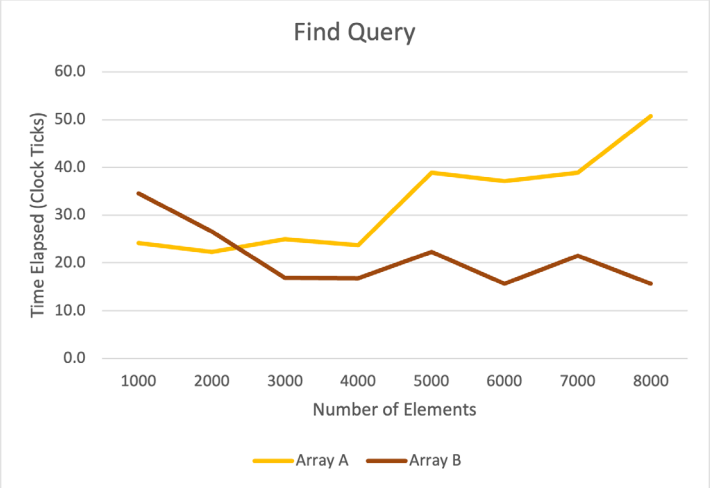
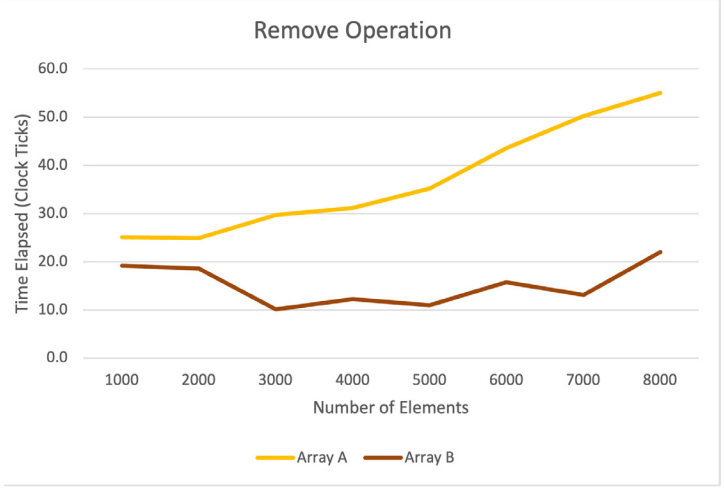
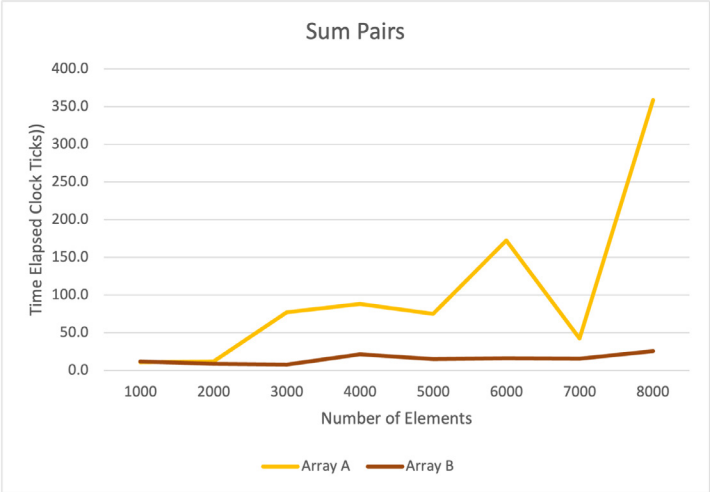


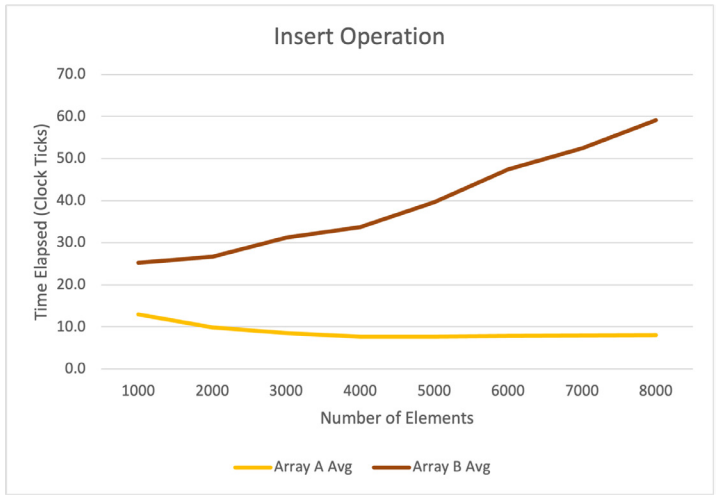
When there were fewer elements in the array, Array A seemed to be faster when a find query was ran using it. However, once it got past about 2000 elements, Array B became the faster one. The Linear Search has a time complexity of O(N), so it makes sense that it would take longer as more elements were added. However, Binary Search has a time complexity of O(LogN), so it's interesting to me that the time required actually decreased as more elements were added. It makes sense that it would become faster than a linear search, but it still should've increased in time as elements were added.



Across all variations of the number of elements, Array B was consistently the fastest. In addtion, as the number of elements increased, it seems as if Array B took less time, or at least had a consistent runtime, while Array A seemed to take longer. This makes sense though. Array A used a linear search algorithm to find the number that should be removed, while Array B used a Binary Search algorithm. Linear Search has a time complexity of O(N) which means it will increase steadily as the number of elements is increased. Binary Search has a time complexity of O(logN) which means the time won't increase a whole lot as elements are added.



For the Sum Pairs operation, Array B took roughly the same amount of time no matter how many elements were in the array. It seemed to have a time complexity of O(1) even though I thought it would have a time complexity of O(N). For Array B, it really just depends on what your target number is. Since Array B is sorted, if your target is lower, it will take less time, and if your target is higher, then it will take more time as it has to go through more of the array. However, it only ever has to go through the array once since it is sorted. Array A did increase in time as more elements were added, which makes sense because it is unsorted which means it has to go through the array multiple times to find the target value.



The Insert operation had a consistent runtime for Array A. This makes sense though since we are adding the new value to the end of the array each time. There is no searching that has to be done, it just gets added at the end, so it will take the same amount of time every time - O(1). For Array B however, this was not the case. It had to search through the Array everytime to find where the new value should be placed, so as the number of elements increased, the runtime did as well. It traverses through the array until it finds the correct spot, so it has a time complexity of O(N).