Problem 1:

We have discussed insertion sort as an algorithm. Now that we know how to analyze algorithms AND asymptotic notation, let's revisit Insertion Sort!

You may refer to class slides/ online resources to refresh your memory on the insertion sort algorithm.

For insertion sort, what is the

Best Case
The best case scenario of insertion sort is if the input array is already sorted. In this case, Insertion Sort only has to verify that the array is sorted, without switching any elements. Therefore, it takes O(n) time to go through the array, and O(1) time for any other operations. In the end, the best case time complexity is O(n).

Average Case
The average time complexity of insertion sort is if we need to switch half the elements in the array, since this is the average between the best and the worst case. In this case, if there are n elements, it needs to compare n/2 elements. Since we are comparing this asymptotically, we will remove the devising factor and therefore it takes O(n) time to compare the elements, and O(n) time to insert each element. Therefore, the average case time complexity is O(n^2).

Worst Case
The worst case of insertion sort occurs when the array is already sorted, but in reverse order. This requires the most amount of elements inserted. Since we need to compare n elements, (every element in the array), the time to compare the elements is O(n). Since we also need to insert every element in reverse order, the time to insert will be O(n). Therefore, the worst case time complexity will be O(n^2) as well.

Problem 2:
Below is the pseudocode to perform matrix multiplication that we discussed in HW3.

```
MATRIX_MULTIPLY(A, B):
  if columns(A) ≠ rows(B):
    raise ValueError("Matrix multiplication is not defined.")

  rows_A ← number of rows in A
  cols_A ← number of columns in A
  cols_B ← number of columns in B
  result ← matrix of size rows_A x cols_B filled with zeros

  for i from 1 to rows_A do:
```

```
    for j from 1 to cols_B do:
    sum ← 0
      for k from 1 to cols_A do:
        sum ← sum + A[i][k] * B[k][j]
      result[i][j] ← sum return result
```

For the above pseudocode, what is the

Best Case

The best case of this pseudo code is O(1). This occurs when the matrices are 1 x 1 matrices. This would be equivalent to simple multiplication, and the loops are essentially not executed. This takes constant time.

Average Case

The average case of this pseudo code is O(n^3), assuming that A and B are not 1 x 1 matrices, and have differing dimensions. In this case, all 3 loops will be executed, from any number from 2 to n/2. Since we are doing time complexity, the division of 2 is ignored and the resulting average time complexity is O(n) * O(n) * O(n), which is O(n^3).

Worst Case

The worst case of this pseudo code is also O(n^3), also assuming that A and B are not 1 x 1 matrices, and have differing dimensions. In this case, all 3 loops will be executed from any number from 2 to n. Due to this, we are looking at the time complexity of O(n) * O(n) * O(n), and it results in a worst-case time complexity of O(n^3).