DATA 295P Project 1 Report

April 24, 2024

0.1 DATA260P Project 1: Comparing Sorting Algorithms

Connor McManigal and Peyton Politewicz

[ ]:
```
import pandas as pd
import numpy as np

tr_df = pd.read_csv('tr_table.csv')
as_df = pd.read_csv('as_table.csv')

def get_theoretical_big_o(algo):
    if algo in ['Merge', 'Simple Tim']:
        return 'n log n'
    elif algo in ['Quick', 'Insertion', 'Shell731', 'Shell1000', 'Binary␣
↪Insertion']:
        return 'n^2'
    elif algo == 'Radix':
        return 'nd'
    elif algo == 'Bucket':
        return 'n'
    else:
        return 'Unknown'  # Just in case I mess up

tr_df['Theoretical Big-O'] = tr_df['Algo'].apply(get_theoretical_big_o)
as_df['Theoretical Big-O'] = as_df['Algo'].apply(get_theoretical_big_o)
```

[ ]:
```
print(tr_df)
```

```
              Algo  Data Size  Observed Runtime     Ratio  Emp Big-O  \
0            Merge       1000          0.001358       NaN        NaN
1            Merge       2000          0.002968  2.185281   1.127819
2            Merge       4000          0.006353  2.140799   1.098149
3            Merge       8000          0.013436  2.114739   1.080480
4            Merge      16000          0.028682  2.134772   1.094082
5            Quick       1000          0.000983       NaN        NaN
6            Quick       2000          0.002183  2.220533   1.150906
7            Quick       4000          0.004996  2.288156   1.194186
8            Quick       8000          0.012189  2.439670   1.286686
9            Quick      16000          0.031078  2.549712   1.350334
10       Insertion       1000          0.014235       NaN        NaN
```

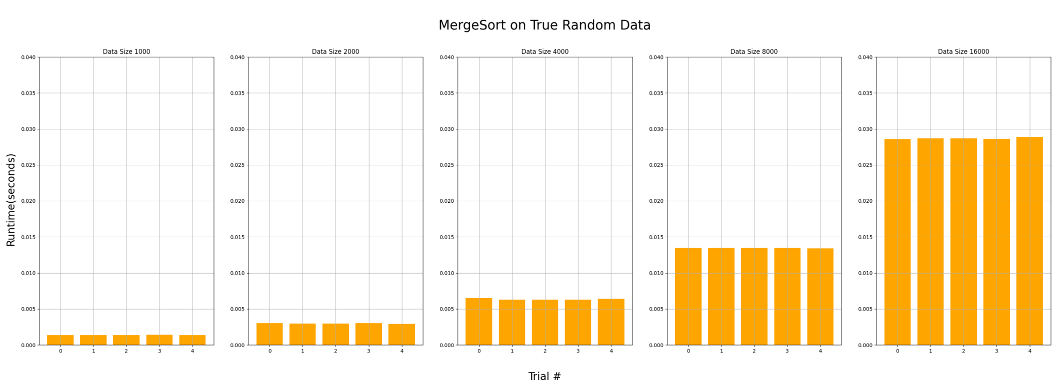
11	Insertion	2000	0.059809	4.201434	2.070882
12	Insertion	4000	0.237579	3.972304	1.989976
13	Insertion	8000	0.959732	4.039635	2.014225
14	Insertion	16000	3.863937	4.026060	2.009369
15	Shell731	1000	0.004920	NaN	NaN
16	Shell731	2000	0.018754	3.811846	1.930490
17	Shell731	4000	0.072155	3.847520	1.943929
18	Shell731	8000	0.280652	3.889582	1.959615
19	Shell731	16000	1.113573	3.967800	1.988339
20	Shell1000	1000	0.003393	NaN	NaN
21	Shell1000	2000	0.010192	3.003536	1.586662
22	Shell1000	4000	0.028034	2.750647	1.459771
23	Shell1000	8000	0.078536	2.801432	1.486165
24	Shell1000	16000	0.220780	2.811172	1.491172
25	Bucket	1000	0.000166	NaN	NaN
26	Bucket	2000	0.000308	1.854998	0.891417
27	Bucket	4000	0.001994	6.475608	2.695016
28	Bucket	8000	0.000865	0.434106	-1.203880
29	Bucket	16000	0.001611	1.861552	0.896506
30	Radix	1000	0.000544	NaN	NaN
31	Radix	2000	0.001131	2.079548	1.056270
32	Radix	4000	0.002226	1.968889	0.977382
33	Radix	8000	0.004393	1.973605	0.980833
34	Radix	16000	0.008782	1.998955	0.999246
35	Binary Insertion	1000	0.001888	NaN	NaN
36	Binary Insertion	2000	0.005836	3.091340	1.628233
37	Binary Insertion	4000	0.021105	3.616503	1.854595
38	Binary Insertion	8000	0.090807	4.302527	2.105184
39	Binary Insertion	16000	0.382527	4.212543	2.074692
40	Simple Tim	1000	0.001106	NaN	NaN
41	Simple Tim	2000	0.002461	2.225071	1.153852
42	Simple Tim	4000	0.005390	2.190182	1.131051
43	Simple Tim	8000	0.011606	2.153265	1.106526
44	Simple Tim	16000	0.025035	2.156951	1.108993

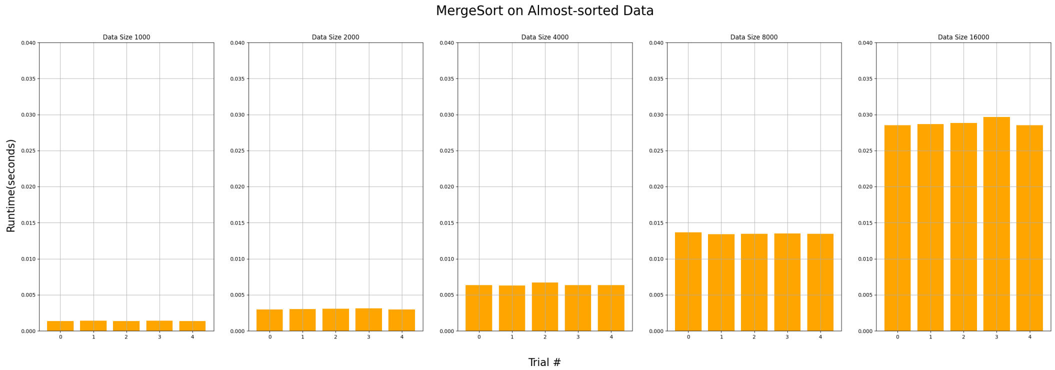
	Theoretical Big-O
0	n log n
1	n log n
2	n log n
3	n log n
4	n log n
5	n^2
6	n^2
7	n^2
8	n^2
9	n^2
10	n^2
11	n^2

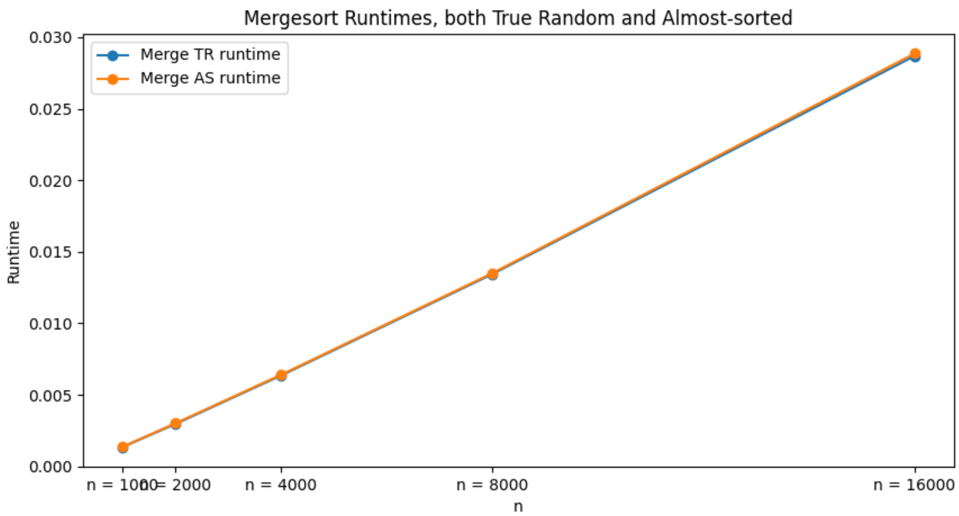
```
12              n^2
13              n^2
14              n^2
15              n^2
16              n^2
17              n^2
18              n^2
19              n^2
20              n^2
21              n^2
22              n^2
23              n^2
24              n^2
25                n
26                n
27                n
28                n
29                n
30               nd
31               nd
32               nd
33               nd
34               nd
35              n^2
36              n^2
37              n^2
38              n^2
39              n^2
40          n log n
41          n log n
42          n log n
43          n log n
44          n log n
```

