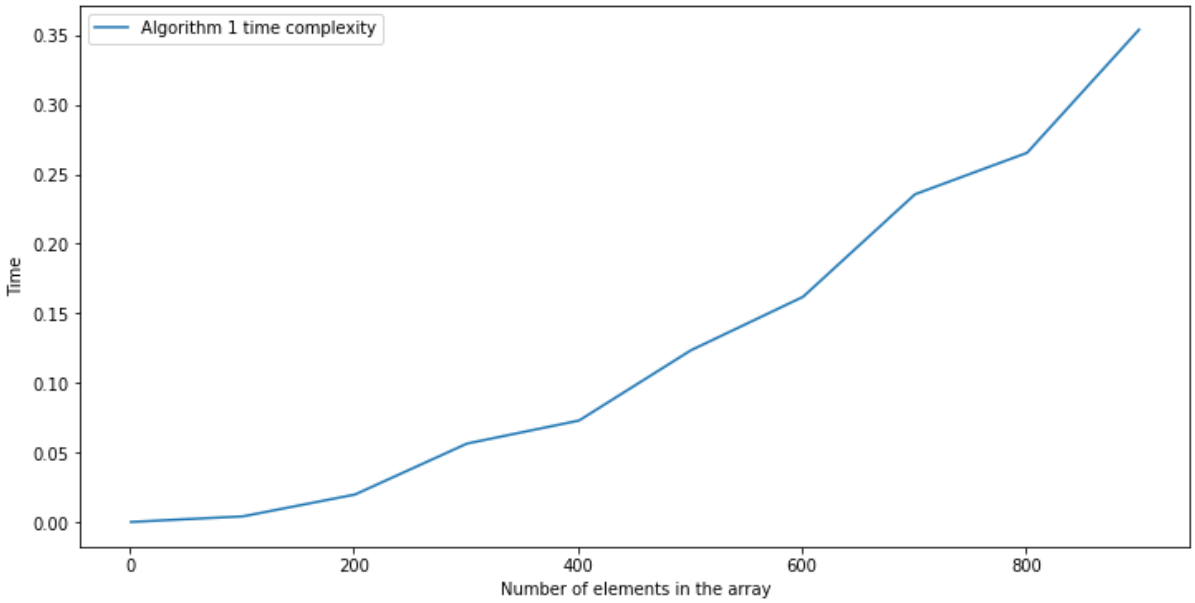
Lab Exercise 2 – Report

Algo 1:

Algo 1 code is an implementation of the bubble sort algorithm. The algorithm executes 2 loops to sort the list elements in ascending order. In every pass, adjacent elements are compared, and the largest element (in the remaining list) is pushed (bubbled) to the end of the list. The worst-case scenario is when the list is in the reverse sorted order. In this case we need to perform n-1 (where n = no of elements) passes and in each pass, we need to perform 1 less comparison (Since in each pass we reduce one element which is the largest one). Hence, for reverse sorted array we perform (n-1 + n-2 + …. 1) comparison.

$$\text{Time Complexity} = \sum_{1}^{n-1} i = \frac{(n-1)*n}{2} \text{ or } O(n^2)$$

