final_report

May 29, 2024

0.1 DATA260P Project 1: Comparing Sorting Algorithms

Connor McManigal and Peyton Politewicz

[1]:
```
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)
```

[2]:
```
print(tr_df)
```

```
                Algo  Data Size  Observed Runtime     Ratio  Emp Big-O  \
0              Merge       1000          0.001916       NaN        NaN
1              Merge       2000          0.004056  2.117361   1.082267
2              Merge       4000          0.008482  2.091213   1.064340
3              Merge       8000          0.018181  2.143459   1.099941
4              Merge      16000          0.038370  2.110502   1.077586
5              Quick       1000          0.001324       NaN        NaN
6              Quick       2000          0.003008  2.272121   1.184039
7              Quick       4000          0.006822  2.267949   1.181388
8              Quick       8000          0.016247  2.381692   1.251987
9              Quick      16000          0.041129  2.531412   1.339942
10         Insertion       1000          0.018981       NaN        NaN
```

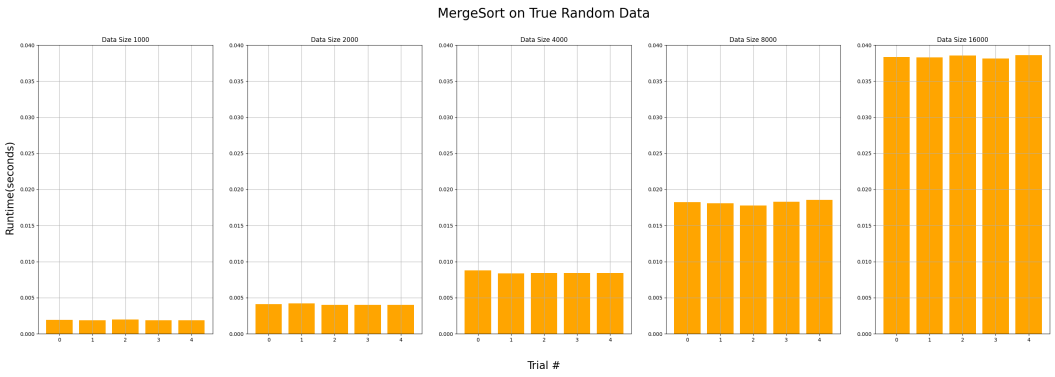
11	Insertion	2000	0.076569	4.033956	2.012195
12	Insertion	4000	0.299134	3.906733	1.965963
13	Insertion	8000	1.224779	4.094420	2.033659
14	Insertion	16000	4.817695	3.933522	1.975822
15	Shell731	1000	0.007089	NaN	NaN
16	Shell731	2000	0.026349	3.716697	1.894021
17	Shell731	4000	0.100089	3.798601	1.925468
18	Shell731	8000	0.386434	3.860910	1.948941
19	Shell731	16000	1.547525	4.004635	2.001671
20	Shell1000	1000	0.005101	NaN	NaN
21	Shell1000	2000	0.013417	2.630007	1.395067
22	Shell1000	4000	0.036741	2.738504	1.453388
23	Shell1000	8000	0.099846	2.717555	1.442309
24	Shell1000	16000	0.281685	2.821187	1.496302
25	Bucket	1000	0.000217	NaN	NaN
26	Bucket	2000	0.000405	1.866606	0.900418
27	Bucket	4000	0.000733	1.809985	0.855978
28	Bucket	8000	0.001267	1.727493	0.788679
29	Bucket	16000	0.002480	1.958157	0.969497
30	Radix	1000	0.001246	NaN	NaN
31	Radix	2000	0.002378	1.908704	0.932593
32	Radix	4000	0.004622	1.943660	0.958776
33	Radix	8000	0.009829	2.126646	1.088580
34	Radix	16000	0.019688	2.003031	1.002185
35	Binary Insertion	1000	0.001555	NaN	NaN
36	Binary Insertion	2000	0.004537	2.918067	1.545013
37	Binary Insertion	4000	0.016499	3.637011	1.862753
38	Binary Insertion	8000	0.066804	4.048897	2.017529
39	Binary Insertion	16000	0.284079	4.252426	2.088286
40	Simple Tim	1000	0.001462	NaN	NaN
41	Simple Tim	2000	0.003102	2.122537	1.085790
42	Simple Tim	4000	0.006723	2.167151	1.115800
43	Simple Tim	8000	0.016296	2.423932	1.277349
44	Simple Tim	16000	0.031620	1.940303	0.956282