0.2 Experimental Time Analysis

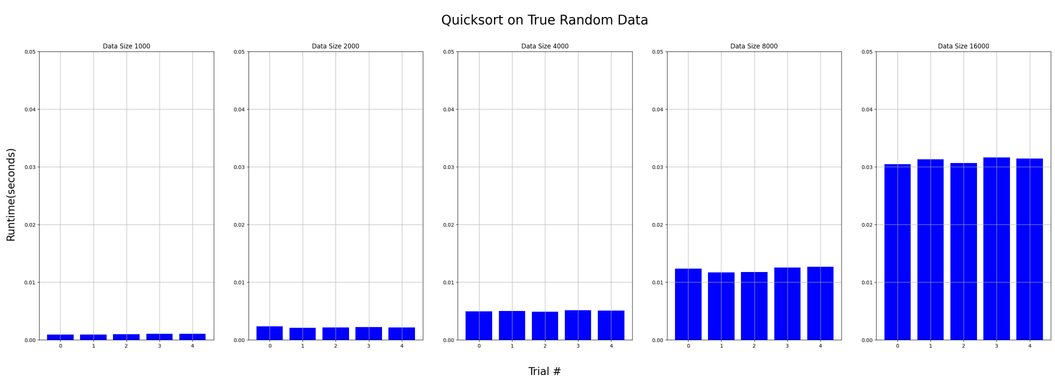
0.2.1 MergeSort Time Analysis

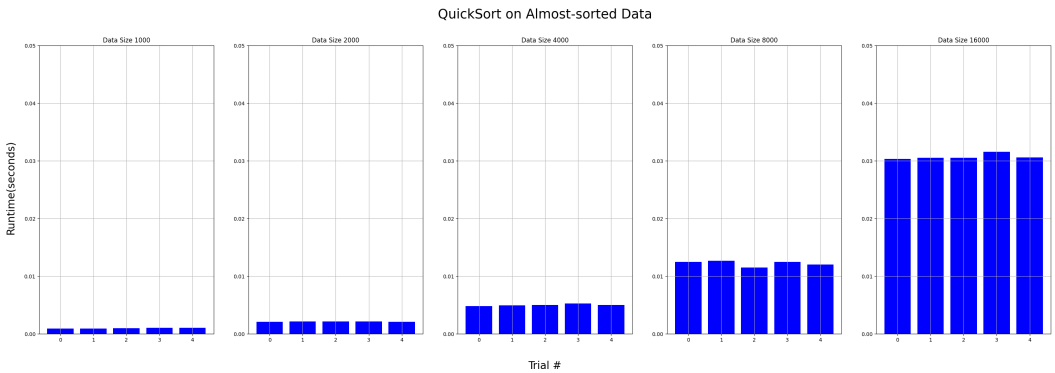


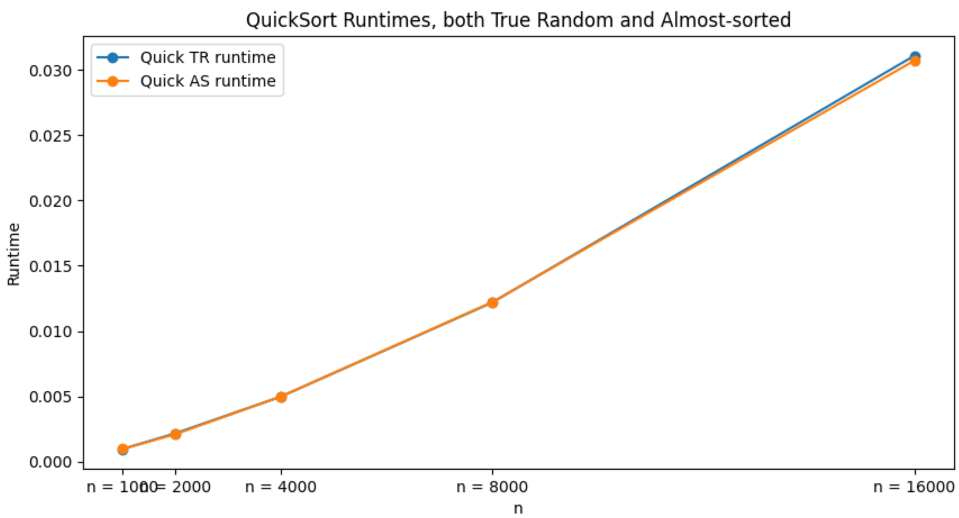




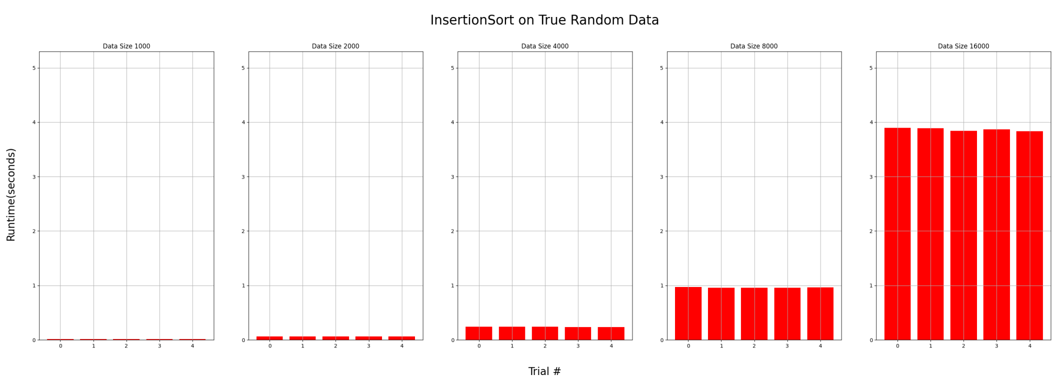
0.2.2 QuickSort Time Analysis

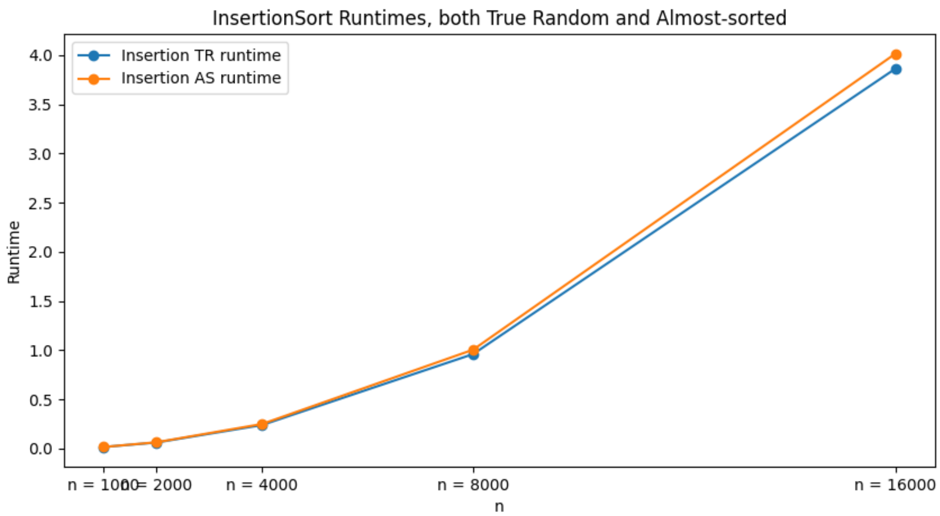




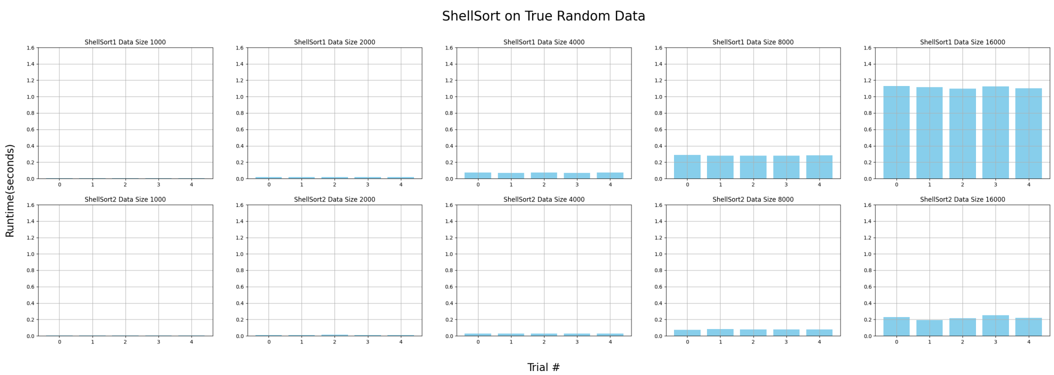


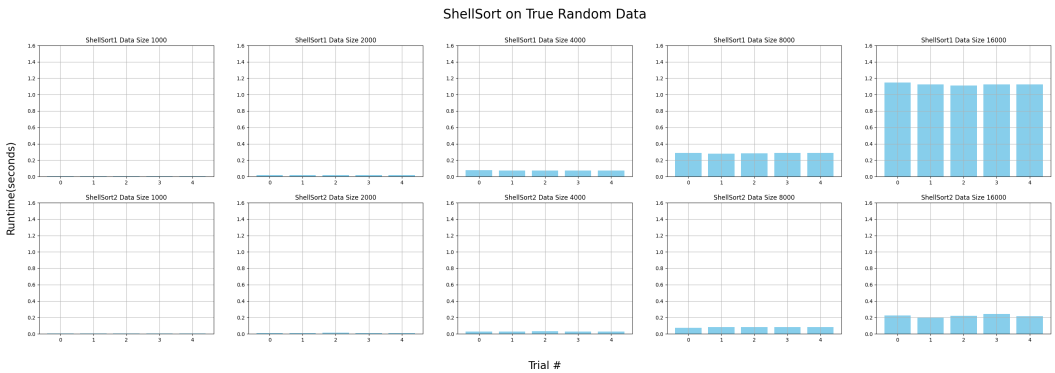
0.2.3 InsertionSort Time Analysis

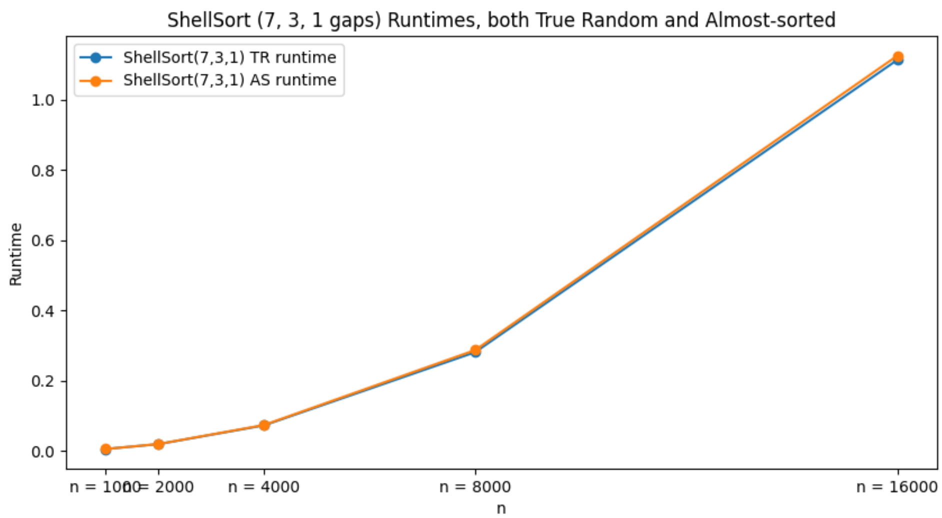


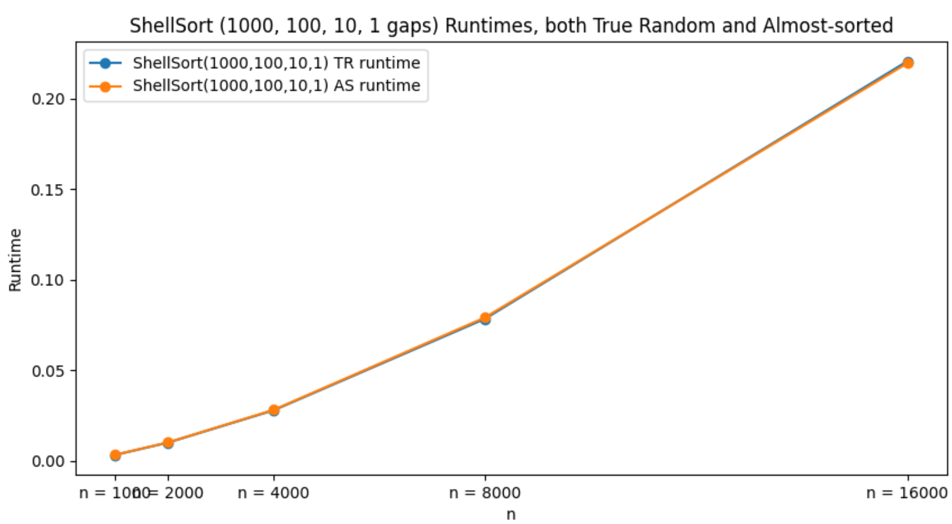


0.2.4 ShellSort Time Analysis

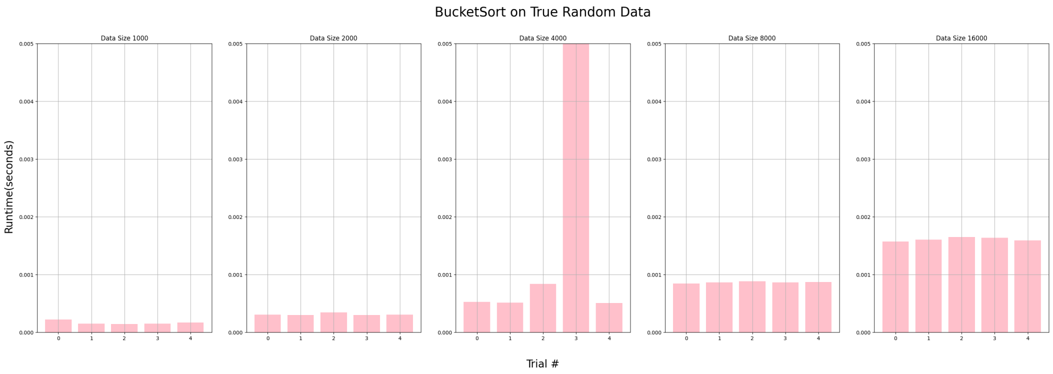


