| Function | ExtraLargeArray | LargeArray | MediumArray | SmallArray | TinyArray |
|---|---|---|---|---|---|
| **Length** | 1000000 | 100000 | 10000 | 100 | 10 |
| **Insert** | 703.089709 ms | 6.488583 ms | 120 µs | 21.5 µs | 14.667 µs |
| **Append** | 1.4905 ms | 382.917 µs | 79.542 µs | 88.125 µs | 43.917 µs |

By observing both functions, **doublerAppend** and **doublerInsert,** double every number in the given array. However, the execution times for these functions vary significantly depending on the size of the input array.

The **doublerAppend** function, which uses the push method to add elements to the end of the array, shows a relatively constant and fast execution time as the array size increases. It scales quite well with larger arrays, as seen from the execution times for the extraLargeArray, largeArray, and mediumArray.

On the other hand, **doublerInsert** function, which uses the unshift method to add elements to the beginning of the array, exhibits a different pattern. Its execution time increases noticeably as the size of the input array grows. For the extraLargeArray, the execution time is significantly longer compared to the doublerAppend function. The doublerInsert function does not scale as efficiently with larger arrays as the doublerAppend function does.

The slower function, **doublerInsert**, is significantly slower because of the way it adds elements to the array. In this function, the unshift method is used to insert elements at the beginning of the array, which has a time complexity of **O(n),** where **n** is the length of the array. This means that every time a new element is added to the array, all existing elements need to be shifted one position to the right to accommodate the new element at the beginning.

As the size of the input array increases, this shifting process becomes more time-consuming, resulting in slower execution times. For example, when dealing with large arrays like extraLargeArray, the unshift operation becomes highly inefficient since it requires moving a massive number of elements for each iteration. This inefficiency is the reason why the doublerInsert function exhibits a steep increase in execution time as the size of the array grows.

In contrast, the faster function, **doublerAppend**, uses the push method to add elements to the end of the array. The push operation has a time complexity of **O(1)**, which means it takes constant time regardless of the array size. When adding elements to the end of the array, no existing elements need to be shifted, making the process faster and more efficient, especially with larger arrays. This efficiency in adding elements at the end of the array is the key reason why the **doublerAppend** function outperforms the doublerInsert function in terms of execution time.