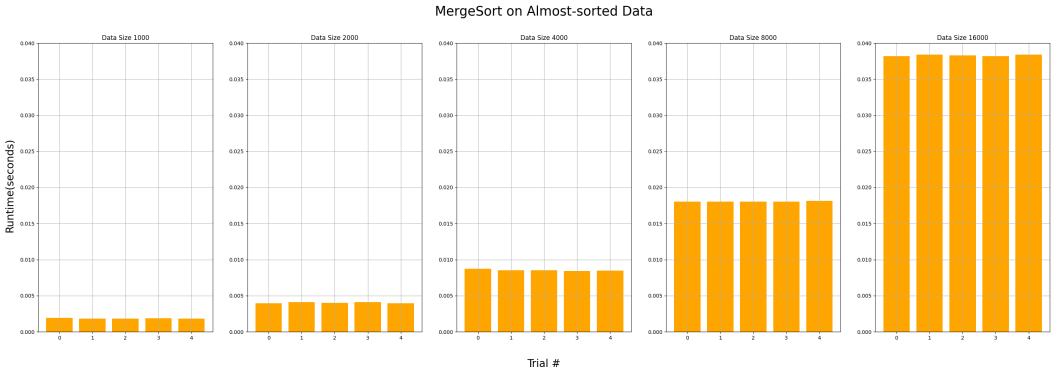
	Theoretical Big-O
0	n log n
1	n log n
2	n log n
3	n log n
4	n log n
5	n^2
6	n^2
7	n^2
8	n^2
9	n^2
10	n^2
11	n^2