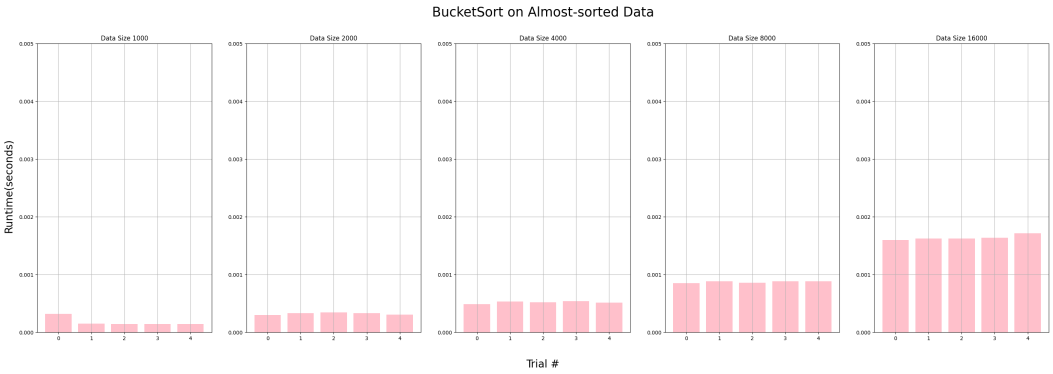

ShellSort on True Random Data
ShellSort1 Data Size 1000
ShellSort1 Data Size 2000
ShellSort1 Data Size 4000
ShellSort1 Data Size 8000
ShellSort1 Data Size 16000
ShellSort2 Data Size 1000
ShellSort2 Data Size 2000
ShellSort2 Data Size 4000
ShellSort2 Data Size 8000
ShellSort2 Data Size 16000
Runtime(seconds)
Trial #

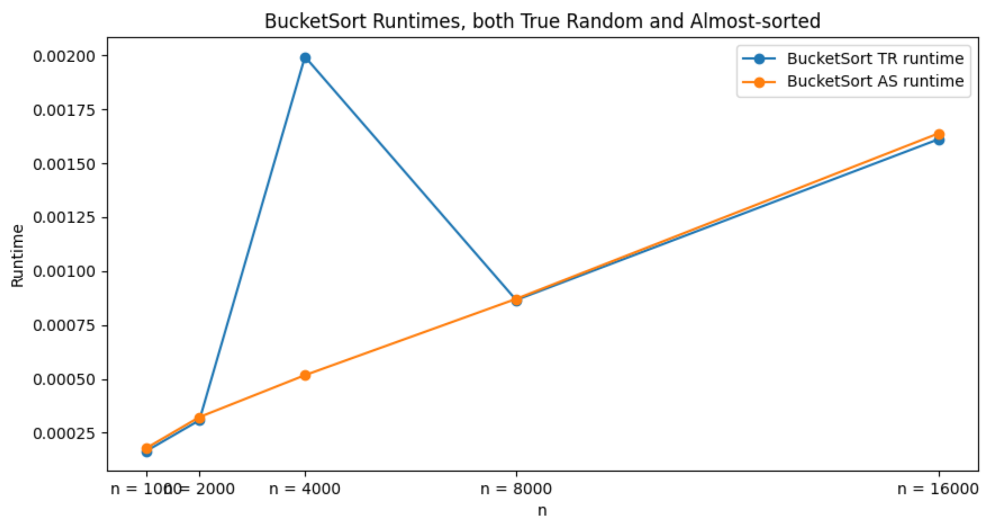

ShellSort (7, 3, 1 gaps) Runtimes, both True Random and Almost-sorted
ShellSort(7,3,1) TR runtime
ShellSort(7,3,1) AS runtime
Runtime
0.0
0.2
0.4
0.6
0.8
1.0
n = 1000
n = 2000
n = 4000
n = 8000
n = 16000
n


