b) Correctness of Selection Sort algorithm

There are 2 loop invariants for this algorithm: one for the inner loop and another for the outer loop. In order for Selection Sort algorithm to be correct, the loop invariants must be true during initialization, maintenance and termination.

Inner loop invariant: At the start of each iteration of the for loop, array [min_el] is less than or equal to array [i…j-1]

Initialization: Before the first iteration of the loop, min_el is set to i. So, min_el shows the smallest element of subarray [i…j-1]. As it is the only element of the array, we can say the loop invariant is true.

Maintenance: At the beginning of each iteration, the loop invariant must be true. min_el shows the smallest element in the subarray A[i..j-1]. There are two cases:

- array[j] < array[min_el] -> If this is true, it means that array [j] is less than or equal to elements in subarray A[i...j-1]. So, min_el is assigned value of j since it is the smallest.
- array[j] ≥ array[min_el] -> If this is true, nothing is executed. min_el shows the smallest element of A[i..j].

Termination: At termination, j will have the value num (that is why the loop does not continue to iterate anymore). Assuming the loop invariant was true during all iterations, min_el shows an element less than or equal to all elements in subarray [i…j-1] which is equal to A[i..num-1]

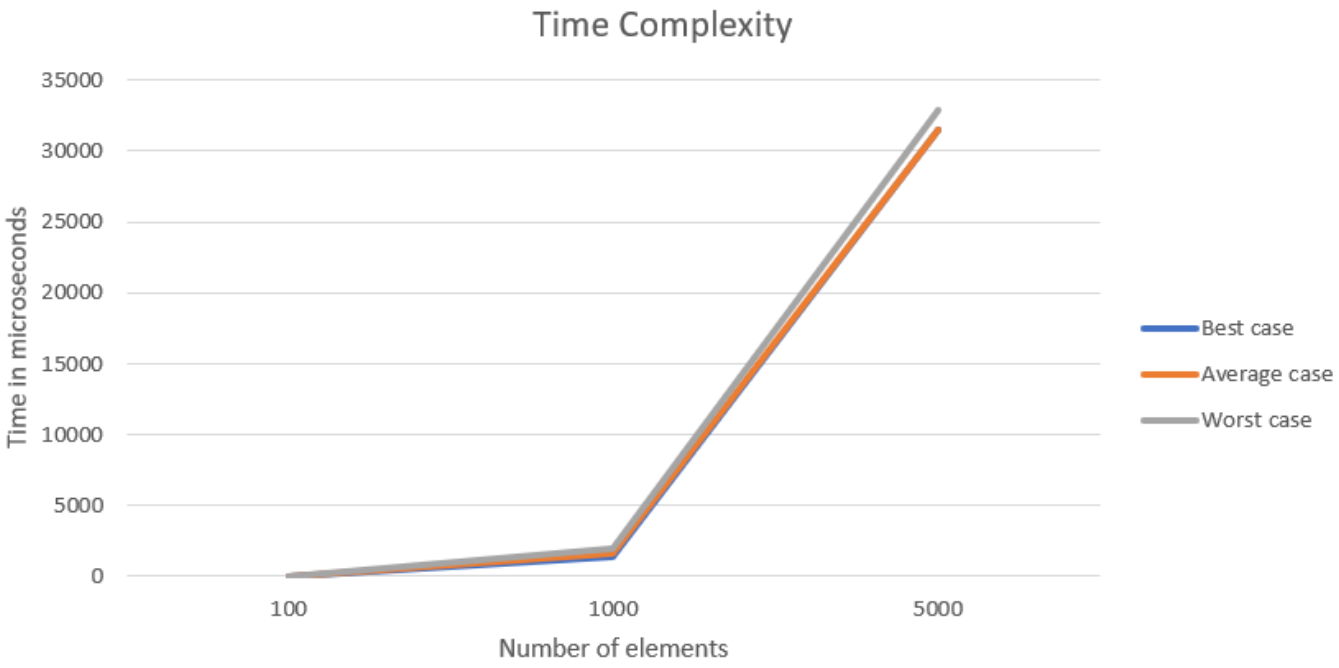
Outer loop invariant: At the beginning of each iteration of the for loop, the subarray [0…i-1] is always sorted in ascending order and contains i-1 elements smallest elements of array [0…num-1].

Initialization: When we start looping for the first time, there are no elements in the subarray [0…i-1]. Hence the subarray is empty and sorted.

Maintenance: At the beginning of each iteration, the loop invariant must be true. From the iterations before the actual iteration, we assume that the elements of the subarray [0…i-1] are sorted and are in their correct order. They are smaller than the elements of the unsorted array [i…num-2]. The array enters the for loop and when it reaches the inner for loop, the smallest element of the unsorted subarray [i…num-2] will be swapped with the element at position i. Array[i] will be bigger than the elements of the subarray [0…i-1] and smaller than the elements of the subarray [i+1…num-2]. The element at position i will be added at the sorted subarray, hence, now the sorted subarray will be from 0 to i. This iteration ends and we will go to the next iterations and the same process will be repeated until the outer for loop is not true anymore.

Termination: The for loop ends when i = num-1. Assuming the loop invariant was true during all iterations, we can say that the subarray [0…i-1], which at termination is equal to [0…num-2] is sorted and all the elements of the subarray are smaller than the last element, which is at position num-1. Since there is only one element in the unsorted array and it is bigger than all the other elements, this element at position num-1 is definitely at the right place. So, the whole array [0…num-1] is sorted.

d)



e) The total time for the Selection Sort algorithm has 3 parts:

- The running time for the inner loop

- The running time for the swapping part

- The running time for the rest of the outer for loop

Swapping part takes **n** calls, same as the rest of the outer for loop, which is just testing and incrementing the loop variable. Both take constant time, so the running time is **O(n)**. The running time for the inner loop is n(n+1)/2 no matter of the input. Hence it belongs to **$O(n^2)$**.

We add the running time of the 3 parts, and we see that the leading term is **n^2**. So, the running time for selection sort is **$O(n^2)$** whether it is best case, worst case or average case. We can also see it from the graph above that the lines have parabolic shape. The only slight difference of these lines is the swapping part. They have different number of calls for swapping. But **n^2** is dominant to n, so the 3 cases belong to **$O(n^2)$**.