PART 1: Runtime Analysis

Function	**extraLargeArray**	**largeArray**	**mediumArray**	**smallArray**	**tinyArray**
insert	1.828999s	11.2387ms	220.5μs	71.8μs	59.1μs
append	4.6836ms	722μs	179.8μs	155.2μs	132.8μs

4.Read over the results, and write a paragraph that explains the pattern you see. How does each function “scale”? Which of the two functions scales better? How can you tell?

Based on the the results obtained, We can clearly identify the pattern of , how these two functions scale with different size of arrays.

For **doubleAppend** function,execution time increases with the increase size of input array which indicates that this function scale linearly with the input size.If we compare the execution times for different size of arrays ,it clearly visible that the time taken to process the **tinyArray** is significantly less than that for the **smallArray, mediumArray**, **largeArray**, and **extraLargeArray**.

On the other hand, for the **doublerInsert** function, which doubles each number in the array and inserts it at the beginning of a new array using **unshift**, we can observe a different pattern. The execution time for this function also increases as the array size increases, but the increase is much more significant compared to the **doublerAppend** function. In fact, the execution time for the **doublerInsert** function grows exponentially with the input size. The time taken to process the **tinyArray** is significantly less than that for the **smallArray**, **mediumArray**, and **largeArray**, but even the **largeArray** takes much longer to process compared to the **doublerAppend** function.

Based on these patterns, we can conclude that the **doublerAppend** function scales better than the **doublerInsert** function.

5.For extra credit, do some review / research on why the slower function is so slow, and summarize the reasoning for this.

The slower function, **doublerInsert**, utilizes the **\`unshift\`** method to insert elements at the beginning of the array. The reason why this function is slower compared to **doublerAppend** is due to the underlying behavior and complexity of the \`unshift\` operation.

The **unshift** operation requires shifting all existing elements in the array to make space for the new element at the beginning.The time complexity of the **unshif**t operation is O(n), where n is the number of elements in the array. This is because each shift requires moving all elements to a new position. Consequently, for each element insertion in the **doublerInsert** function, the time complexity is O(n), resulting in a quadratic time complexity overall.

In contrast, the **push** operation used in the **doublerAppend** function has a time complexity of O(1). The **push** operation simply adds the new element to the end of the array, without requiring any shifting of existing elements. This allows for a constant-time insertion for each element.

Therefore, as the size of the array increases, the **doublerInsert** function's execution time grows significantly due to the repeated shifting of elements, resulting in an exponential increase in execution time. In comparison, the **doublerAppend** function has a linear increase in execution time as it only performs constant-time operations for each element.

In summary, the slower execution of the **doublerInsert** function can be attributed to the use of the **unshift** operation, which requires shifting elements and has a time complexity of O(n). This leads to a quadratic time complexity overall and significantly slower execution times as the array size increases.