ShellSort (1000, 100, 10, 1 gaps) Runtimes, both True Random and Almost-sorted
ShellSort(1000,100,10,1) TR runtime
ShellSort(1000,100,10,1) AS runtime
Runtime
0.00
0.05
0.10
0.15
0.20
n = 1000
n = 2000
n = 4000
n = 8000
n = 16000
n

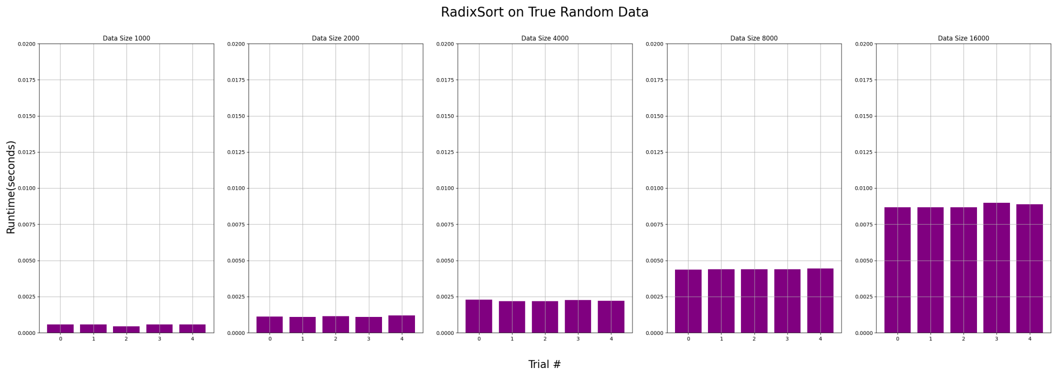
0.2.5 BucketSort Time Analysis

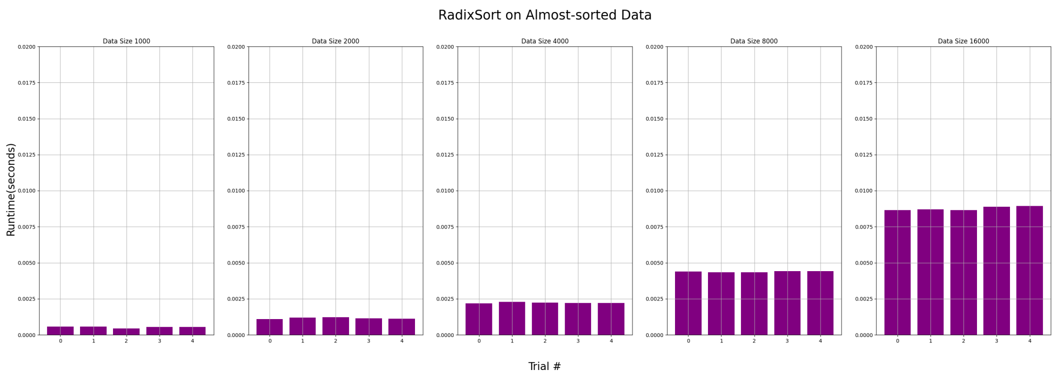


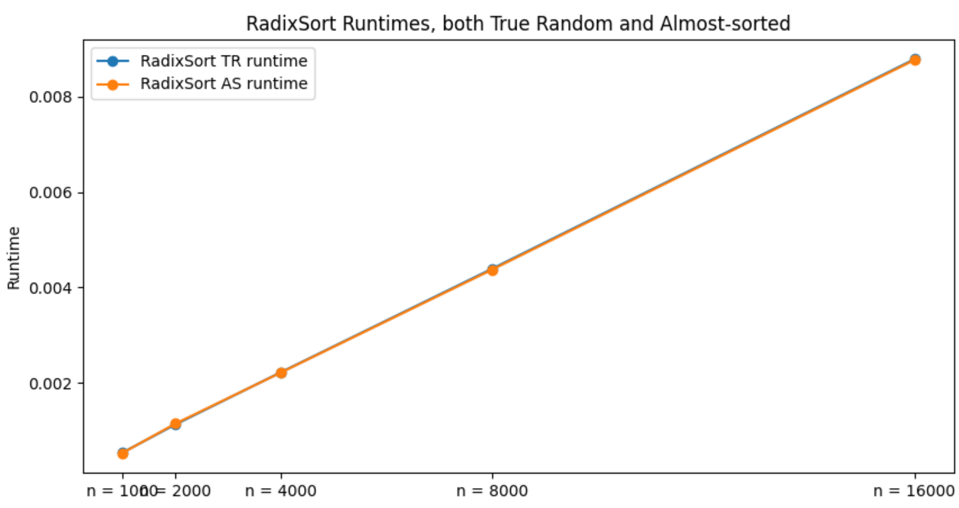




0.2.6 RadixSort Time Analysis







0.2.7 BinaryInsertionSort Time Analysis

I wrote the BinaryInsertionSort algorithm in an effort to improve runtime from the slow and clunky InsertionSort implementation(it appeared to be the slowest of our algorithms). After running InsertionSort and observing ~4 second runtimes on the larger data size(16000), I wanted to find an approach that could drastically enhance its performance on large dataset sizes. I used two helper functions, one to perform the binary search to find the correct position to insert an element into the sorted subarray(binary_search()) and the other to execute the sorting logic in conjunction with the binary search mechanism(sort()). After completing my implementation for BinaryInsertionSort, both the truly random and almost sorted data of size 16000 saw immense improvements: roughly ~4 seconds runtimes on truly random and almost sorted data of size 16000 with InsertionSort to under 0.4 seconds with BinaryInsertionSort. BinaryInsertionSort roughly improved runtime from InsertionSort by around 90%. (Connor)

BinaryInsertionSort Natural Language PseudoCode:

Input: truly random generated array or almost sorted array of numbers *Output*: array in ascending order

```
1. (sort()) For each element (starting from the second element) in the array:
        1.a Set "current" to the element at the current index of the loop
        1.b Set "j" to a binary_search() call to find the correct position to insert "curr
                1.bi (nested binary_search()) While the start index "start" is less than th
                        1.bi(a) Calculate the midpoint index "mid" by finding the halfway p
                        1.bi(b) If the value of the midpoint "mid" is less than the target
                                1.bi(bi) Set the start index "start" to the midpoint plus 1
                        1.bi(c) Else:
                                1.bi(ci) Set the end index "end" to the midpoint index "mid
                1.bii Return the start index "start" as the position for which the "value"
        1.c Shift elements from "data" index "i - 1" to "j + 1" by one position to make roo
        1.d Place the "current" element at index "j" of "data"
2. Return the sorted array "data"
```

- *Input for binary_search()*: sorted array “data”, value to be searched for “value”(“current” in sort()), start index of array “start”, and end idex of array “end”
- *Output for binary_search()*: index where target value should be inserted

BinaryInsertionSort PsuedoCode:

```
class BinaryInsertionSort(CustomSort1):
    def __init__(self,):
        self.time = 0

    def binary_search(self, data to be sorted, target value for insertion, start index, end ind
        while start index < end index:
            midpoint index = (start index + end index) // 2
            if data to be sorted[midpoint index] < target value:
                start index = midpoint + 1
            else:
                end index = midpoint index
        return start index

    def sort(self, data to be sorted):
        for index i from 1 to length(data) - 1:
            current value = data to be sorted[ index i]
            index j = binary_search(data to be sorted, current value, 0, index i)
```

```
            data to be sorted[index j + 1: index i + 1] = data to be sorted[index j:index i]
            data to be sorted[index j] = current value
        return data sorted
```

Let's take a look at the runtime improvements from InsertionSort to BinaryInsertionSort.

