Step 1:
In your notes document, take note of the timing result for the extraLargeArray results– comparing when the extraLargeArray is passed to doublerAppend and doublerInsert.

Array	doublerAppend runtime (ms)	doublerInsert runtime (ms)
extraLargeArray	4.645327	974.952411

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.

Array	doublerAppend runtime (ms)	doublerInsert runtime (ms)
tinyArray	0.10928	0.045086
smallArray	0.113154	0.05285
mediumArray	0.155877	0.166403
largeArray	0.613829	6.750396
extraLargeArray	4.645327	974.952411
doubleXtraLargeArray = getSizedArray(1000000)	38.019932	180000

How does each function "scale"? Which of the two functions scales better? How can you tell?

We can tell from both the data collected and the way the functions are coded that doublerAppend's time complexity is O(n) and scales better, while doublerInsert runs $O(n^2)$ and scales worse.

Based on the runtime results, doublerInsert outperformed doublerAppend only to a point - at about the mediumArray where they performed almost equally fast. Beyond that size, doublerAppend drastically outperformed doublerInsert and clearly scales better. This is painfully evident with the additional runtime data provided by an array I added to the code out of curiosity - "doubleXtraLargeArray" which was 10 times larger than "extraLargeArray". While doublerAppend processed it in a tidy 38 ms, doublerInsert by contrast took over 3 minutes to run.

Given that both functions are coded with a for-loop - starting them both off at a baseline best-case time complexity of O(n) - the reason doubleAppend scales better at higher inputs (and stays O(n)) is because it utilizes *.push* which does *not* affect the indexes of the previous elements in the array since it simply appends an item to the *end* of the array. Thus, its runtime is only based on the size of the original input.

The doubleInsert function however uses *.unshift* which adds an item to the beginning of the array at index 0, thereby 'unshifting' the index of every single other item already inside the array each time an item is added. Thus, doubleInsert's use of both the for-loop and the .unshift method make its runtime dependent on the size of the input multiplied by the number of times the previous array items are re-indexed with each input - time complexity $O(n^2)$.