Hybrid Sort Performance

Hybrid Sorts are commonly used to take advantage of algorithms that perform better at small input sizes and handle large input sizes using an algorithm with better runt time complexity. In this experiment, we will use the insertion sort and quick sort algorithms from the previous experiment to find the optimal pivot point for input size that our hybrid sort should switch between the algorithms.

Hypothesis

Our previous experiment found Insertion Sort to perform poorly on large input sizes due to its quadratic $\Theta(n^2)$ time complexity, much slower than Quick Sort's $O(n \log n)$ time complexity. However, Insertion sort performed better than Quick Sort on small data sets due to its simple code and low memory overhead. The value of *n* at which Quick Sort starts to outperform Insertion sort was *n*=30.

Therefore, we hypothesize that by creating a hybrid sorting algorithm that uses insertion sort and quick sort, along with a well-chosen pivot value, *k*, will outperform both algorithms by themselves. Quick sort will be used for partitions of size > *k* and will sort smaller partitions of size <= *k* with insertion sort.

We also hypothesize that the optimal value for *k* will be 30, which is consistent with our previous findings that show that insertion sort outperforms quicksort for *n < 30.* This means our hybrid sort algorithm can take advantage of insertion sort for the small values of *n* that it performs well for, while still reaping the benefits of quick sort for larger input sizes.

To test our hypothesis, we will measure run time complexity on a hybrid sort algorithm on a large range of *k*-values as well as *n* to determine two things:

a) What is the optimal value for *k* to use in our implementation of hybrid sort?
b) Does input size, *n*, affect what value of *k* will perform the best?

Methods

Full code can be found at [this GitHub repo]()

Design: For this study, we compare the run time performance of hybrid sort based on different k values. For each value of k, we run time tests on input sizes, *n*, ranging from 5 to 200. The data set given to the sorting algorithm is a list of random integers from 1 to 1000, with the same seeding value on each test to ensure the dataset is consistent.

The hybrid sort algorithm is very similar to our code for the previous study, with one small change. Within our partition function, quicksort_inner, after checking if the left and right pointers have crossed, we also check if the size of the partition (right – left) is less than or equal to our given k-value. If it is, then we will use insertion sort to sort this partition. Our hybrid sort algorithm is given as follows:

```
def quicksort_hybrid(data, k):
    """
    https://github.com/sidb70/CSE-331/blob/main/Projects/Project%204%20Sorting%20algorithms/solution.py
```

```
Sorts a list in place using a hybrid of quicksort and insertion sort
:param data: Data to sort
:param k: Threshold for switching to insertion 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

    if right - left <= k:
        insertion_sort(data[left:right + 1])
        return

    # 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:

To collect our test data, we created a data structure that would allow us to test a wide range of *k* values as well as a wide range of *n* for every *k*. We created a dictionary of dictionaries, in which the top level keys corresponded to values of *k* from 10 to 100. Each *k* in the dictionary corresponds to a dictionary of *n* values ranging from 10 to 200. When we test each value of *k*, run time tests will be executed for every value of *n*, and the results will be placed in the lists that correspond to the respective *n* values. The data structure was initialized as follows:

```
hybrid_sort_runtimes = {}
for i in range(5,21):
    hybrid_sort_runtimes[i*5] = {20:[], 30:[], 50:[], 100:[], 200:[]}
```

We then iterate over each *k*-value and call our run time test for the specified *k*.

```
for k in hybrid_sort_runtimes.keys():
    measure_hybrid_sort(k, hybrid_sort_runtimes)
```

Our measure_hybrid_sort will execute 10 run time tests on an array of size *n* for every *n* the inner level data structure (*n=20,30,50,100,200*). It will then populate the data structure with the average run time per element for the respective test.

