Insertion Sort vs Quick Sort

Insertion Sort and Quick Sort are two of the most used sorting algorithms. Insertion sort has an expected run time complexity of $\Theta(n^2)$, while Quick Sort has an expected time complexity of $\Theta(n \log n)$. At first glance, it may seem that Quick Sort will always be the faster sorting algorithm, but in practice, the results may differ based on the size of the input.

Hypothesis

Given the theoretical time complexities of insertion sort and quick sort, it seems that quick sort will outperform insertion sort. However, in practice, the performance these sorting algorithms may vary based on the size of *n*.

Insertion sort will likely outperform quicksort for lower values of *n* (<15), due to less overhead in terms of code. Insertion sort only needs to compare values and make swaps, while Quick Sort must compare and swap values, do recursive calls, and use a partitioning function. As a result, Insertion Sort will have lower time and space complexity than Quick Sort for small values of *n*.

On the other hand, Quick Sort will likely outperform Insertion Sort for large data sets due to its faster time complexity. For large values of *n*, the initial overhead to implement Quick Sort will be negligible compared to the performance of $O(n \log n)$ over Insertion Sort's $\Theta(n^2)$. This is assuming that a good partitioning scheme is chosen, and the data is not manipulated to abuse the partitioning scheme.

Methods

Full code can be found at [this GitHub repo]

Design: For this study, we compared run times to execute insertion sort and quick sort on arrays of varying sizes of *n*, ranging from 5 to 200. The array data was populated using random number generation with a constant seeding value to ensure results were unbiased. An algorithm for each sort is included in the source code. The recursive quicksort function was defined as follows:

```
def quicksort(data):
    """
    https://github.com/sidb70/CSE-331/blob/main/Projects/Project%204%20Sorting%20algorithms/solution.py
    Sorts a list in place using quicksort
    :param data: Data to sort
    """

    def quicksort_inner(first, last):
        """
        Sorts portion of list at indices in interval [first, last] using quicksort
        :param first: first index of portion of data to sort
        :param last: last index of portion of data to sort
        """
```

```
    # List must already be sorted in this case
    if first >= last:
        return

    left = first
    right = last

    # Need to start by getting median of 3 to use for pivot
    # We can do this by sorting the first, middle, and last elements
    midpoint = (right - left) // 2 + left
    if data[left] > data[right]:
        data[left], data[right] = data[right], data[left]
    if data[left] > data[midpoint]:
        data[left], data[midpoint] = data[midpoint], data[left]
    if data[midpoint] > data[right]:
        data[midpoint], data[right] = data[right], data[midpoint]
    # data[midpoint] now contains the median of first, last, and middle elements
    pivot = data[midpoint]
    # First and last elements are already on right side of pivot since they are sorted
    left += 1
    right -= 1

    # Move pointers until they cross
    while left <= right:
        # Move left and right pointers until they cross or reach values which could be swapped
        # Anything < pivot must move to left side, anything > pivot must move to right side
        #
        # Not allowing one pointer to stop moving when it reached the pivot (data[left/right] == pivot)
        # could cause one pointer to move all the way to one side in the pathological case of the pivot being
        # the min or max element, leading to infinitely calling the inner function on the same indices without
        # ever swapping
        while left <= right and data[left] < pivot:
            left += 1
        while left <= right and data[right] > pivot:
            right -= 1
        # Swap, but only if pointers haven't crossed
        if left <= right:
            data[left], data[right] = data[right], data[left]
            left += 1
            right -= 1
    quicksort_inner(first, left - 1)
```

```
        quicksort_inner(left, last)

    # Perform sort in the inner function
    quicksort_inner(0, len(data) - 1)
```

The insertion sort algorithm was as follows:

```
def insertion_sort(arr):
    """
    In place implementation of insertion sort
    :param arr: List of integers
    :return: Sorted list of integers
    """
    for i in range(len(arr)):
        curr = arr[i]
        j = i
        while j > 0 and arr[j - 1] > curr:
            arr[j] = arr[j - 1]
            j -= 1
        arr[j] = curr
```

Data collection:

For each sorting algorithm, a dictionary was used to record runtimes for various values of *n*. They were initialized as follows:

```
    # Create a dictionary mapping n to a list of run times for each algorithm
    quicksort_runtimes = {5: [],10:[],15:[],20:[],30:[], 40:[], 50: [],100:[],200:[]}
    insertion_sort_runtimes ={5: [],10:[],15:[],20:[],30:[],40:[], 50: [],100:[],200:[]}
```

The keys for the dictionaries correspond to *n* values, and the values are lists to be populated with run times for the respective n.

