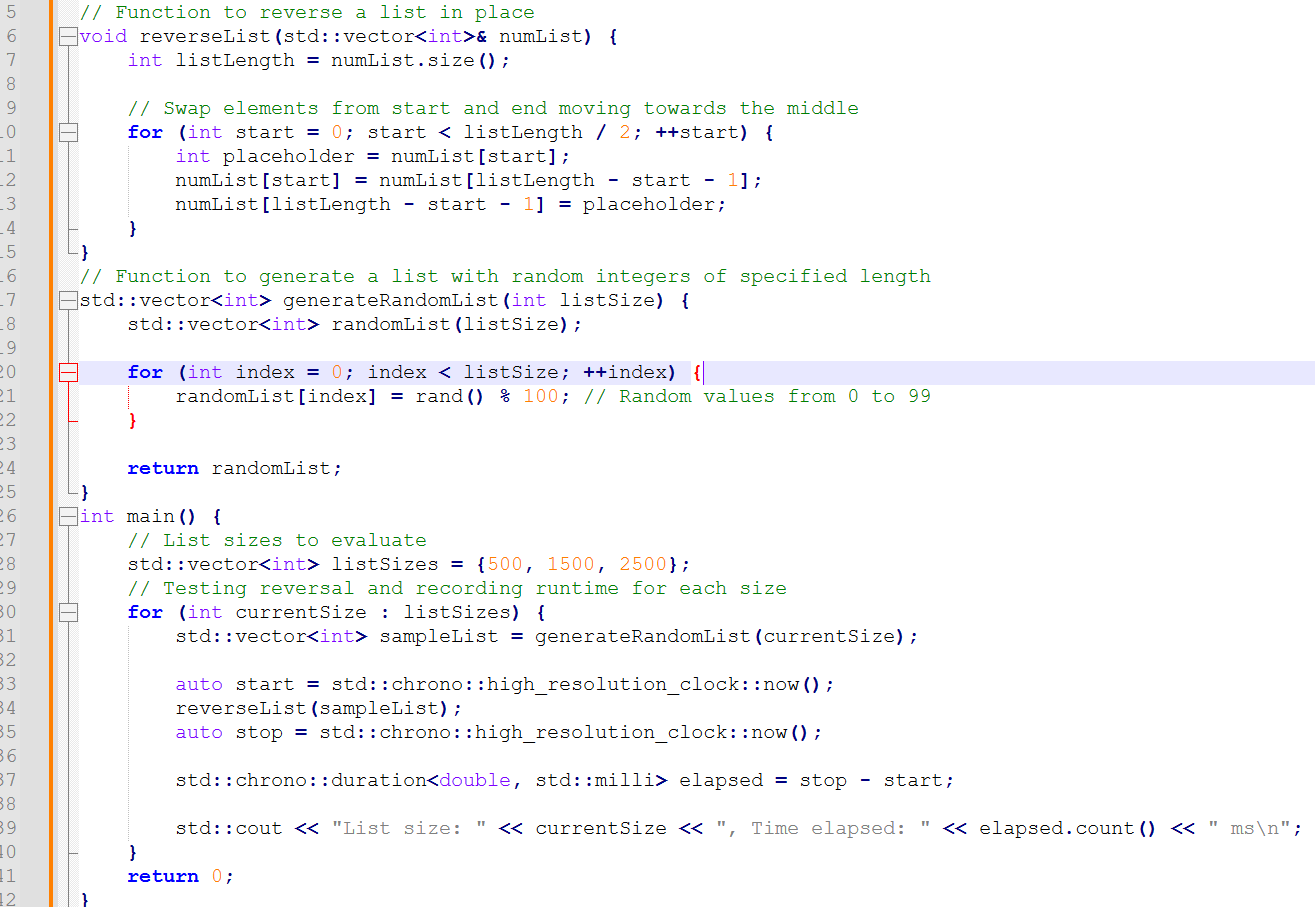
### **Space Complexity (How much memory it uses)**

The function needs:

1. The original array, which holds integers.
2. One temporary variable to help with swapping, using a single memory location.