```
def measure_hybrid_sort(k, runtimes):
    """
    Measures the run time of hybrid sort on lists of random integers of size n
    :param function: Function to measure
    :param sizes: Dictionary mapping size of input to list of run times
    """
     #
    random.seed(0)
    for size in runtimes[k].keys():
        for _ in range(10):
            arr = [random.randint(0, 1000) for _ in range(size)]
            start = time.time()
            quicksort_hybrid(arr,k)
            end = time.time()
            runtimes[k][size].append((end - start)/size)
```

Finally, for every *n* tested for every *k*, we will take the average of the 10 tests to plot on our graph. We can then construct a pandas dataframe that we can use with matplotlib's lineplot.

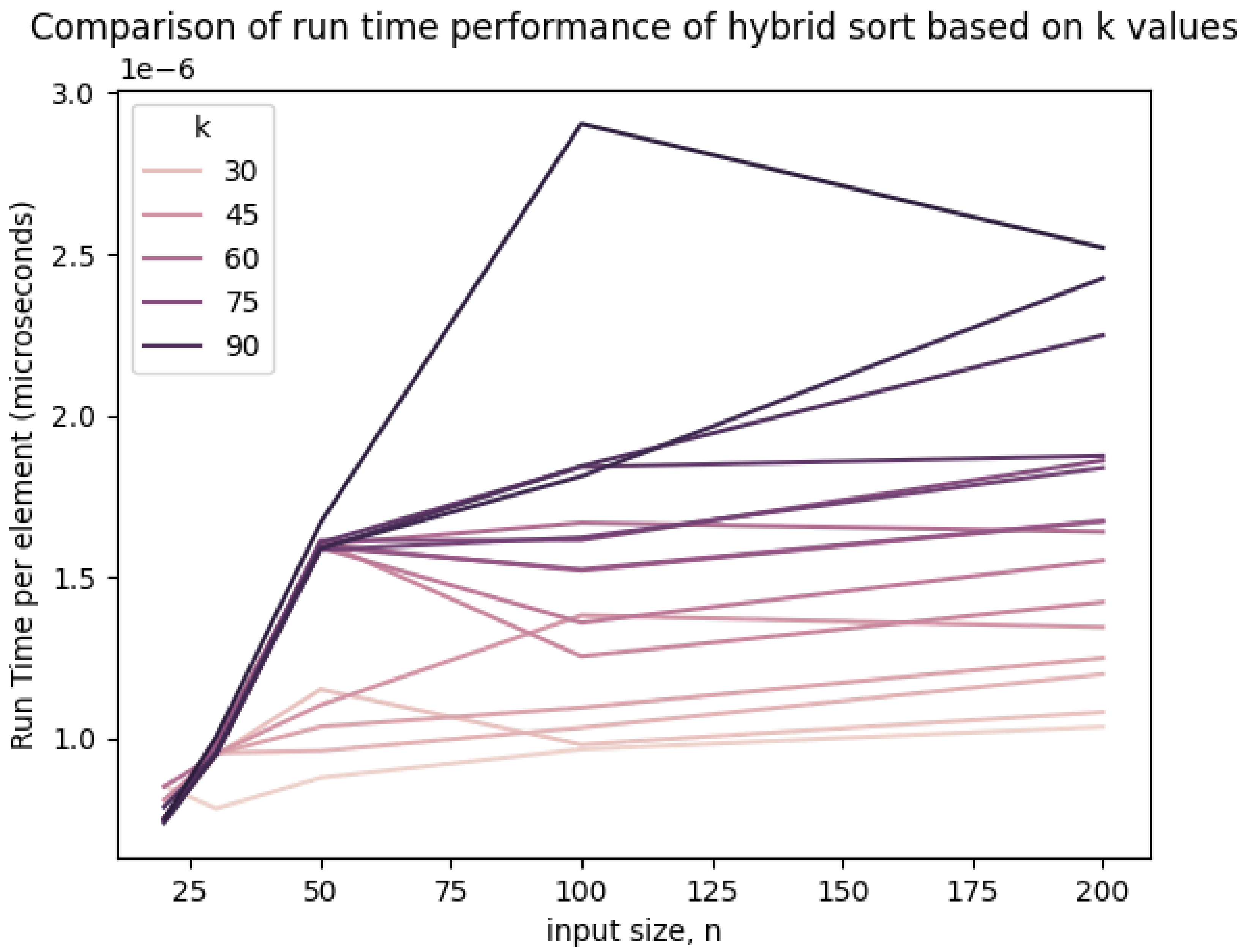
```
# Average results
for k in hybrid_sort_runtimes.keys():
    for size in hybrid_sort_runtimes[k].keys():
        # Average the 10 run times for each size
        hybrid_sort_runtimes[k][size] = sum(hybrid_sort_runtimes[k][size])/len(hybrid_sort_runtimes[k][size])
```