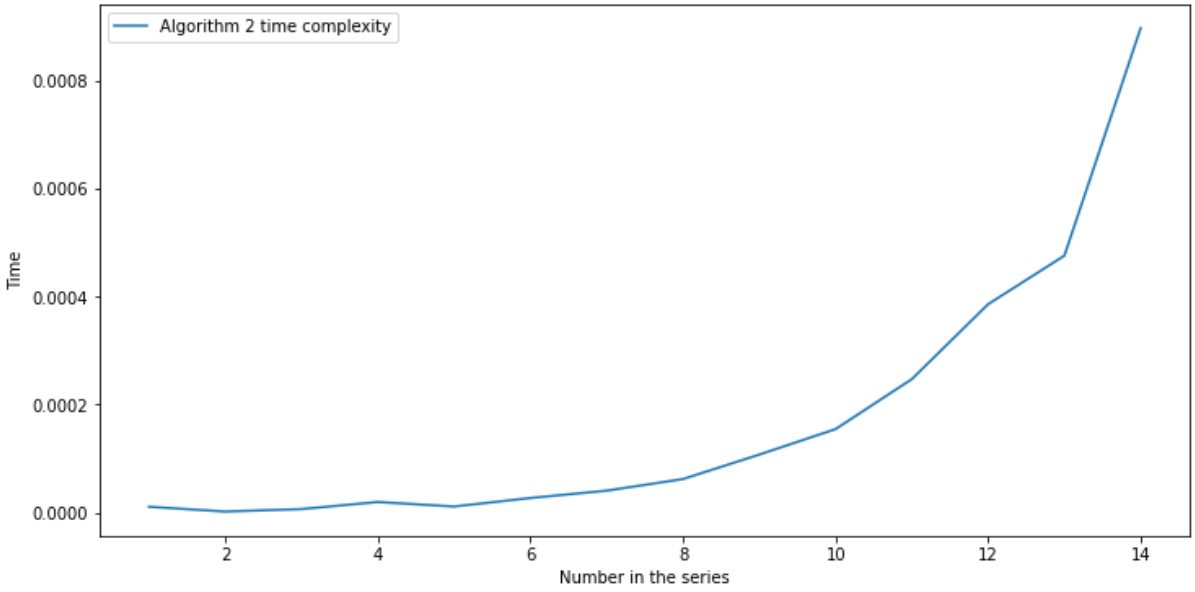


Algo 2:

Algo 2 is an implementation of calculating Fibonacci series numbers using recursion. The algorithm recalls itself at every step twice. Thus, if we start at level 1 with the number n, the function will be called twice at the first level (once for F(n-1) and once for F(n-2)), four times for the second level (twice each from F(n-1) and F(n-2)) and so on. Thus, as we increase the value of n to large numbers the total number of function calls will be roughly equal to $1+2+4+….2^{n-1}$.

$$\text{Time Complexity} = \sum_{1}^{n-1} 2^i = \frac{2^{n-1}-1}{2-1} \text{ or } O(2^n)$$

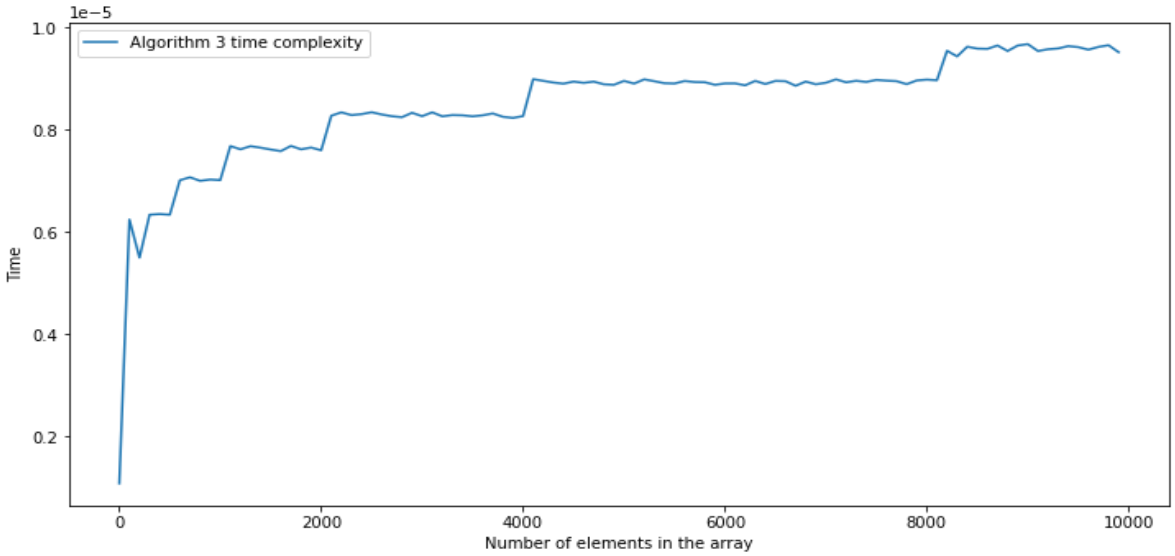
(The actual time complexity is less than 2^n (around 1.6^n) since many function calls return earlier when they reach the base case of n=2 or 1)