Adding these together gives **n + 1**, but for large inputs, the "+1" becomes insignificant, so the space complexity is approximately **O(n)**.

Part 3: Measuring Runtime **and Plotting Results**



### **Explanation of the Timing Code**

1. **Input Sizes**: The program tests how well the reversal algorithm performs with arrays of different sizes—500, 1500, and 2500 elements.
2. **Random Data Generation**: The generateRandomList function creates arrays populated with random numbers. This ensures the algorithm is tested with a variety of datasets, simulating real-world use cases.
3. **Measuring Time**: The program utilizes the chrono library to precisely measure the time it takes to reverse each array. This provides clear insights into how the algorithm scales with larger input sizes.

### **Visualizing the Results**

To better understand how the list reversal algorithm performs, I decided to visualize the results using Google Collab. I’ve attached the Jupiter Notebook with this assignment, so you can see how I went about it.

#### **Data Preparation**

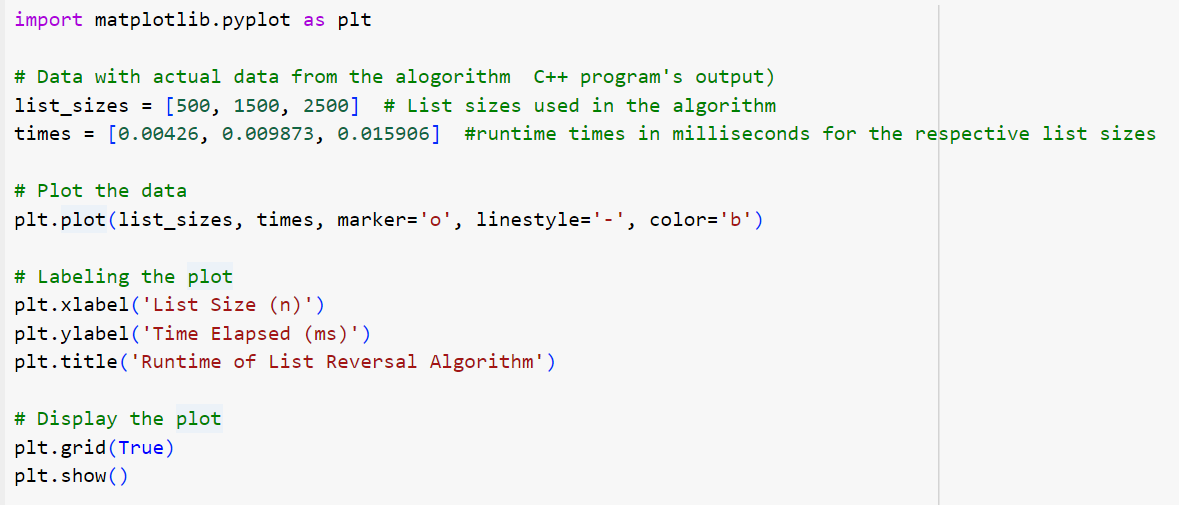
The data I used for the plot came from the output of the C++ program, which timed how long it took to reverse a list of different sizes. The sizes I tested were 500, 1500, and 2500 items, and I recorded how much time each reversal took, measured in milliseconds.

In the list\_sizes array, I stored the sizes of the lists, and in the times array, I kept track of how long each reversal took. The time is in milliseconds, so they give me an idea of how the algorithm’s speed changes as the list size increases.

