# Problem 2:

For 10,000 randomly generated numbers:

| K | Number of Recursion Calls | Number of Move Operations |
|---|---|---|
| 2 | 19997 | 25010130 |
| 3 | 19997 | 25005287 |
| 4 | 19993 | 12509747 |
| 5 | 19991 | 10029321 |
| 6 | 19989 | 8357090 |
| 7 | 19987 | 7161699 |
| 8 | 19985 | 6269081 |
| 9 | 19983 | 5596105 |
| 10 | 19981 | 5027870 |

## Pseudocode:

```
Partition(A, lo, hi):
  For j from lo to hi:
    I = (lo – 1) + 1
    If A[j] <= A[hi]
      Swap -> A[hi] with A[I]
    Swap -> A[I + 1] with A[hi]
    Return I + 1
QuickSort(A, lo, hi):
    If len(inputArray) <= 1:
      Return inputArray
    If lo < hi:
      Pivot = Partition(A, lo, hi)
      Quicksort(A, pivot + 1, hi)
      Quicksort(A, lo, pivot - 1)

 InsertionSort(A)
```

```
For j from 1 to A.length:
  I = j – 1
    While I >= 0 and A[I] > A[j]
        A[I + 1] = A[I]
        I -= 1
    A[I + 1] = A[j]
```

```
QuickInsertionSort(A, n, k):
    If n > k:
        QuickSort()
    InsertionSort()
```

## Analysis:

QuickSort():

This is a recursive function that takes in a list to sort, a starting index, and an ending index. It then calls a second function Partition() that takes an element, and sorts the array by placing all the larger elements before it in the array, and the larger elements after it. Once the partition is finished, QuickSort() recursively calls itself to sort the left and right halves of the array. The complexity of Partition() is linear, it is O(n) as it only goes through the array once. The complexity of QuickSort() as a result of this is O(nlogn).

InsertionSort():

This is an iterative algorithm that sorts in place. It iterates through the input list and compares each of the items to its neighbor. If the value is less than its neighbor, it iterates backwards through the list and comparing the item to the sorted list so far. When it encounters another item that is less than the current item, it inserts that value before it in the list. The time complexity of this is O(n^2) as it must traverse the list multiple times.