```
# Plot results
df = pd.DataFrame(hybrid_sort_runtimes)
df = df.transpose()
df = df.reset_index()
df = df.melt(id_vars=['index'], value_vars=[20,30,50,100,200])
df.columns = ['k', 'n', 'time/element']
sns.lineplot(data=df, x='n', y='time/element', hue='k')
plt.title('Comparison of run time performance of hybrid sort based on k values')
plt.xlabel('input size, n')
plt.ylabel('Run Time per element (microseconds)')
plt.savefig('hybrid_sort.png')
plt.show()
```

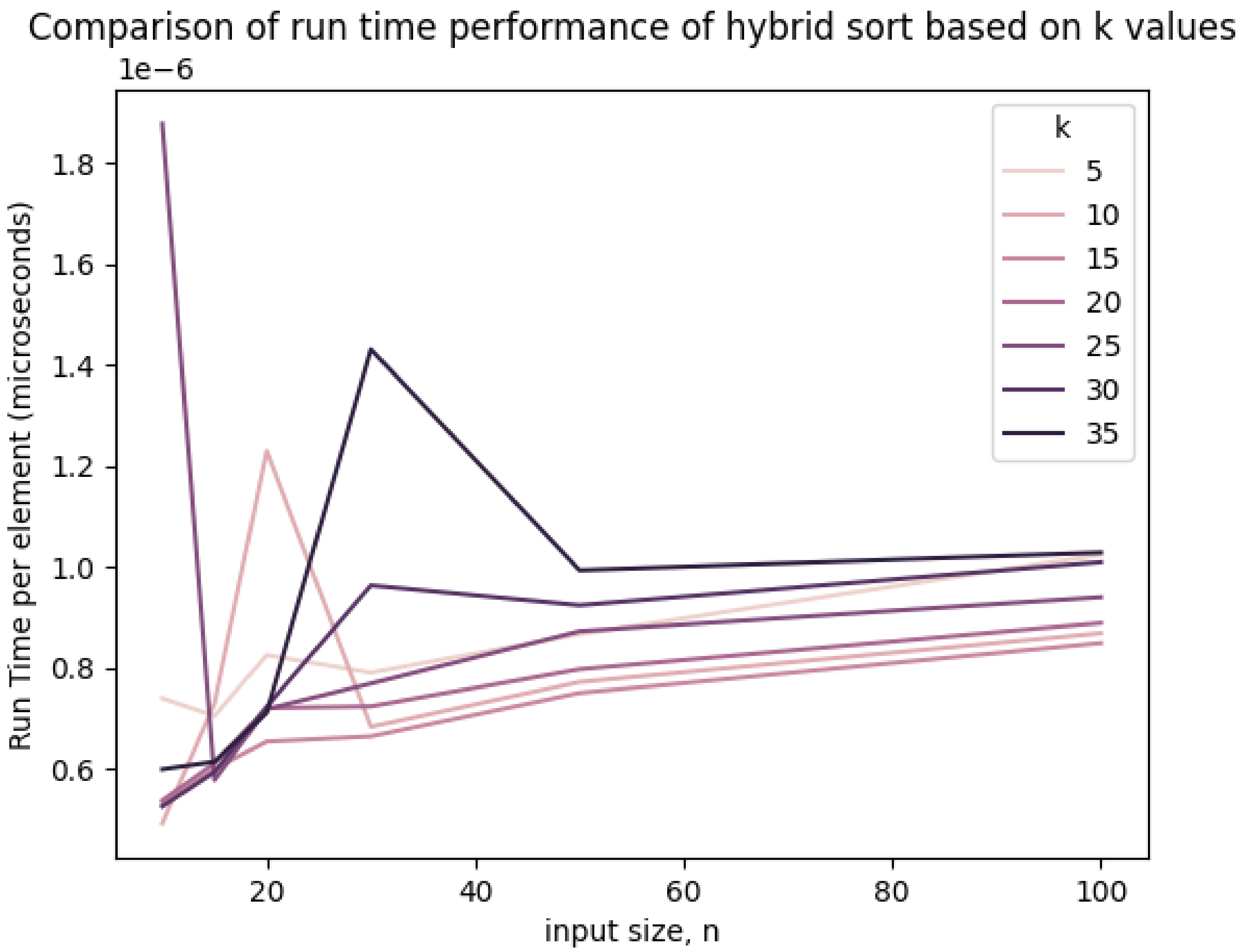
The program was executed on a 2020 MacBook Pro with an Apple M1 processor with 8GB of main memory.

Results

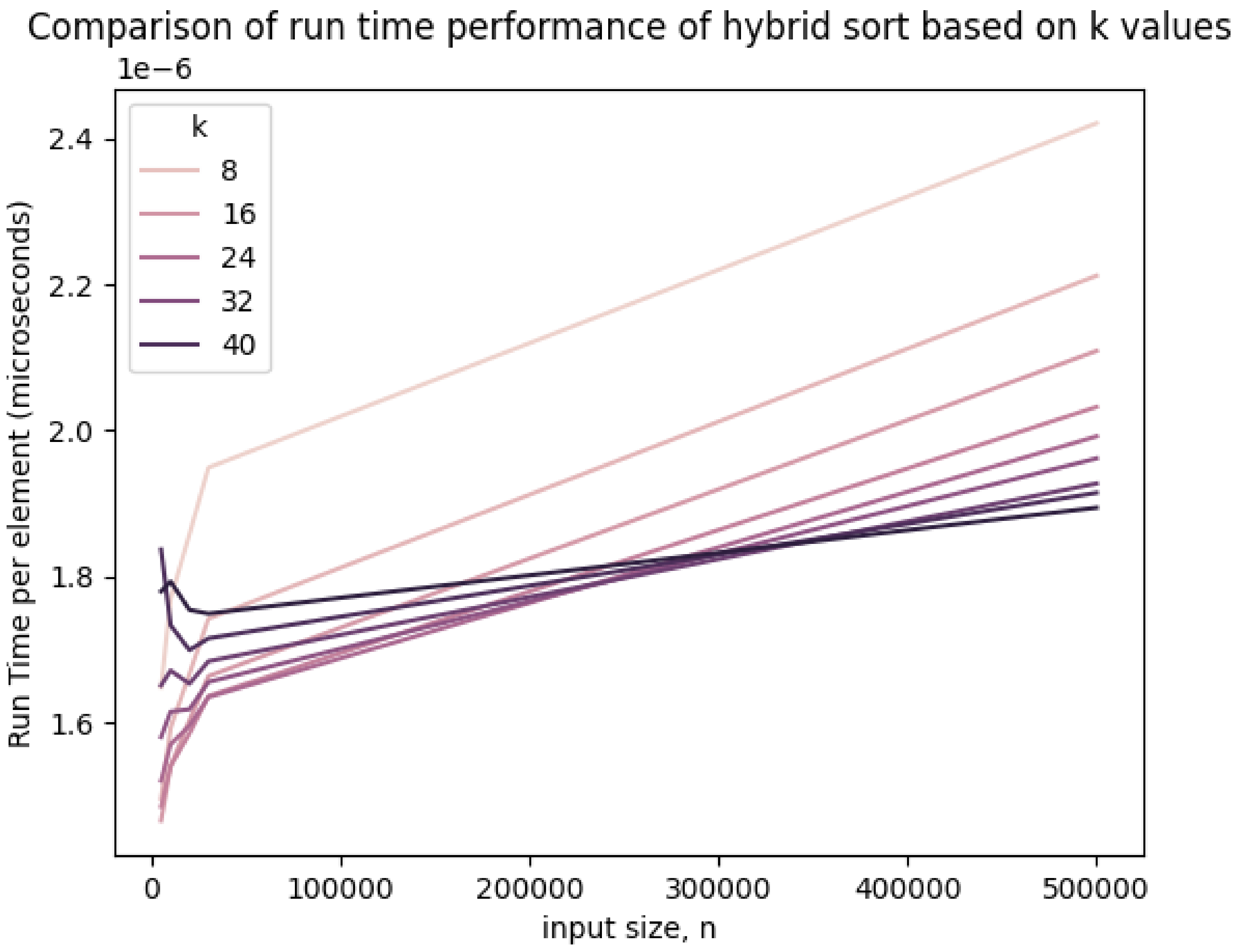
The run times for a wide range of *k* values from 20 to 100 were initially tested. For each *k*, we also tested a wide range of input sizes, from 20 to 200. Data sets for each input size were identical to ensure consistency between the tests. Tests for every input size was ran 10 times to provide accurate results, then the average was taken of these results to get closer to the true mean. The results of the initial test is given in the following graph:


Comparison of run time performance of hybrid sort based on k values
1e−6
3.0
2.5
2.0
1.5
1.0
Run Time per element (microseconds)
k
30
45
60
75
90
25
50
75
100
125
150
175
200
input size, n

The results for larger values of *n* was clear, but we decided to look more in-depth at what was happening for smaller values of *k*. We recreated the test, but bounded *k* between 5 and 40. We testing up to *n*=100. The results were as follows:


