Project 1
Empirical Analysis of Algorithms

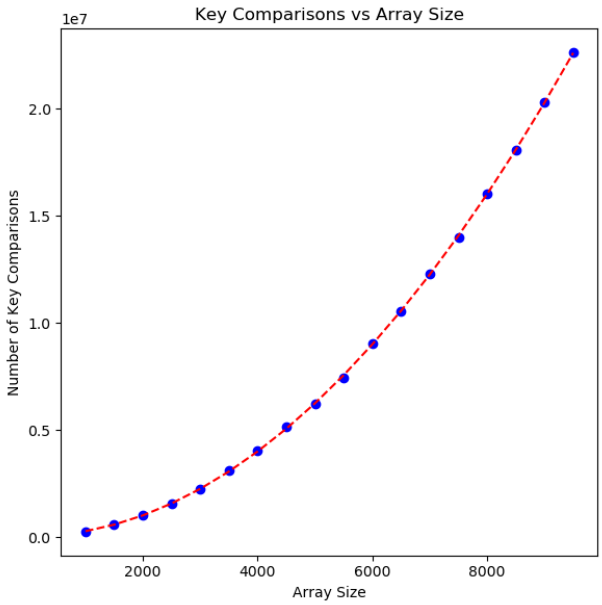
Ashwini Sudheer Kumar

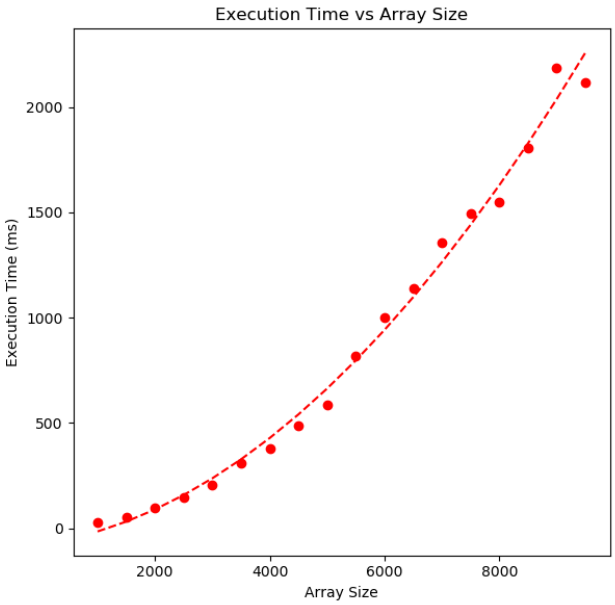
Digvijay Hethur Jagadeesha

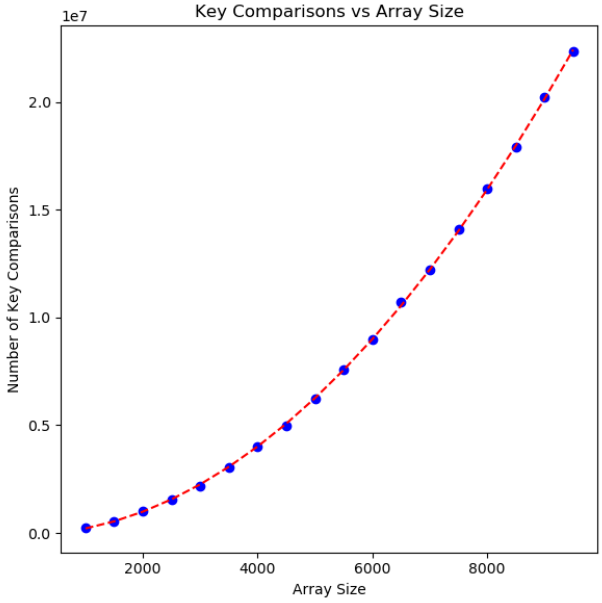
Jateen Joharapurkar

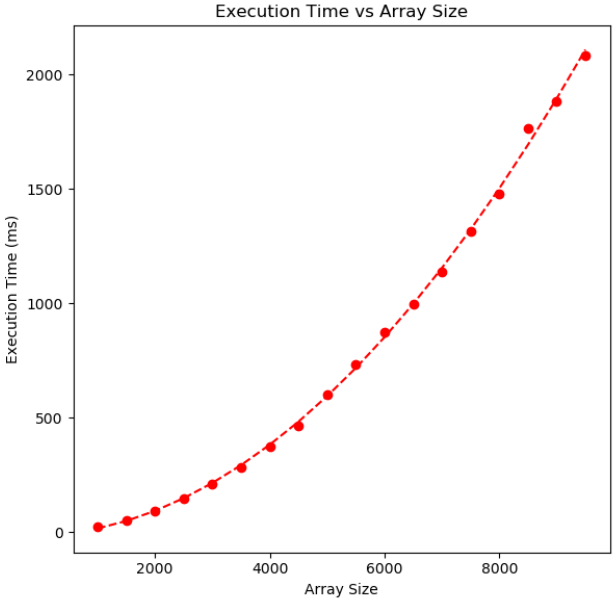
1. Corrected Pseudocode for SortAnalysis:

```
Algorithm SortAnalysis(A[0..n-1])
// Input: An array A[0..n-1] of n orderable elements
// Output: The total number of key comparisons made
count ← 0
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        count ← count + 1
        A[j + 1] ← A[j]
        j ← j − 1
    if j ≥ 0 then
        count ← count + 1
    A[j + 1] ← v
return count
```