The measure_times function was used to populate the run times data set. It is given a pointer to the function we want to measure as well as the dictionary of its runtimes. For every value of *n* we are testing, an array of *n* random integers between 0 and 1000 are placed into an array that we will sort. Python's random module was used to generate the random numbers. The seed value is always set to zero prior to generating the numbers to get the same numbers for every time measure_times is called. The current time is recorded, then the given function is invoked with the array of integers. The end time is recorded, then the run time averaged for each element is added to the dataset.

```
def measure_times(function,sizes):
```

```
    """
    Measures the run time of a function for different sizes of input
    :param function: Function to measure
    :param sizes: Dictionary mapping size of input to list of run times
    """
    random.seed(0)
    for size in sizes.keys():
        arr = [random.randint(0, 1000) for _ in range(size)]
        start = time.time()
        function(arr)
        end = time.time()
        sizes[size].append((end - start)/size)
```

To get sufficient data, the measure_times was called 20 times for each sorting algorithm. This means that for each value of *n*, both algorithms run times were recorded 20 times.

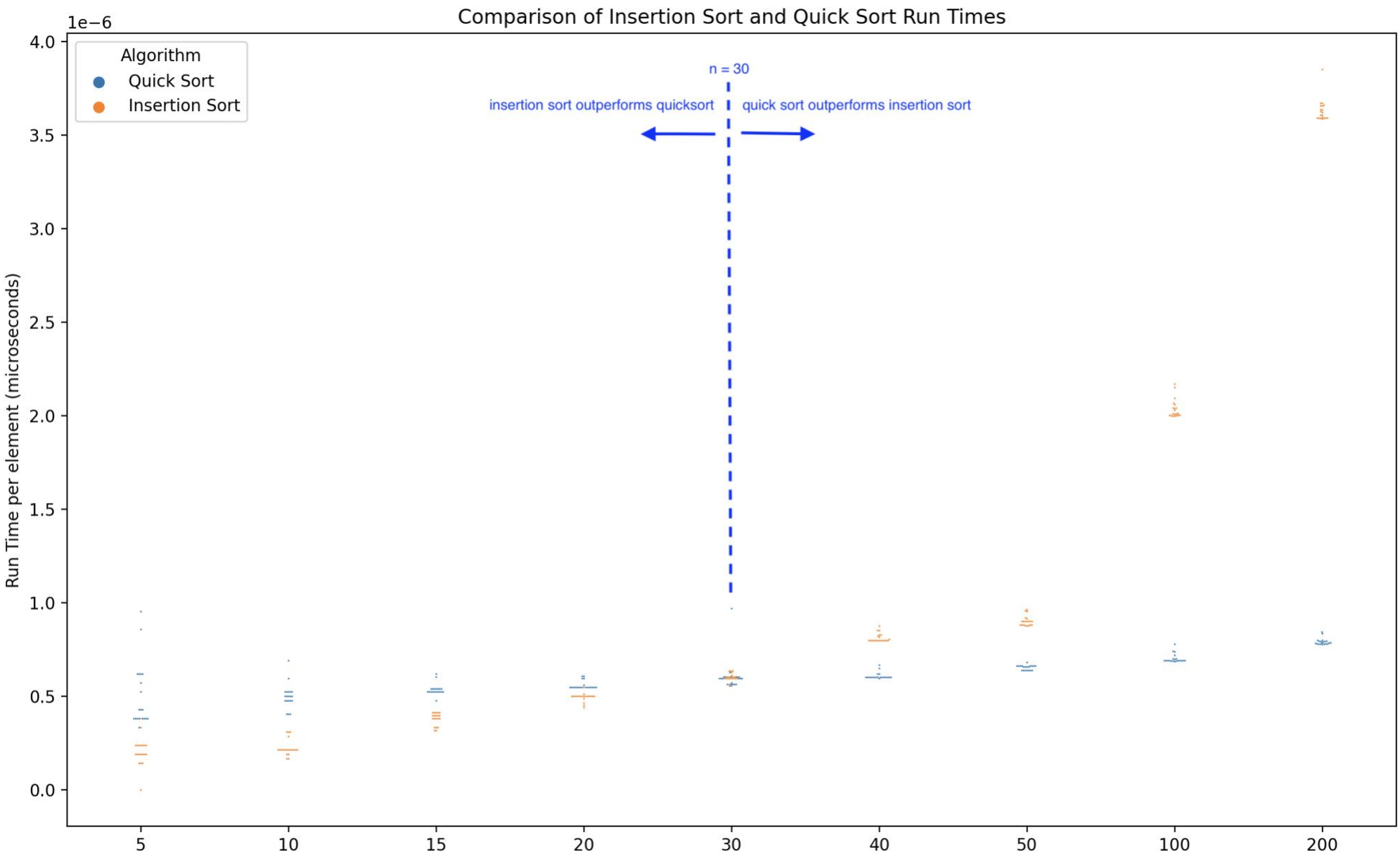
```
# Populate each algorithm's dictionary with 20 run time tests for each n value
for i in range(20):
    measure_times(quicksort,quicksort_runtimes)
    measure_times(insertion_sort,insertion_sort_runtimes)
```

