Task 1

Bubble sort compares two elements which are next to each other. If the element on the left is greater than the element on the right they are being swapped. Because of two for statements Bubble sort is iterating the whole array multiple times untill it is not sorted. Last iteration is checking if everything is in correct position. It is efficient for small arrays.

Insertion sort, key is the element from array(starts at second element) which is being compared to every element untill a smaller element is found, if found, the smaller element is replaced with the key element then key value is the next element and the process is being repeated. This sorting function is also efficient for small arrays but is faster than bubble sort.

Quick sort, there are few variations of this sorting algorithm regarding choosing the position of pivot element. In my application I’m choosing the version with witch choosing the rightmost element, which is also the last element of the array. Next the array is is being rearranged – elements smaller than pivot are on the left side and the greater than pivot are on the right side. The function partition is swapping elements and at the end it takes the variable i and swaps the pivot with element in the middle. Quick sort function does that for left side of the array and for the right so the array is correctly sorted.

Merge sort is based on the dividing and conquer. First we split the array into two arrays - subarrays. Then we are sorting the both subarrays, At the end the from the both subarrays a new sorted array is created. Merge sort function uses the recursive function which divides the array into two halves until the subarray is of size 1. Then the function merge is called to merge sorted subarrays.

I tested running times for every type of these sorts with random integers and the array which is getting bigger. Bubble sort has the worst time every time when the array is bigger than 35 elements. It is not worth using at all. Insertion sort works great for very small arrays 5-75 elements. Quick sort works very well also with small and big arrays, but is slightly slower for smaller element arrays than insertion sort. Merge sort has bigger running times than quick sort at the begginning but if the array is bigger both of them have similiar running time. They both seem to be good choice and doesn’t matter which to choose better. My implementation counts average time based on sorting one element in the array.

Task 2

Counting sort – max\_element finds greatest element from the vector and min\_element finds the smallest element from the vector, then 2 new vectors are being created count and output. Count stores count of each element. And then stores the cumulative count of each vector. Next the function is finding index of each element of the original vector and placing it in the output vector. And finally the pin vector is being replaced by sorted pins. I also added in this task the vector temp\_out which follows the indexes of the pins and changes them in the credit cards number vector so the vectors are sorted by pins but there are two seperate vectors. I also used the merge sort algorithm which I changed to be more efficient and now it is based on vector and not on array which makes it faster. Vector result1 is being sorted and result2 follows the indexes of result1 and takes the credit card numbers in correct way. To perform this task I imported the creditcardsnumbers.txt by skipping first 2 lines and seperating the line by two and pushing it back to two seperate vectors. Then I created 2 new vectors which has the same values as the originial to compare the sorting times performed by merge sort and counting sort. Then the pins are being converted to strings and 0’s are added to them beacuse while converting to int 0’s at the beggining were removed. Counting sort time is faster it takes around 0.006 seconds while merge sort need around 0.12 seconds to sort all the pins for the whole 20 000 elements. But the merge sort is faster up to array of 20-21 elements. Every next array of elements number bigger than that counting sort sorts it faster. I also tested sorting times for arrays of smaller elements.

Task 3

My implementation of own vector is based on array, the dynamic array. Class has declaration of capacity, elements and array memory. The whole declaration of my own vector is based on the standard vector from vector library. Functions work the same way. I tested perforamnce for pushing back elements, reading values from vector and pushing to another vector, erasing, resizing. My vector is better at pushing back and reading. Standard library vector wins at erasing and resizing and the difference is huge. Read times are interesting because times are similiar and once wins my vector and once the standard vector. I think it depedens on the random values. From my observation I can say that using the standard vector is the better option than impletenting own for most of the solutions. Of course we can decrease the time of our vector but we need to remove some security checks which can be risky.