Algo 3:

Algo 3 is an implementation of binary search algorithm using recursion. Here, we try to search for an element in the (sorted) array by dividing the array into 2 halves and searching for the target element in the upper half if it is larger than the middle element or lower half if it is smaller. Since, we discard one half of the numbers on every function call and each call just checks if the target element is equal to, greater than, or smaller than the middle element, so we perform constant time operations in each step. Thus, for large values of n, in the worst cases, the time taken for searching is roughly equal to 1+1+1+1+…. k times (where k is the largest number such that $n/2^k <= 1$). Taking equality for simplicity we get, $n/2^k = 1$ or $k = \log n$.

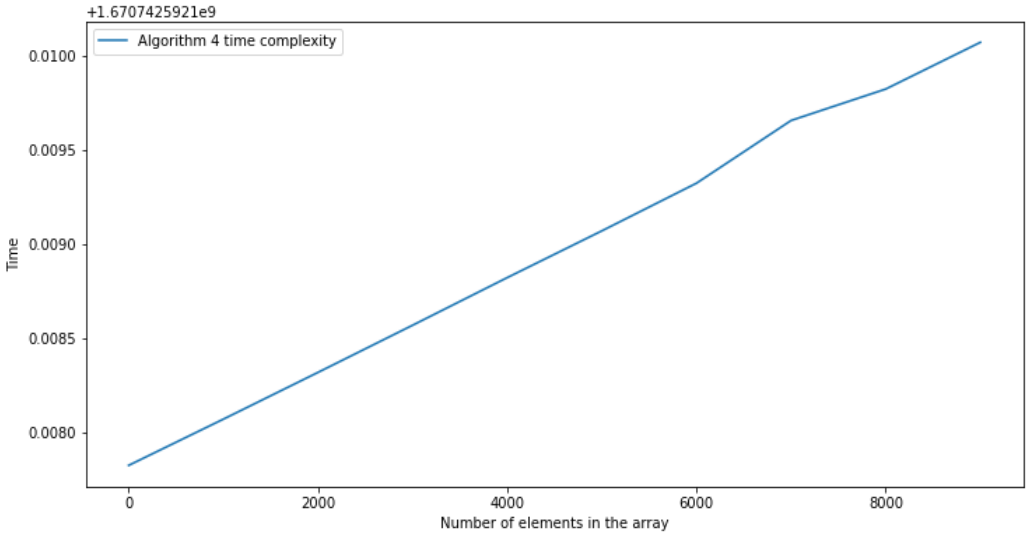
$$\text{Time Complexity} = \sum_{1}^{k} 1 = k = \log_2 n \text{ or } O(\log_2 n)$$



Algo 4:

Algo 4 is an implementation of linear search algorithm. This algorithm searches for an element by iterating through the entire array and returning the element if found. In the worst case, this algorithm makes n comparisons (where n is the no of elements) if the target element is at the end of the array.

$$\text{Time Complexity} = O(n)$$



Algo 5:

Algo 5 is an implementation of merge sort using recursion. Here, we try to sort a list of elements by breaking the array into 2 halves (left and right partition) at each step (recursively until we have lists of 1 element each), then sorting each sub-partition, finally merging the sorted arrays. The total number of partitions are of the order of log n (similar to binary search explained above) but since at each step we need to perform merge on each sub partition which is of the order O(n), so the final time complexity is of the order O (n log n)

$$\text{Time Complexity} = \sum_{1}^{\log n} \sum_{1}^{n} 1 = n \log_2 n \text{ or } O(n \log_2 n)$$