Key Comparisons vs Array Size
1e7
Number of Key Comparisons
Array Size
0.0
0.5
1.0
1.5
2.0
2000
4000
6000
8000


Execution Time vs Array Size
Execution Time (ms)
Array Size
0
500
1000
1500
2000
2000
4000
6000
8000


Key Comparisons vs Array Size
1e7
Number of Key Comparisons
Array Size
0.0
0.5
1.0
1.5
2.0
2000
4000
6000
8000


Execution Time vs Array Size
Execution Time (ms)
Array Size
0
500
1000
1500
2000
2000
4000
6000
8000

3. Hypothesis about the algorithm's average-case efficiency:

Insertion Sort Algorithm

Algorithm used in this project is **Insertion Sort** which is a simple sorting algorithm that builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

- It is simple to implement.
- It is efficient for (quite) small data sets.
- It is stable; it does not change the relative order of elements with equal keys.
- It is in-place; it only requires a constant amount O(1) of additional memory space.

Average-Case Efficiency Analysis:

1. **Key Comparisons**:
    - Insertion sort works by iterating through the array and inserting each element into its correct position in the sorted portion of the array.
    - For each element A[i] (from the second element to the last), it is compared against elements in the sorted portion to find its correct position.
    - For an element at index i, the number of comparisons in the worst case is i, because it might need to be compared with all previous i elements if it is the smallest element so far.

    The total number of key comparisons, in the worst case, is the sum of the first n−1 integers:

    Total Comparisons (Worst Case) = $\sum_{i=1}^{n-1} i = \frac{n\,(n-1)}{2}$

    In the average case, assuming the elements are randomly ordered, the number of comparisons for each element would be roughly half of the worst-case comparisons.

    Average Comparisons = $\frac{1}{2} \cdot \sum_{i=1}^{n-1} i = \frac{1}{2} \cdot \frac{n\,(n-1)}{2} = \frac{n\,(n-1)}{4}$

2. **Running Time**:

    $T(n) \approx \frac{n\,(n-1)}{4}$

3. **Big-O Notation**:

    $T(n) = O(n^2)$

4. **Empirical Data Analysis**

   From the empirical data obtained:

```
/usr/bin/python3 /Users/digvijay/PycharmProjects/daaProjects/sort_analysis.py
Array Size: 1000 | Key Comparisons: 262679 | Execution Time (ms): 25.776147842407227
Array Size: 1500 | Key Comparisons: 577198 | Execution Time (ms): 54.09717559814453
Array Size: 2000 | Key Comparisons: 1009313 | Execution Time (ms): 95.95632553100586
Array Size: 2500 | Key Comparisons: 1575877 | Execution Time (ms): 148.63824844360352
Array Size: 3000 | Key Comparisons: 2216462 | Execution Time (ms): 207.95583724975586
Array Size: 3500 | Key Comparisons: 3095733 | Execution Time (ms): 310.560941696167
Array Size: 4000 | Key Comparisons: 4014570 | Execution Time (ms): 378.2956600189209
Array Size: 4500 | Key Comparisons: 5127746 | Execution Time (ms): 487.5447750091553
Array Size: 5000 | Key Comparisons: 6229774 | Execution Time (ms): 586.1270427703857
Array Size: 5500 | Key Comparisons: 7454418 | Execution Time (ms): 817.6400661468506
Array Size: 6000 | Key Comparisons: 9043109 | Execution Time (ms): 1002.1719932556152
Array Size: 6500 | Key Comparisons: 10558580 | Execution Time (ms): 1140.1309967041016
Array Size: 7000 | Key Comparisons: 12265153 | Execution Time (ms): 1355.315923690796
Array Size: 7500 | Key Comparisons: 14007632 | Execution Time (ms): 1494.0471649169922
Array Size: 8000 | Key Comparisons: 16041583 | Execution Time (ms): 1547.7559566497803
Array Size: 8500 | Key Comparisons: 18065665 | Execution Time (ms): 1805.2279949188232
Array Size: 9000 | Key Comparisons: 20286903 | Execution Time (ms): 2183.87508392334
Array Size: 9500 | Key Comparisons: 22606873 | Execution Time (ms): 2114.346981048584

Process finished with exit code 0
```

4. Estimation for a Randomly Generated Array of Size 10,000:

To estimate the number of key comparisons and the running time for an array of size 10,000, we can use the derived average-case formula:

1. **Theoretical Key Comparisons**:

   Average Comparisons = $\frac{10000 \times (10000 - 1)}{4} = \frac{10000 \times 9999}{4} = 24997500$

2. **Running Time**: From the empirical data obtained

```
/usr/bin/python3 /Users/digvijay/PycharmProjects/daaProjects/Project1Comparision.py
Size: 10000 | Key Comparisons: 25172759 | Time (ms): 2550.5218505859375

Process finished with exit code 0
```