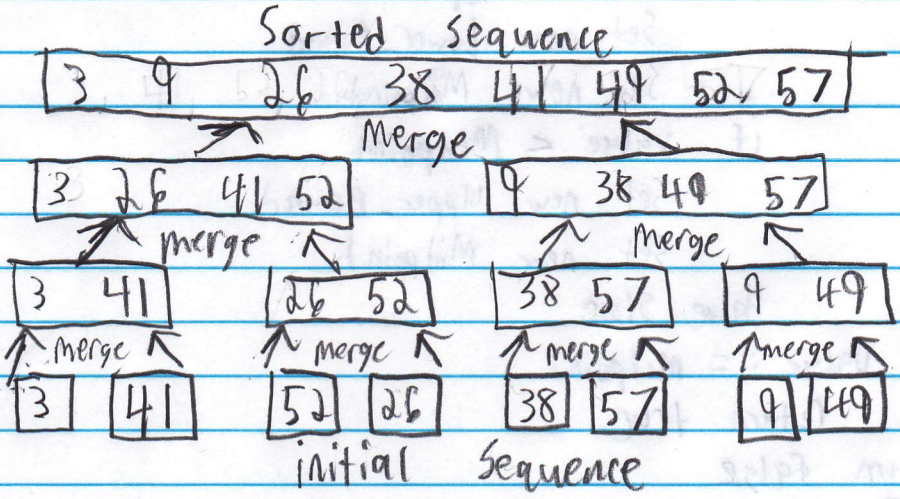
1. Pseudo-code:

```
for i = 1 to n-1        *indexed at 1, not 0
lowest = A[i]
    for j = i+1 to n-1
    if A[j] <= A[i] then lowest = A[j]
A[i] = lowest
```

- The initial for loop is the invariant. It must iterate through all n-1 elements in the array. The second for loop deals with the right side, or the unsorted side, and is decreasing with each iteration of the initial for loop.
- This only runs n-1 times since it is sorting numbers as it iterates. By the time iteration n is reached, there are no numbers left to compare to, since it is the last element in the array. In other words, it is naturally sorted in the process.
- The best/worst case runtimes are $O(n^2)$. This is because selection sort is unaffected by initial order. It must continuously loop and compare all elements in the array, even if it is already sorted.

2.



3. Worst case runtime is O(n), in which the nth element must be inserted in the front of the sorted array.

$$T(n) \begin{cases} O(1) & \text{if } n=1 \\ T(n-1)+O(n) & \text{if } n>1 \end{cases}$$

* If n=1, the array is already sorted
* If n>1, the total time is the time it takes to sort n-1 elements plus O(n) which is the time it takes to insert the nth element in the worst case.

4.
```
Binary_Search (Array, Value)
Check if array is empty
Set Size (Array size)
Set Midpoint of Array
Set Lower_Bound of Array
Set Upper_Bound of Array

While Size >1
    if Value == midpoint
        return true
    if Value > Midpoint
        Set new Lower_Bound
        Set new Midpoint
    if Value < Midpoint
        Set new Upper_Bound
        Set new Midpoint
    Halve Size
if Value == Midpoint
    return true
return false
```
* In the worst case scenario, the Value does not exist in the array. So the code must keep halving itself until the size becomes one. In turn, this

relation of constant halving to a size n can be described as log n. For example if the size of the array is 8, it is halved to 4, 2, then 1. This is 3 comparisons, which is the same as $\log_2 8$ or $2^x = 8$ or 3.

5.

a. (1,5)
(2,5)
(3,4)
(3,5)
(4,5)

b. The array with the most inversions is an ordered array from greatest to least. For example: [5,4,3,2,1]. This can be expressed as (n-1)!

C. The number of inversions is the same as the amount of times a number is moved in an insertion sort. This is because insertion sort uses the idea of inversions. For a number to be placed in the right spot, it must go through all of its inversions, and then it will arrive at the first number that is lower than it.

d.
```
int inversions_count(int[] array) {
    int inversions = 0;
    for(int i=0; i<array.size(); i++) {
        for(int j=i+1; j<array.size(); j++) {
            if(array[i] > array[j]) {
                inversions++;
            }
        }
    }
    return inversions
}
```