#### **Plotting the Data**

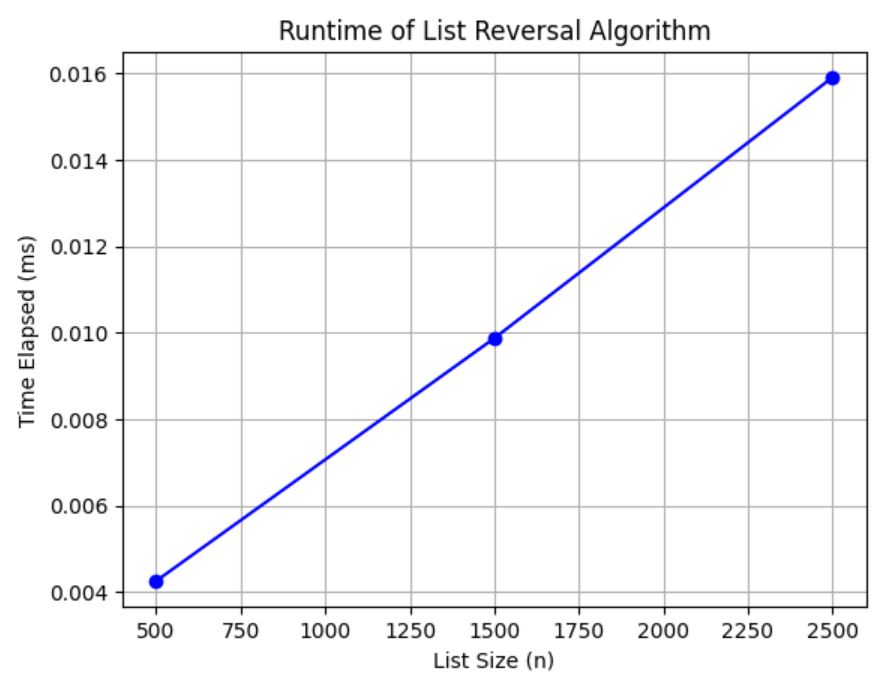
To create the plot, I used the matplotlib.pyplot library, which is perfect for this kind of visualization. The plot () function takes the list sizes and their corresponding times and creates a simple line plot, where the x-axis represents the size of the list, and the y-axis shows the time it took to reverse it.

Here’s the code I used to generate the plot:



This code generates a line plot with circular markers at each data point. The line connects the markers, showing how the runtime increases as the list size gets bigger. I also added labels for the x and y axes and a title to make the plot clearer. The grid makes it easier to read the graph.

#### **Understanding the Results**



Looking at the plot, I can see how the time to reverse the list increases as the list size grows. The relationship isn’t dramatic for the sizes I tested, but it's noticeable. This tells me that the algorithm performs well for smaller lists, but as the list size gets larger, the time will continue to increase.

The plot gives me a simple, visual way to see the connection between list size and runtime, which is helpful for understanding how the algorithm will scale if I were to use much larger lists.

#### **Conclusion**

In the end, this visualization helps me see how the reversal algorithm behaves with different input sizes. The plot shows the relationship between list size and runtime, making it easy to spot trends. This is a great way to analyze the performance of the algorithm, and it gives me useful insights into how well it scales with larger data.

**References**

**Python Documentation for Matplotlib**: *Matplotlib Developers. (n.d.). matplotlib - Visualization with Python. Matplotlib.* Retrieved from <https://matplotlib.org/stable/contents.html>

**C++ Programming Concepts**: *Stroustrup, B. (2013). The C++ programming language (4th ed.). Addison-Wesley Professional.*

**Chrono Library in C++ for Time Measurement**: *C++ Standard Library. (n.d.). chrono - C++ reference.* Retrieved from [https://en.cppreference.com/w/cpp](https://en.cppreference.com/w/cpp/chrono)/chrono

**Google Colab for Python Code Execution**: *Google. (n.d.). Google Colaboratory. Retrieved from* [https://colab.research.google.com](https://colab.research.google.com/)/

**Big-O Notation and Algorithm Performance**: Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd ed.). Addison-Wesley.*

**Data** Visualization Best Practices: *Few, S. (2012). Show Me the Numbers: Designing Tables and Graphs to Enlighten. Analytics Press.*

**General Overview on Algorithm Complexity**: *Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.*