[ ]:
```python
bis_df = tr_df.loc[tr_df['Algo'] == 'Binary Insertion', ['Data Size', 'Observed Runtime']].copy()
bis_df.rename(columns={'Observed Runtime': 'BIS Runtime'}, inplace=True)

insertion_df = tr_df.loc[tr_df['Algo'] == 'Insertion', ['Data Size', 'Observed Runtime']].copy()
insertion_df.rename(columns={'Observed Runtime': 'Insertion Runtime'}, inplace=True)

comparison_df = pd.merge(bis_df, insertion_df, on='Data Size')
comparison_df['Runtime Ratio (BIS / Insertion)'] = comparison_df['BIS Runtime'] / comparison_df['Insertion Runtime']
comparison_df.set_index('Data Size', inplace=True)

print("Comparison of BinaryInsertionSort to InsertionSort runtime on True Random data:")
print(comparison_df)
```

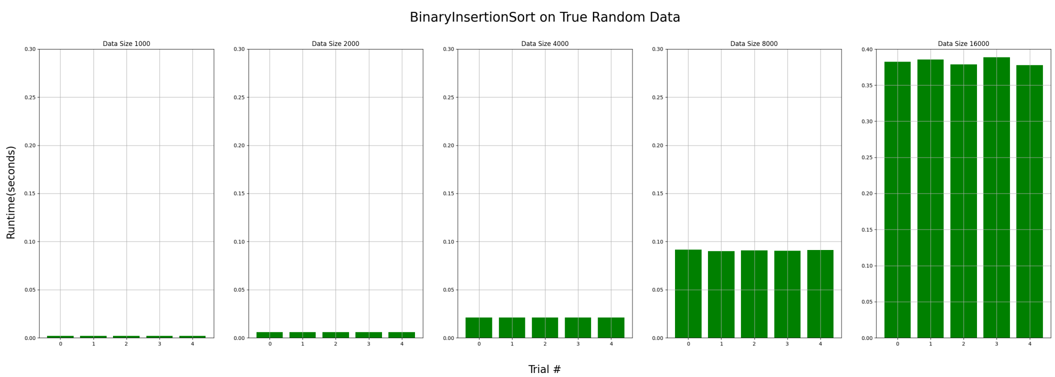
```
Comparison of BinaryInsertionSort to InsertionSort runtime on True Random data:
           BIS Runtime  Insertion Runtime  Runtime Ratio (BIS / Insertion)
Data Size
1000          0.001888           0.014235                         0.132614
2000          0.005836           0.059809                         0.097575
4000          0.021105           0.237579                         0.088835
8000          0.090807           0.959732                         0.094617
16000         0.382527           3.863937                         0.098999
```

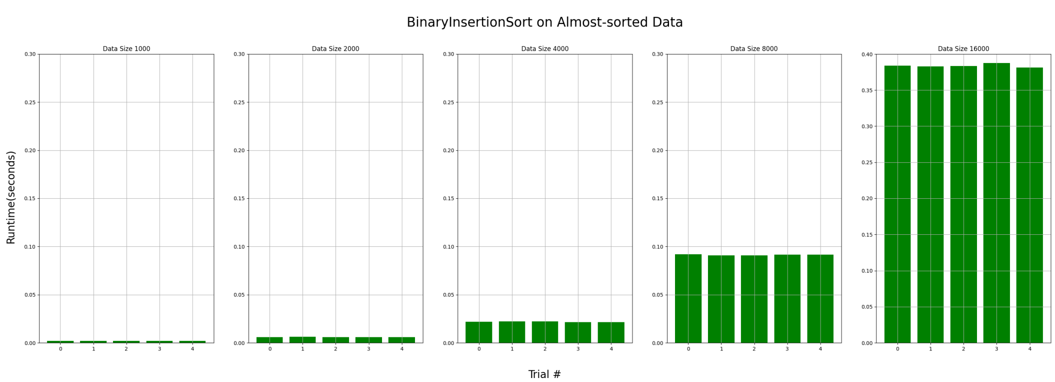
[ ]:
```python
bis_df = as_df.loc[as_df['Algo'] == 'Binary Insertion', ['Data Size', 'Observed Runtime']].copy()
bis_df.rename(columns={'Observed Runtime': 'BIS Runtime'}, inplace=True)

insertion_df = as_df.loc[as_df['Algo'] == 'Insertion', ['Data Size', 'Observed Runtime']].copy()
insertion_df.rename(columns={'Observed Runtime': 'Insertion Runtime'}, inplace=True)

comparison_df = pd.merge(bis_df, insertion_df, on='Data Size')
comparison_df['Runtime Ratio (BIS / Insertion)'] = comparison_df['BIS Runtime'] / comparison_df['Insertion Runtime']
comparison_df.set_index('Data Size', inplace=True)
```

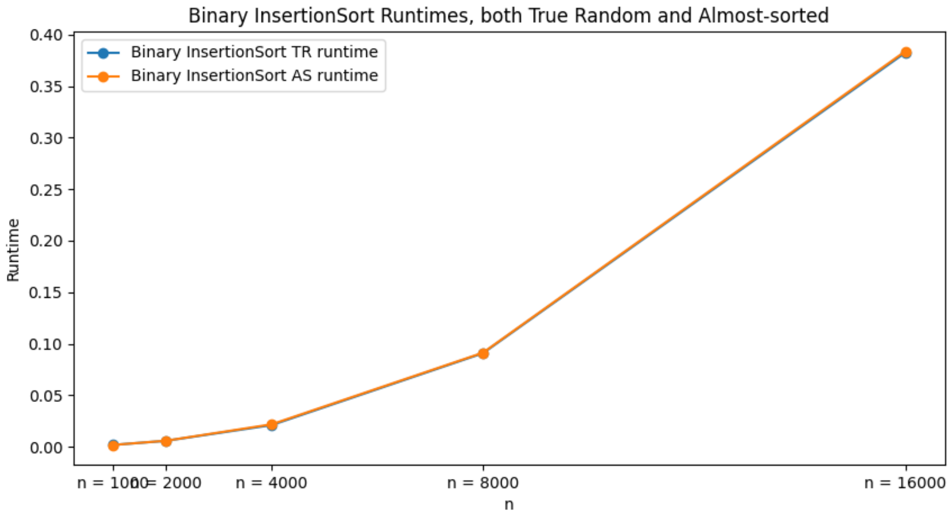
```python
print("Comparison of BinaryInsertionSort to InsertionSort runtime on True␣
↪Random data:")
print(comparison_df)
```

```
Comparison of BinaryInsertionSort to InsertionSort runtime on True Random data:
           BIS Runtime  Insertion Runtime  Runtime Ratio (BIS / Insertion)
Data Size
1000          0.001852           0.014492                         0.127823
2000          0.005927           0.062298                         0.095140
4000          0.021826           0.247955                         0.088024
8000          0.091293           1.004275                         0.090904
16000         0.383837           4.016604                         0.095562
```

As we can see, these results clearly illustrate the substantial runtime improvements achieved by BinaryInsertionSort. Across both true random and almost sorted inputs, BinaryInsertionSort consistently demonstrated lower mean runtimes compared to InsertionSort. The above two tables show that as the size of the input data increases, the runtime ratio of BinaryInsertionSort to InsertionSort remains relatively stable, ranging from 0.09 to 0.13. These ratios reflect that BinaryInsertionSort improved run times by 88-92%. This illustrates how the combination of insertion sort and binary search is more efficient in terms of runtime than InsertionSort alone(regardless of the data size). By halving the search space with each comparison, it reduced the total number of comparisons needed to find the insertion index, thus leading to faster runtimes.







0.2.8 Simplified Timsort Time Analysis

Timsort was an appealing discovery during my research into iterative improvements upon these sorting algorithms, as Timsort's most robust and feature-complete version is actually used at the core of Python's built-in sort() and sorted() functions. I sought to duplicate at least some of its functionality - in particular, its utilization of building 'runs' with insertion sort, that are then brought together with mergesort. This 'run' component is the only aspect of its robustness I sought to integrate for performance gains in our relatively straightforward use case.

1 Timsort Pseudocode

Class Timsort: Initialize with some minimum length of each 'run': Set MIN_RUN = 32

```
'sort' method, taking parameter 'data':
    Call recursive timsort_basic method, passing 'data'
    Return sorted 'data' upon completion of recursive sort

'timsort_basic' method with parameter 'data':
    Set 'n' to the length of 'data'
    Create runs of at least MIN_RUN size using 'insertion_sort'

    Initialize 'size' to MIN_RUN
    While 'size' is less than 'n' (merge the array, iteratively doubling the size of chunks to
        For each 'left' starting from 0, stepping by '2 * size':
            Calculate midpoint 'mid' as minimum of 'n - 1' and 'left + size - 1'
            Calculate 'right' as minimum of '(left + 2 * size - 1)' and '(n - 1)'
            If 'mid' is less than 'right', merge the current sections
        Double the 'size'

'insertion_sort' method with parameters 'data', 'left', 'right':
    For each position 'i' in range from 'left + 1' to 'right':
```

```
        Set 'key' to the value of 'data' at index 'i'
        Initialize 'j' to 'i - 1'
        While 'j' is greater than or equal to 'left' and 'data[j]' is greater than 'key':
            Move 'data[j]' one position to the right
            Decrease 'j' by 1
        Place 'key' in the correct sorted position

'merge' method with parameters 'data', 'left', 'mid', 'right':
    Initialize an empty list 'temp'
    Set 'i' to 'left' and 'j' to 'mid + 1'
    While either 'i' is less than or equal to 'mid' or 'j' is less than or equal to 'right':
        Compare elements from both halves and append the smaller one to 'temp'
        Increment 'i' or 'j' accordingly
    Append any remaining elements from either half to 'temp'
    Copy 'temp' back into 'data' starting from index 'left'
```

Below, let's look at how this simplified timsort improves upon mergesort performance.

