Modified Pseudocode

The original pseudo code had the counter incrementing only when elements got moved around, but what we want is for the counter to increment with every comparison.

The solution was to move the `A[j] > v` comparison outside the while loop into it's own if clause right after incrementing count as long as 'j >= 0' is true.

```
SortAnalysis(A[0 .. n-1])
//input: Array A[0..n-1] of n orderable elements
//output: Total number of key comparisons made

count <- 0
for i <- 1 to n-1 do
    v <- A[i]
    j <- i-1
    while j>=0 do
        count <- count + 1
        if A[j] > v then
            A[j+1] <- A[j]
            j <- j - 1
        else
            break
    A[j+1] <- v
```

Summary of Results

Based on our analysis, we found that the *comparisons per* n^2 stayed relatively consistent (around 0.25)

Raw Data

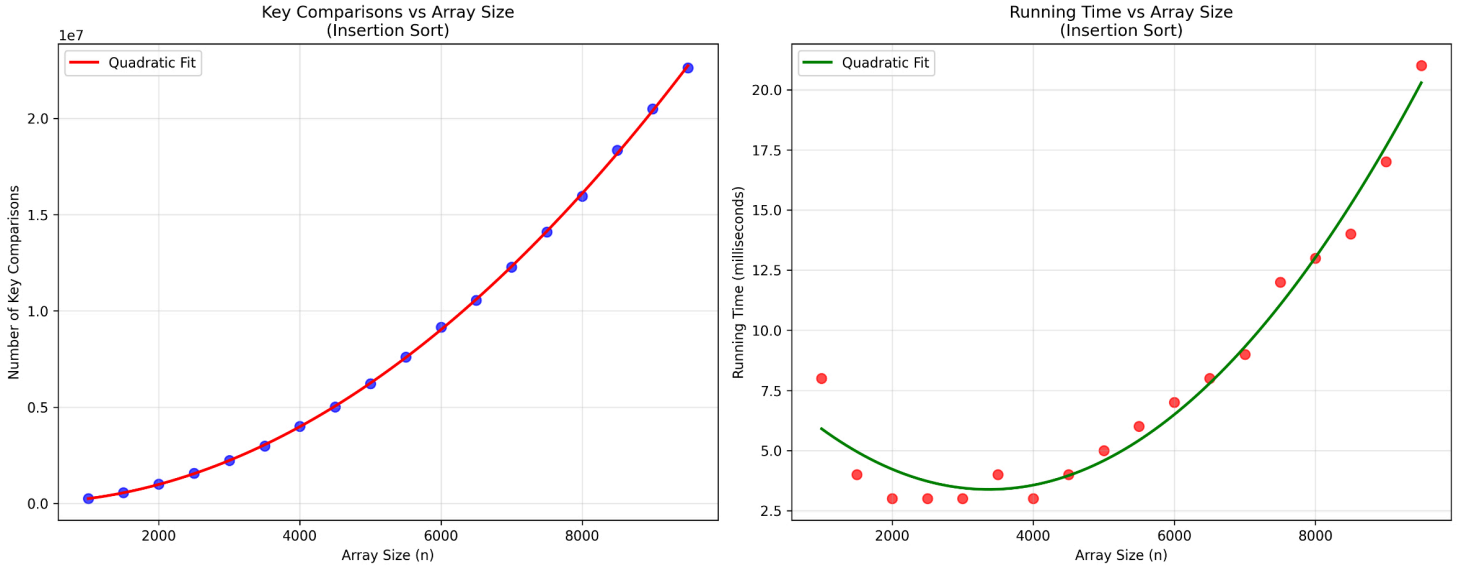
```
:!python3 /home/dev/projects/school/CSC401-Project1/analysis.py
=== DATA SUMMARY ===
Data points analyzed: 18
Array size range: 1000 - 9500
Minimum comparisons: 244,783
Maximum comparisons: 22,127,480

Predicted comparisons for n=10,000: 24,801,693
Predicted running time for n=10,000: 20.8 ms

=== DETAILED RESULTS TABLE ===
Size     | Comparisons | Time (ms) | Comp/n²    | Theoretical
-----------------------------------------------------------------
 1000 |     244,783 |      4.0 | 0.244783 |     249,750
 1500 |     559,028 |     13.0 | 0.248457 |     562,125
 2000 |     997,692 |      1.0 | 0.249423 |     999,500
 2500 |   1,580,953 |      2.0 | 0.252952 |   1,561,875
 3000 |   2,270,047 |      3.0 | 0.252227 |   2,249,250
 3500 |   3,069,543 |      2.0 | 0.250575 |   3,061,625
 4000 |   4,020,316 |      3.0 | 0.251270 |   3,999,000
 4500 |   5,033,262 |      3.0 | 0.248556 |   5,061,375
 5000 |   6,191,463 |      4.0 | 0.247659 |   6,248,750
 5500 |   7,524,604 |      5.0 | 0.248747 |   7,561,125
 6000 |   8,909,020 |      7.0 | 0.247473 |   8,998,500
 6500 |  10,621,300 |      7.0 | 0.251392 |  10,560,875
 7000 |  12,164,766 |      9.0 | 0.248261 |  12,248,250
 7500 |  14,166,612 |     12.0 | 0.251851 |  14,060,625
 8000 |  16,191,149 |     13.0 | 0.252987 |  15,998,000
 8500 |  18,027,718 |     13.0 | 0.249519 |  18,060,375
 9000 |  20,180,375 |     15.0 | 0.249140 |  20,247,750
 9500 |  22,127,480 |     17.0 | 0.245180 |  22,560,125


Press ENTER or type command to continue
```

Graph



From the data, we can conclude that comparisons roughly quadruple when we double the array size, which implies a time complexity of $O(n^2)$

The theoretical analysis on insertion sort expects about $(n^2)/4$ comparisons for any given random data. This is consistent with our ratio of ~0.249.

Predicting n=10,000

Based on our program, our predictions would be:

- **24,801,693 comparisons**
- **~20.8 milliseconds**

The theoretical prediction would be:

$$T(10000) = \frac{10000^2}{4} = 25,000,000 \; comparisons$$

This is pretty close to our results we got.

Algorithm Analysis

For Insertion sort:

- Best case - when array is already sorted -> $O(n)$
- Average case - for random data -> $O(n^2)$
- Worst case - reverse sorted array -> $O(n^2)$

This confirms our experiment which was testing for the average case.