Bubble Sort

For i <- 0 to n-1

    For j <- 0 to n-1-k

        Do if vector[j] > vector [j+1]

            swap vector [j] <-> vector [j+1]

The bubble sort has a bound of $O(n^2)$. In my code the first iteration of bubble sort has n-1 elements. Thus, bound is O(n-1) but in big o the summation and subtraction operation is not valid which equals to O(n) and the second iteration has n-1-k. Thus bound is O(n-k-1) which also equals to O(n). However, the swaps operation bound is O(4). Because the iterations are nested the bound values multiplied. So the bound is O(n*n*4) which equals to $O(n^2)$.

MergeSort(carList, left, middle)

Do if left+1 < right

    Middle = left + ((right – left)/2)

    MergeSort(vector, left, middle)

    MergeSort (vector, left, middle)

    Merge(vector, left, middle, right)

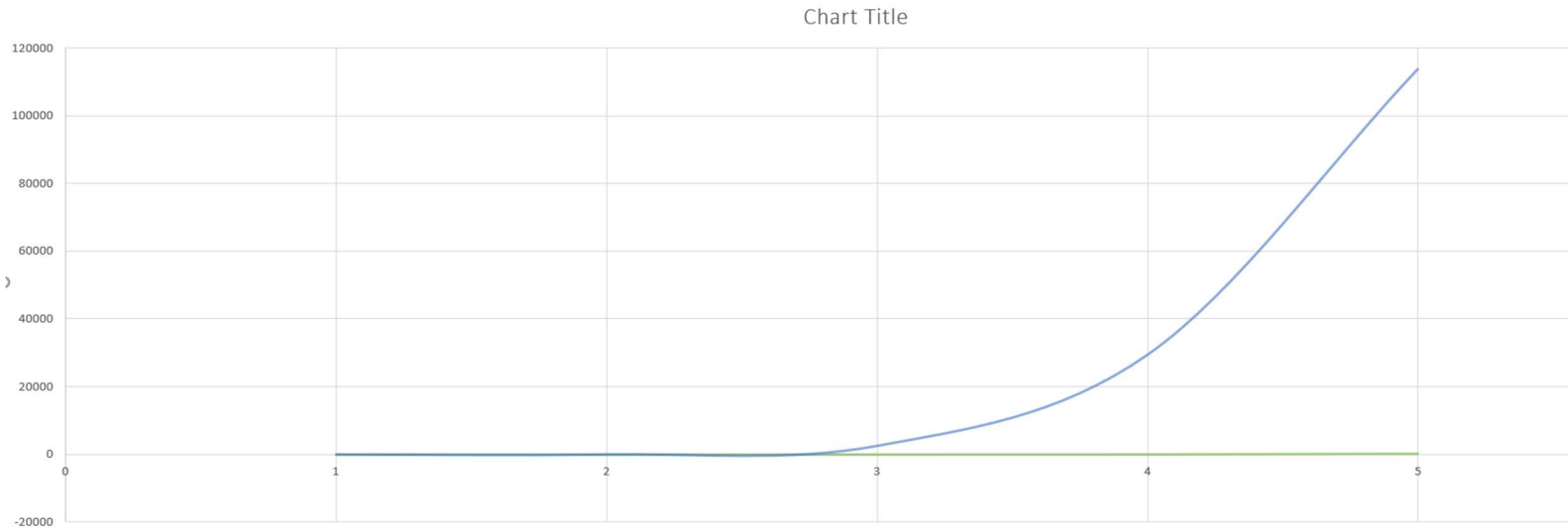
Merge Sort has two functions one of them to split the vector, the other one is to sort and merge them. MergeSort function is the one that splits the vector into half each time it iterates. It calls itself with half of its length until it reaches a vector segment that has one element which means MergeSort function iterates log(n) times.

Merge(carList, left, middle, right)

```
Vector vectTemp
While leftPoint < middle and rightPoint < right
        Do if array[leftPoint] <= array[rightPoint]
                Temp <-array[leftPoint]
                vectTemp <- temp
                leftPoint++
        else
                Temp <-array[rightPoint]
                vectTemp <- temp
                rightPoint ++
While leftPoint < middle
         Temp <-array[leftPoint]
        vectTemp <- temp
        leftPoint++
While rightPoint < right
         Temp <-array[rightPoint]
        vectTemp <- temp
        rightPoint ++
For i<-left to right
        carList[i] = vecTemp[i-left]
```

In the merge function first, it looks at the list until it reaches until the left or the right limitations. While doing that it compares the right and left side of the limits and puts them in a sorted way to the temporary list. Then puts rest of the values in the left and right sides of the list. Finally, the code puts the sorted temporary list to the original one. So the algorithm loops n each time. MergeSort function calls the Merge function so we can say they are acting in a nested manner. So we should multiply the boundary so we can find the big O of Merge sort. So we multiply the O(n) and O(logn). The result is O(nlogn) so it holds the condition of merge sort.

	1000	10000	100000	500000	1000000
Merge Sort	0.01241	0.176434	2.07977	4.90944	23.2883
	0.01239	0.194775	2.08972	4.28904	23.7052
	0.01222	0.176163	2.05545	4.04455	19.1183
	0.012527	0.176396	2.0709	4.08618	22.9546
	0.012776	0.177276	2.05576	4.90422	22.8504
	0.012392	0.17842	2.05419	4.19333	22.9078
	0.012154	0.177806	2.05918	4.21369	22.9613
	0.013634	0.177949	2.05266	4.3081	22.806
	0.012206	0.178826	2.06643	4.18557	23.0694
	0.012333	0.176566	2.057	4.25935	17.1759
Average	**0.012504**	**0.179061**	**2.06411**	**4.339347**	**22.08372**
Bubble Sort	0.07637	7.21194	970.869	29892.983	109492.637
	0.094211	7.11944	936.93	29539.789	117631.786
	0.079108	7.09165	994.642	29371.842	109734.94
	0.076766	7.13974	963.361	29287.097	113527.075
	0.076319	7.17301	1033.05	28979.539	116947.854
	0.086985	7.30608	1119.58	29458.736	113569.604
	0.076244	7.24843	974.425	29172.78	111247.932
	0.076222	7.14437	1015.83	29148.176	109818.762
	0.076053	7.1955	1076.39	29876.359	117250.534
	0.077001	7.67264	993.769	29578.483	119531.937
Average	**0.079528**	**7.23028**	**1007.88**	**29430.58**	**113875.31**



In plot line the blue line is the bubble sort which is $O(n^2)$ and the green one is merge sort which is also O(nlogn) in the figure we can see that the bubble sort increases faster than merge sort. When bubble sort has 1.000.000 elements we can see the difference is clear. Thus, it is a prove of bubble sorts asymptotic bound n^2 and the cost is more than merge sort.