[ ]:
```python
simple_tim_df = tr_df.loc[tr_df['Algo'] == 'Simple Tim', ['Data Size',␣
 ↪'Observed Runtime']].copy()
simple_tim_df.rename(columns={'Observed Runtime': 'Simple Tim Runtime'},␣
 ↪inplace=True)

merge_df = tr_df.loc[tr_df['Algo'] == 'Merge', ['Data Size', 'Observed␣
 ↪Runtime']].copy()
merge_df.rename(columns={'Observed Runtime': 'Merge Runtime'}, inplace=True)

comparison_df = pd.merge(simple_tim_df, merge_df, on='Data Size')

comparison_df['Runtime Ratio (Simple Tim / Merge)'] = comparison_df['Simple Tim␣
 ↪Runtime'] / comparison_df['Merge Runtime']

comparison_df.set_index('Data Size', inplace=True)

print("Comparison of Simple Timsort to MergeSort runtime on True Random data:")
print(comparison_df)
```

```
Comparison of Simple Timsort to MergeSort runtime on True Random data:
           Simple Tim Runtime  Merge Runtime  \
Data Size
1000                 0.001106       0.001358
2000                 0.002461       0.002968
4000                 0.005390       0.006353
8000                 0.011606       0.013436
16000                0.025035       0.028682

           Runtime Ratio (Simple Tim / Merge)
Data Size
```

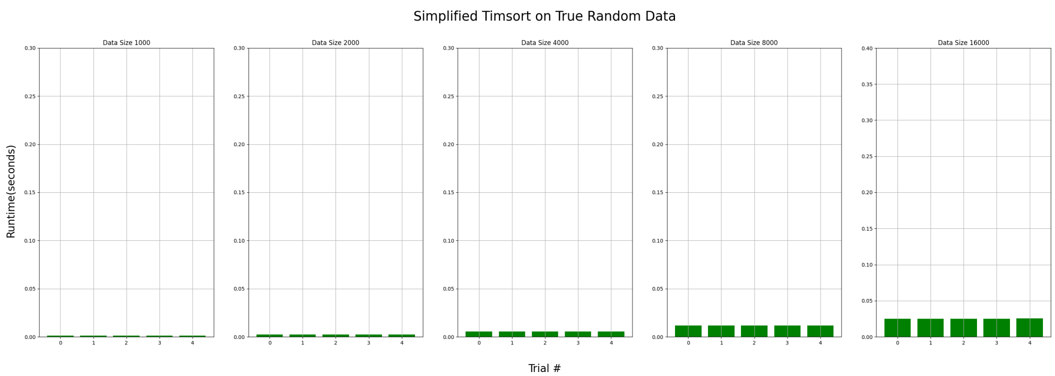
```
1000                                    0.814431
2000                                    0.829260
4000                                    0.848390
8000                                    0.863846
16000                                   0.872820
```

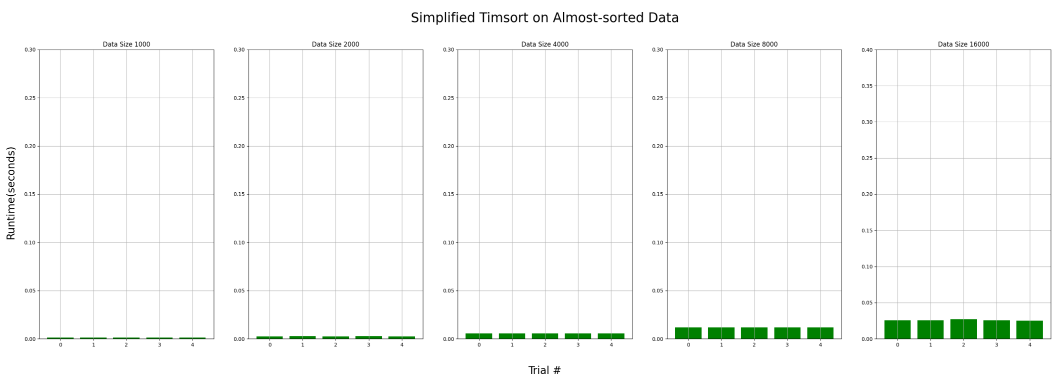
[ ]:
```
simple_tim_df = as_df.loc[as_df['Algo'] == 'Simple Tim', ['Data Size',␣
 ↪'Observed Runtime']].copy()
simple_tim_df.rename(columns={'Observed Runtime': 'Simple Tim Runtime'},␣
 ↪inplace=True)

merge_df = as_df.loc[as_df['Algo'] == 'Merge', ['Data Size', 'Observed␣
 ↪Runtime']].copy()
merge_df.rename(columns={'Observed Runtime': 'Merge Runtime'}, inplace=True)

comparison_df = pd.merge(simple_tim_df, merge_df, on='Data Size')

comparison_df['Runtime Ratio (Simple Tim / Merge)'] = comparison_df['Simple Tim␣
 ↪Runtime'] / comparison_df['Merge Runtime']

comparison_df.set_index('Data Size', inplace=True)

print("Comparison of Simple Timsort to MergeSort runtime on Almost-sorted data:
 ↪")
print(comparison_df)
```

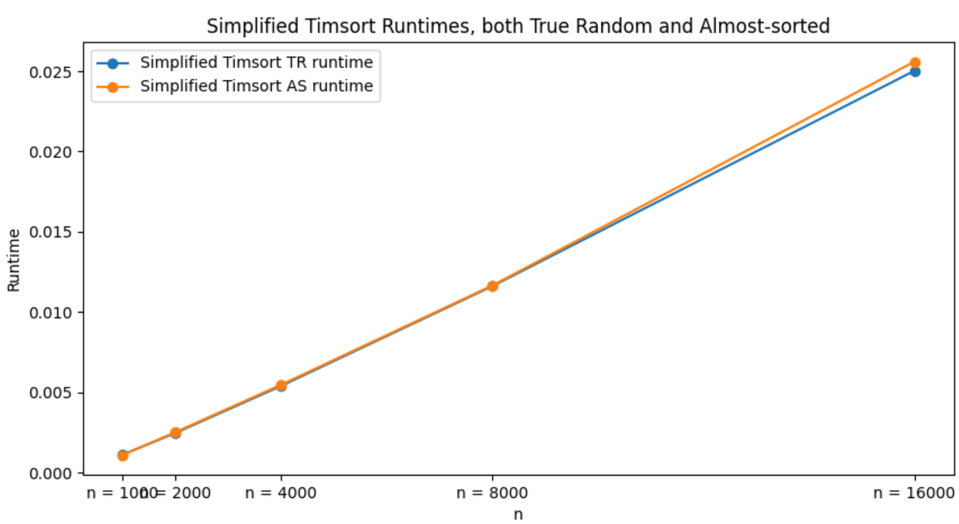
```
Comparison of Simple Timsort to MergeSort runtime on Almost-sorted data:
           Simple Tim Runtime  Merge Runtime  \
Data Size
1000                 0.001088       0.001372
2000                 0.002509       0.003013
4000                 0.005462       0.006403
8000                 0.011643       0.013497
16000                0.025603       0.028850

           Runtime Ratio (Simple Tim / Merge)
Data Size
1000                                 0.793026
2000                                 0.832756
4000                                 0.853102
8000                                 0.862636
16000                                0.887475
```

2 Simple Timsort Time Analysis

We can see that this simple implementation of Timsort provides a modest runtime improvement over MergeSort at the data sizes under consideration. While the performance delta is shrinking as n grows (from approximately a 20% improvement at n = 1000, to a 12% improvement at n = 16,000), this could be potentially be mitigated by adjusting Simple Timsort's starting size of calculated runs, perhaps seeding it as a log-base-two value that scales depending on n. We also see that Timsort is one of the algorithms that suffers a performance hit when working with almost-sorted data, likely derived from the fact that it uses insertion sort as one of its internal mechanisms.







2.1 Comparative Time Analysis

For our comparative time analysis, let's bring in some code and import results.