Comparison of run time performance of hybrid sort based on k values
1e−6
Run Time per element (microseconds)
0.6
0.8
1.0
1.2
1.4
1.6
1.8
input size, n
20
40
60
80
100
k
5
10
15
20
25
30
35

Finally, we decided to test our hybrid sort at very large input sizes (5000 to 50,000) to see if large *n* values will affect which *k* value will be optimal. The results of this final test are as follows:


Comparison of run time performance of hybrid sort based on k values
1e−6
k
8
16
24
32
40
2.4
2.2
2.0
1.8
1.6
Run Time per element (microseconds)
0
100000
200000
300000
400000
500000
input size, n

Discussion

From our first test, we tested k between 30 and 100 with n between 10 and 200. The result of this experiment shows that as the input sizes grew, it was clear that k = 30 was performing much better than the higher k values. This validates our hypothesis of an optimal k value of 30. However, we felt that we should test a larger range of k as well as n. The next test expanded our range of k to 5 to 40 and bounded our n from 10 to 100. The results of this test were surprising. At small n (10 to 15), all k values performed similarly, but from n > 20, it was clear that k = 15 was the most performant of the pivot points tested, while the hypothesized k = 30 performed the poorest. Because k = 15 outperformed values of k < 15 as well as k > 15, this indicates that k = 15 is the optimal pivot for small values of n (under 200).

Because our hypothesis of k = 30 was rejected by the tests indicating that k = 15 was optimal, we decided to do one final test, with larger input sizes and large range of k. We tested 5000 <= n <= 50,000 with 5 <= k =< 50.

The results of this final experiment were more surprising. We found that smaller values of k , under 20, were not as performant for large input sizes. As n increased, the larger values of k had the lowest run times, until k = 40 was the most performant at our largest input size of 50,000.

So, putting these results together tells us that the optimal value of k for hybrid sort varies based on the input size. For small input sizes (n within the magnitude of a few hundred elements), smaller values for the pivot point (k = 15) are the most performant for our hybrid sort, while as n grows very large (past 10,000), larger values of k outperform the smaller pivot points.

As the data sets get larger, larger k values outperformed the smaller ones because less partitions are required with a large k. For smaller data sets, smaller k values were able to take advantage of insertion sort's fast run time on small partitions. Thus, the optimal value for k depends on the size of the data set.

Conclusion

Under the conditions tested, the optimal value for k depends heavily on the size of the input. For most data sets, a value of k = 15 will balance speed and overhead. However, for very large data sets, k must be increased appropriately.

Sources and Tools:

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

Quick Sort Source Code - Github link

Full source code - Github link