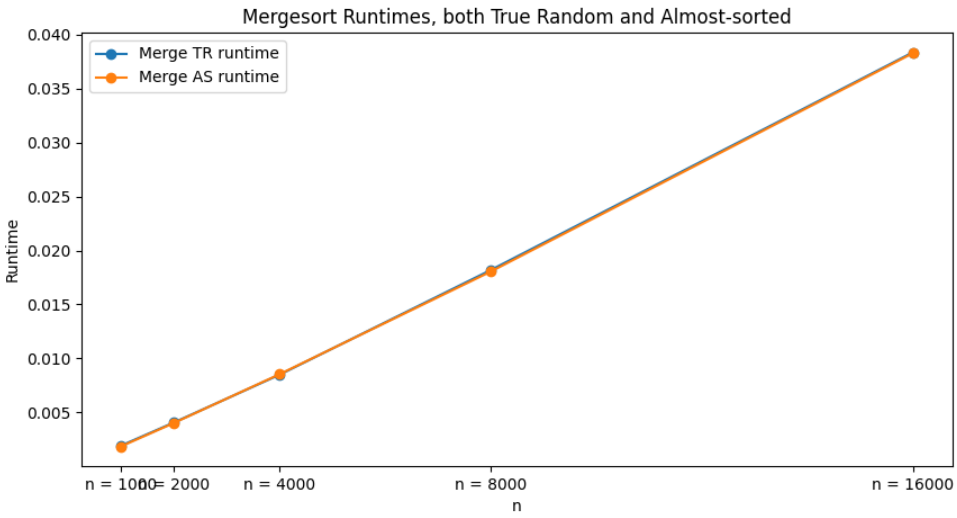
```
12              n^2
13              n^2
14              n^2
15              n^2
16              n^2
17              n^2
18              n^2
19              n^2
20              n^2
21              n^2
22              n^2
23              n^2
24              n^2
25                n