3 Ranking Table, per data size: True Random permutations

[ ]:
```
data_sizes = tr_df['Data Size'].unique()

# Prepare an empty dict to hold the algorithms and their runtimes for each data␣
↪size
rankings_with_runtime = {}

for size in data_sizes:
    # Filter rows matching current 'Data Size'
    filtered_df = tr_df[tr_df['Data Size'] == size]
    filtered_df = filtered_df.sort_values(by='Observed Runtime')

    # Combine 'Algo' and 'Observed Runtime' into a single string for each row
    combined_info = filtered_df.apply(lambda x: "{} ({:.6f}s)".
↪format(x['Algo'], x['Observed Runtime']), axis=1).values

    sorted_by_runtime = filtered_df.sort_values(by='Observed␣
↪Runtime')['Observed Runtime'].values
    sorted_combined_info = [info for _,info in sorted(zip(sorted_by_runtime,␣
↪combined_info))]

    rankings_with_runtime[size] = sorted_combined_info

max_length = max(len(v) for v in rankings_with_runtime.values())

for size in rankings_with_runtime:
    rankings_with_runtime[size] = list(rankings_with_runtime[size]) + [None] *␣
↪(max_length - len(rankings_with_runtime[size]))

tr_ranked_with_runtime_df = pd.DataFrame(rankings_with_runtime)

tr_ranked_with_runtime_df.index += 1  # Ranking starts from 1

print("True Random execution time rankings, per data size.")
print(tr_ranked_with_runtime_df)
```

```
True Random execution time rankings, per data size.
                     1000                      2000  \
1          Bucket (0.000166s)         Bucket (0.000308s)
2           Radix (0.000544s)          Radix (0.001131s)
3           Quick (0.000983s)          Quick (0.002183s)
4      Simple Tim (0.001106s)     Simple Tim (0.002461s)
```

```
5             Merge (0.001358s)             Merge (0.002968s)
6  Binary Insertion (0.001888s)  Binary Insertion (0.005836s)
7          Shell1000 (0.003393s)         Shell1000 (0.010192s)
8           Shell731 (0.004920s)          Shell731 (0.018754s)
9          Insertion (0.014235s)         Insertion (0.059809s)

                            4000                          8000  \
1            Bucket (0.001994s)            Bucket (0.000865s)
2             Radix (0.002226s)             Radix (0.004393s)
3             Quick (0.004996s)        Simple Tim (0.011606s)
4        Simple Tim (0.005390s)             Quick (0.012189s)
5             Merge (0.006353s)             Merge (0.013436s)
6  Binary Insertion (0.021105s)         Shell1000 (0.078536s)
7         Shell1000 (0.028034s)  Binary Insertion (0.090807s)
8          Shell731 (0.072155s)          Shell731 (0.280652s)
9         Insertion (0.237579s)         Insertion (0.959732s)

                            16000
1            Bucket (0.001611s)
2             Radix (0.008782s)
3        Simple Tim (0.025035s)
4             Merge (0.028682s)
5             Quick (0.031078s)
6         Shell1000 (0.220780s)
7  Binary Insertion (0.382527s)
8          Shell731 (1.113573s)
9         Insertion (3.863937s)
```

4 Ranking Table, per data size: Almost-sorted permutations

[ ]:
```
data_sizes = as_df['Data Size'].unique()

# Prepare an empty dict to hold the algorithms and their runtimes for each data␣
↪size
rankings_with_runtime = {}

for size in data_sizes:
    # Filter rows matching current 'Data Size'
    filtered_df = as_df[as_df['Data Size'] == size]
    filtered_df = filtered_df.sort_values(by='Observed Runtime')

    # Combine 'Algo' and 'Observed Runtime' into a single string for each row
    combined_info = filtered_df.apply(lambda x: "{} ({:.6f}s)".
↪format(x['Algo'], x['Observed Runtime']), axis=1).values
```

```
    sorted_by_runtime = filtered_df.sort_values(by='Observed␣
↪Runtime')['Observed Runtime'].values
    sorted_combined_info = [info for _,info in sorted(zip(sorted_by_runtime,␣
↪combined_info))]

    rankings_with_runtime[size] = sorted_combined_info

max_length = max(len(v) for v in rankings_with_runtime.values())

for size in rankings_with_runtime:
    rankings_with_runtime[size] = list(rankings_with_runtime[size]) + [None] *␣
↪(max_length - len(rankings_with_runtime[size]))

as_ranked_with_runtime_df = pd.DataFrame(rankings_with_runtime)

as_ranked_with_runtime_df.index += 1  # Ranking starts from 1
print("Almost-sorted execution time rankings, per data size.")
print(as_ranked_with_runtime_df)
```

```
Almost-sorted execution time rankings, per data size.
                          1000                         2000   \
1             Bucket (0.000180s)             Bucket (0.000323s)
2              Radix (0.000535s)              Radix (0.001149s)
3              Quick (0.000985s)              Quick (0.002114s)
4         Simple Tim (0.001088s)         Simple Tim (0.002509s)
5              Merge (0.001372s)              Merge (0.003013s)
6   Binary Insertion (0.001852s)   Binary Insertion (0.005927s)
7           Shell1000 (0.003496s)          Shell1000 (0.010301s)
8            Shell731 (0.004961s)           Shell731 (0.019170s)
9           Insertion (0.014492s)          Insertion (0.062298s)

                          4000                         8000   \
1             Bucket (0.000517s)             Bucket (0.000871s)
2              Radix (0.002219s)              Radix (0.004375s)
3              Quick (0.004996s)         Simple Tim (0.011643s)
4         Simple Tim (0.005462s)              Quick (0.012218s)
5              Merge (0.006403s)              Merge (0.013497s)
6   Binary Insertion (0.021826s)          Shell1000 (0.079246s)
7           Shell1000 (0.028260s)   Binary Insertion (0.091293s)
8            Shell731 (0.073197s)           Shell731 (0.286670s)
9           Insertion (0.247955s)          Insertion (1.004275s)

                           16000
1             Bucket (0.001638s)
2              Radix (0.008765s)
3         Simple Tim (0.025603s)
4              Merge (0.028850s)
```

```
5             Quick (0.030703s)
6         Shell1000 (0.219721s)
7  Binary Insertion (0.383837s)
8          Shell731 (1.125490s)
9         Insertion (4.016604s)
```

5 Observations regarding rankings, patterns, performance as n changes.

- A few things across the rankings are constant:
  - Bucket and Radix hold the #1 and #2 spot consistently across all data sizes and across both permutation styles. Very fast.
  - Conversely, Shell (7-3-1) and Insertion sort occupy the bottom of the field - #8 and #9 - across all data sizes and permutation styles
  - Insertion's lack of speed is demonstrating itself dramatically as n increases.
- Other notes:
  - Quicksort begins faster than Simple Tim and Mergesort at n = 1000, but by n = 16,000 both of the latter are running faster.
  - Simple Tim seems to cope the best with growing datasize, even in its primitive implementation, compared to rote Quick and Mergesort.
  - Similarly, as data size grows, Shellsort (1000 - 100 - 10 - 1) steals Binary Insertion's #6 rank. As n increases, there seems to be some risk of Binary Insertion dramatically increasing in execution speed - sensible, as an O(n^2) algorithm.