Finally, a pandas DataFrame was created to correctly format the datasets to graph using matplotlib swarmplot.

The program was executed on a 2020 MacBook Pro with an Apple M1 processor with 8GB of main memory.

Results

The run times of insertion sort and quick sort were recorded for arrays of varying sizes ranging from 5 to 200. The run times for reach algorithm were recorded 20 times for each value of n to obtain sufficient data. A pandas DataFrame was used to organize the visualization using matplotlib swarmplot. The results are contained in the following graph:


Comparison of Insertion Sort and Quick Sort Run Times
1e−6
Algorithm
Quick Sort
Insertion Sort
n = 30
insertion sort outperforms quicksort
quick sort outperforms insertion sort
Run Time per element (microseconds)
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
5
10
15
20
30
40
50
100
200

The *x*-axis corresponds to the *n*-values we tested, from 5 to 200. The *y*-axis corresponds to the average run time per element in units of microseconds. The results for insertion sort were plotted with orange points and quicksort with blue. The dotted line indicates the value of *n* at which quick sort begins to outperform insertion sort.

Discussion

Overall, the results of this experiment support our hypothesis that the performance of the sorting algorithms will vary based on the size of the input, and that insertion sort will outperform quicksort at small values of *n*. The results of the experiment show that for inputs ($n < 30$), insertion sort will outperform quick sort. This is due to insertion sort's low overhead and simple algorithm, in contrast quicksort's many recursive calls and partitioning function. However, as the input sizes grew ($n > 30$), quick sort's faster time complexity of $O(n \log n)$ allows it to outperform insertion sort's $\Theta(n^2)$ time complexity. For large values of *n*, insertion sort's run time increases at a quadratic rate compared to quick sort, with insertion sort's run time per element at *n=30* being ~3.5 microseconds, compared to ~.5 microseconds per element for quick sort.

The point of interest for the experiment was *n=30*, which is where the results show that insertion sort and quicksort perform at practically the same run time. The swarm for each algorithm was indistinguishable from the other. This experiment found that the value of *n* under which insertion sort will outperform quicksort was 30. Overall, this validated the hypothesis of a small value of n such as 15. Further testing on different processors and computers would likely yield slightly different results.

Conclusion

Under the conditions tested, the insertion sort algorithm outperforms quick sort for $n < 30$, while quick sort outperforms insertion sort for $n > 30$. For $n = 30$, the difference is not distinguishable.

Sources and Tools:

https://seaborn.pydata.org/generated/seaborn.swarmplot.html

Quick Sort Source Code - Github link

Full source code - Github link