It’s time to run the file! Run the command `node runtime.js` in the same directory as where the file lives.

In your notes document, take note of the timing result for the **extraLargeArray** results–comparing when the **extraLargeArray** is passed to **doublerAppend** and **doublerInsert**.
insert 933.72594 ms
append 3.90994 ms

Next, edit the code in **runtime.js** to obtain timing results for calling the two functions with all of the differently sized arrays– **tinyArray**, **smallArray**, **mediumArray**, **largeArray**, and **extraLargeArray**. Notate these in your document in some kind table table so that you can easily compare the different values for the timers in relation to the size of the array that was passed into each function.

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

Array Size	doublerAppend Time (ms)	doublerInsert Time (ms)
tinyArray	0.03825	0.04017
smallArray	0.38011	1.52839
mediumArray	3.77921	375.26029
largeArray	37.55548	37,115.71619
extraLargeArray	3,909.94	933,725.94

From the results, we can observe a clear pattern: as the size of the array increases, the execution time for doublerAppend remains relatively constant, while the execution time for doublerInsert increases significantly.

The doublerAppend function scales better than doublerInsert. The time complexity of doublerAppend is O(n), where n is the size of the array. It iterates through the array once and performs a constant-time operation (pushing the element to the end of the array). This linear scaling is evident from the timing results, where the execution time increases proportionally to the size of the array.

On the other hand, the doublerInsert function exhibits quadratic scaling. The time complexity of doublerInsert is O(n^2), as for each element, it needs to unshift all the existing elements in the new_nums array, resulting in shifting and copying elements multiple times. This leads to a significant increase in execution time as the array size grows. The timing results clearly demonstrate the exponential growth of execution time for doublerInsert.

The slower performance of doublerInsert is due to the unshift operation inside the loop. Unshifting an element requires shifting all the existing elements to higher indices, which becomes increasingly costly as the array grows. In contrast, the push operation used in doublerAppend adds elements at the end of the array, which is a more efficient operation. Therefore, the repeated unshifting in doublerInsert results in a performance bottleneck and significantly slower execution compared to doublerAppend.