6 True Random permutation comparison tables between algorithms: Observed runtime, Empirical Big-O, Theoretical Big-O.

[ ]:
```python
# Get unique 'Data Size' values
data_sizes = tr_df['Data Size'].unique()

# Dictionary to store DataFrames
dfs_by_data_size = {}

# Select only the required columns
columns_needed = ['Algo', 'Observed Runtime', 'Emp Big-O', 'Theoretical Big-O']

for size in data_sizes:
    # Filter tr_df for the current 'Data Size' and select only the required␣
↪columns
    df_filtered = tr_df[tr_df['Data Size'] == size][columns_needed].copy()

    # Add the filtered DataFrame to the dictionary, using 'Data Size' as the key
    dfs_by_data_size[size] = df_filtered

for data_sizes in dfs_by_data_size:
    print(f"True Random runtimes at Data Size {data_sizes}:")
```

```python
print(dfs_by_data_size[data_sizes])
```

```
True Random runtimes at Data Size 1000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
0              Merge          0.001358        NaN           n log n
5              Quick          0.000983        NaN               n^2
10         Insertion          0.014235        NaN               n^2
15          Shell731          0.004920        NaN               n^2
20         Shell1000          0.003393        NaN               n^2
25            Bucket          0.000166        NaN                 n
30             Radix          0.000544        NaN                nd
35  Binary Insertion          0.001888        NaN               n^2
40        Simple Tim          0.001106        NaN           n log n
True Random runtimes at Data Size 2000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
1              Merge          0.002968   1.127819           n log n
6              Quick          0.002183   1.150906               n^2
11         Insertion          0.059809   2.070882               n^2
16          Shell731          0.018754   1.930490               n^2
21         Shell1000          0.010192   1.586662               n^2
26            Bucket          0.000308   0.891417                 n
31             Radix          0.001131   1.056270                nd
36  Binary Insertion          0.005836   1.628233               n^2
41        Simple Tim          0.002461   1.153852           n log n
True Random runtimes at Data Size 4000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
2              Merge          0.006353   1.098149           n log n
7              Quick          0.004996   1.194186               n^2
12         Insertion          0.237579   1.989976               n^2
17          Shell731          0.072155   1.943929               n^2
22         Shell1000          0.028034   1.459771               n^2
27            Bucket          0.001994   2.695016                 n
32             Radix          0.002226   0.977382                nd
37  Binary Insertion          0.021105   1.854595               n^2
42        Simple Tim          0.005390   1.131051           n log n
True Random runtimes at Data Size 8000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
3              Merge          0.013436   1.080480           n log n
8              Quick          0.012189   1.286686               n^2
13         Insertion          0.959732   2.014225               n^2
18          Shell731          0.280652   1.959615               n^2
23         Shell1000          0.078536   1.486165               n^2
28            Bucket          0.000865  -1.203880                 n
33             Radix          0.004393   0.980833                nd
38  Binary Insertion          0.090807   2.105184               n^2
43        Simple Tim          0.011606   1.106526           n log n
True Random runtimes at Data Size 16000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
```

```
4              Merge          0.028682   1.094082          n log n
9              Quick          0.031078   1.350334              n^2
14         Insertion          3.863937   2.009369              n^2
19          Shell731          1.113573   1.988339              n^2
24         Shell1000          0.220780   1.491172              n^2
29            Bucket          0.001611   0.896506                n
34             Radix          0.008782   0.999246               nd
39  Binary Insertion          0.382527   2.074692              n^2
44        Simple Tim          0.025035   1.108993          n log n
```

7 Almost-sorted permutation comparison tables between algorithms: Observed runtime, Empirical Big-O, Theoretical Big-O.

```
[ ]: # Get unique 'Data Size' values
data_sizes = as_df['Data Size'].unique()

# Dictionary to store DataFrames
dfs_by_data_size = {}

# Select only the required columns
columns_needed = ['Algo', 'Observed Runtime', 'Emp Big-O', 'Theoretical Big-O']

for size in data_sizes:
    # Filter as_df for the current 'Data Size' and select only the required␣
 ↪columns
    df_filtered = as_df[as_df['Data Size'] == size][columns_needed].copy()

    # Add the filtered DataFrame to the dictionary, using 'Data Size' as the key
    dfs_by_data_size[size] = df_filtered

for data_sizes in dfs_by_data_size:
    print(f"Almost-sorted runtimes at Data Size {data_sizes}:")
    print(dfs_by_data_size[data_sizes])
```

```
Almost-sorted runtimes at Data Size 1000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
0              Merge          0.001372        NaN          n log n
5              Quick          0.000985        NaN              n^2
10         Insertion          0.014492        NaN              n^2
15          Shell731          0.004961        NaN              n^2
20         Shell1000          0.003496        NaN              n^2
25            Bucket          0.000180        NaN                n
30             Radix          0.000535        NaN               nd
35  Binary Insertion          0.001852        NaN              n^2
40        Simple Tim          0.001088        NaN          n log n
Almost-sorted runtimes at Data Size 2000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
```

```
1              Merge          0.003013   1.135242          n log n
6              Quick          0.002114   1.101992              n^2
11         Insertion          0.062298   2.103928              n^2
16          Shell731          0.019170   1.950063              n^2
21         Shell1000          0.010301   1.558917              n^2
26            Bucket          0.000323   0.844827                n
31             Radix          0.001149   1.101922               nd
36  Binary Insertion          0.005927   1.677908              n^2
41        Simple Tim          0.002509   1.205767          n log n
Almost-sorted runtimes at Data Size 4000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
2              Merge          0.006403   1.087564          n log n
7              Quick          0.004996   1.240900              n^2
12         Insertion          0.247955   1.992818              n^2
17          Shell731          0.073197   1.932952              n^2
22         Shell1000          0.028260   1.455995              n^2
27            Bucket          0.000517   0.680581                n
32             Radix          0.002219   0.950318               nd
37  Binary Insertion          0.021826   1.880668              n^2
42        Simple Tim          0.005462   1.122388          n log n
Almost-sorted runtimes at Data Size 8000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
3              Merge          0.013497   1.075869          n log n
8              Quick          0.012218   1.290173              n^2
13         Insertion          1.004275   2.018005              n^2
18          Shell731          0.286670   1.969527              n^2
23         Shell1000          0.079246   1.487551              n^2
28            Bucket          0.000871   0.752056                n
33             Radix          0.004375   0.979011               nd
38  Binary Insertion          0.091293   2.064447              n^2
43        Simple Tim          0.011643   1.091902          n log n
Almost-sorted runtimes at Data Size 16000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
4              Merge          0.028850   1.095911          n log n
9              Quick          0.030703   1.329332              n^2
14         Insertion          4.016604   1.999821              n^2
19          Shell731          1.125490   1.973093              n^2
24         Shell1000          0.219721   1.471270              n^2
29            Bucket          0.001638   0.911686                n
34             Radix          0.008765   1.002652               nd
39  Binary Insertion          0.383837   2.071922              n^2
44        Simple Tim          0.025603   1.136866          n log n
```

8 Common Big-O Functions for Each Algorithm, Based On Observed Empiricial Asymptotic Runtime Using Doubling Hypothesis

Note: For these assignments, we're using the doubling hypothesis factor guidelines provided on Edstem and our own judgement based on the trend of observed runtime ratio as data size changes for each algorithm.

- Merge: Ratio of approximately 1 through 1.1. Assigning O(log (n)).
- Quick: Ratio of approximately 1.15 through 1.3, growing as n increases. Assigning O(n).
- Insertion: Ratio of approximately 2. Assigning O(n log(n)).
- Shell (7-3-1): Ratio of approximately 1.95. Assigning O(n log(n)).
- Shell (1000-100-10-1): Ratio of approximately 1.58 to 1.46, decreasing. Assigning O(n).
- Bucket: Ratio of approximately 0.7 - 0.9. Assigning O(log(n)).
  - Note: There was an extreme result in our initial data states that resulted in a peculiar value for the third seed under the true random permutation case. As such, we have a negative ratio. Given Bucket's consistency across every other trial, we are making this assignment by analyzing those trials primarily. We found it amusing to strike such a strange result, and decided to keep it in instead of shuffling our seeding arrangement to sidestep it, given the algorithm reliably sorts.
- Radix: Ratio of approximately 0.95 - 1.1. Assigning O(log (n)).
- Binary Insertion: Ratio of approximately 1.65 at n = 1000, to 2.1 as n increases. Given this progressive delta, assigning O(n).
- Simplified Tim: Ratio of approximately 1.15 to 1.1, shrinking as n increases. Assigning O(log(n)).

9 Noted Differences Between Observed Runtime Versus Theoretical Big-O Runtime

For these comparisons, we're using Big-O time complexity for each algorithm that considers their worst case scenario.

- Merge: Assigned O(log(n)), worst case O(n log(n)). Based on the doubling hypothesis factor, in practice this was faster than linearithmic.
- Quick: Assigned O(n), given its ratio grew as data size increased. Worst case O(n^2). Again, this was much faster than its worst-case Big-O. This is also appreciably faster than its average case Big-O, O(n log(n)).
- Insertion: Assigned O(n log(n)). Reliably right around 2, dithering as data increased. Faster in practice than its worst-case O(n^2) with these data, but quite slow to begin with compared to the competition.
- Shell: We see appreciable differences in the gap assignment between the two Shell schemas provided. (7-3-1)'s ratio held near 2, and was assigned O(n log(n)), while (1000-100-10-1) steadily decreased, and was assigned O(n). A clear case for how the Shell gap schema and data size interact to determine sorting speed relative to Shell's worst-case, O(n^2)
- Bucket: So fast. Assigned O(log(n)). Steadily beneath 1, suggesting that it was getting relatively faster as the data size increased. Likely due to the fact that as n increased, the numer of possible buckets never changed - it was always 1000. Interesting, and clearly ahead

of its O(n) theoretical runtime in practice.

- Radix: Ratio around 1, dithering, assigned O(log(n)). Almost as fast as bucket; begs inquiry into what relationship between n-tuple wordsize or bucket count necessitates a switch from one to the other. Outperformed worst-case O(nd).
- Binary Insertion: Clear improvement from Insertion, but its ratios were slightly higher than Insertion as data size increased. This may suggest that in huge datasets, regular Insertion catches up. Assigned O(n), performing ahead of its O(n^2) worst-case.
- Timsort: Satisfying combination that takes advantage of the strengths of Insertion and Merge. Pulled ahead of everything non-Bucket/Radix at n = 16,000. Assigned O(log(n)), better than its worst case of O(nlog(n)).