Runtimes:

XXXXXXXXX	Tiny	Small	Medium	Large	Extra Large
Insert	38.2 µs	46.5 µs	169.4 µs	8.0865 ms	972.2335 ms
Append	89.3 µs	96.4 µs	139 µs	576 µs	2.6681 ms

Scaling:

XXXXXXXXXXX	Tiny → Small	Small → Medium	Medium → Large	Large → Extra Large
Insert	1.217	3.643	47.736	120.229
Append	1.079	1.442	4.144	4.632

Explanation:

From just the math alone, it is pretty apparent that the ‘Insert’ function scales exponentially and the ‘Append’ function scales more linearly. The fact that the ‘Insert’ scaling jumps three times as much from 10 to 100 is already a sign of this. As the value of the array increases, the scale widens by a large margin. Whereas, if you look at the ‘Append’ function, you can see that the scale is pretty consistent from 10 → 100, 100 → 1000, 1000 → 10000, and 10000 → 100000. As such, it is clear that the ‘Insert’ function is comparable to $O(n^2)$ and the ‘Append’ function is comparable to O(n) - meaning ‘Append’ scales much better.

Extra Credit:

Looking up the runtime complexity of unshift() and push(), it can be said that I was right on the money. As, by default, unshift() and push() are O(n) O(1), respectively. Unshift() has to iterate through all of the elements in the array to change their positions since they have been bumped by one index value. Push(), on the other hand, doesn't change anything of the elements’ index value and only needs to be placed at the end. Since that is the case, as we are also wrapping each method inside of a for loop, that effectively changes the runtime complexity to as I described above ($O(n^2)$ for ‘Insert’ and(O(n) for ‘Append’).