26                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

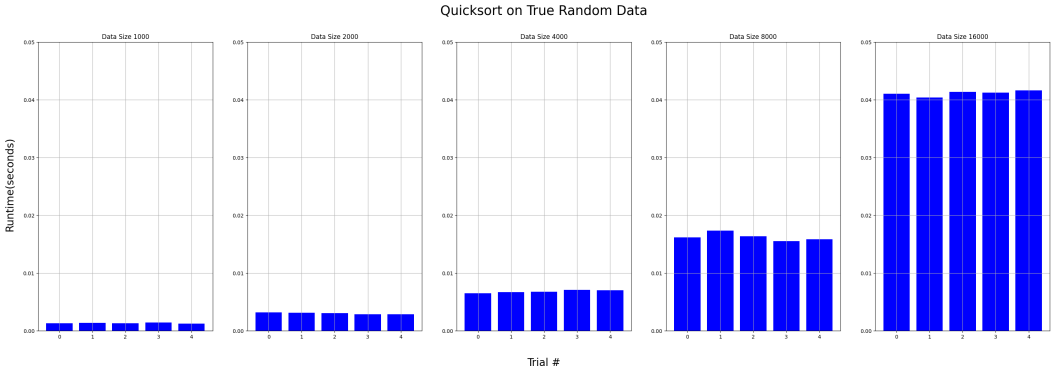
0.2.1 MergeSort Time Analysis

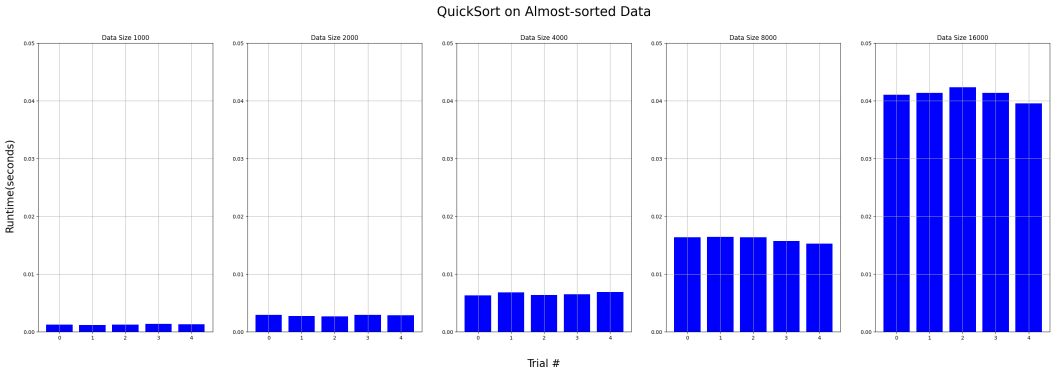


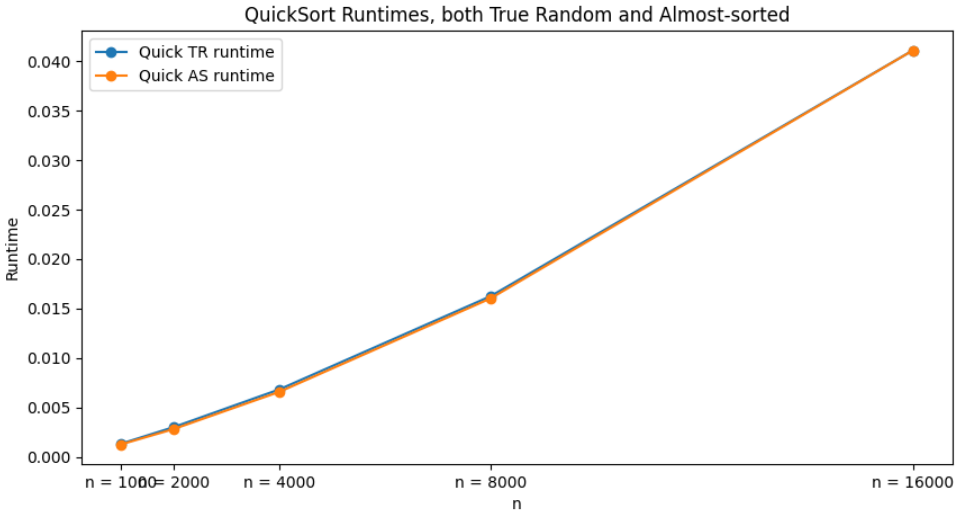




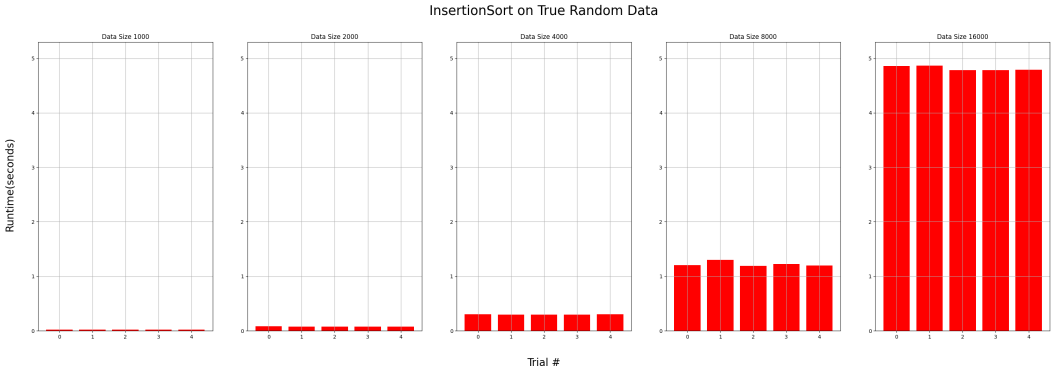
0.2.2 QuickSort Time Analysis

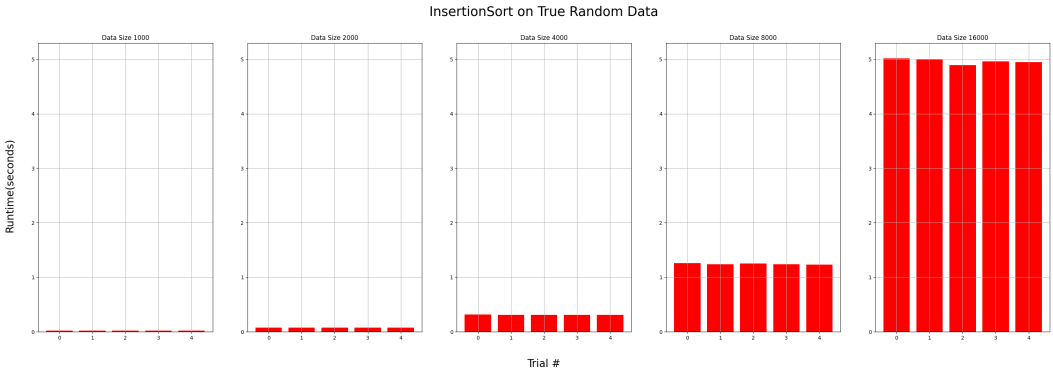


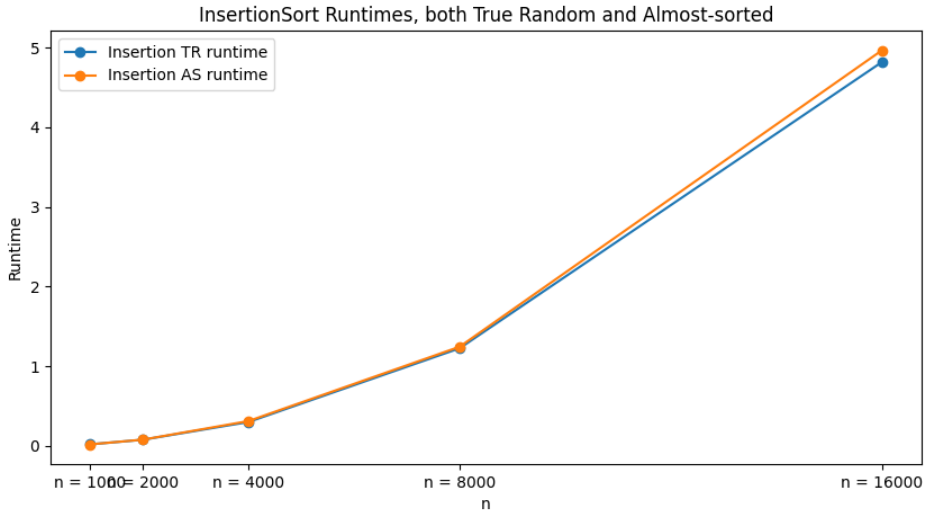




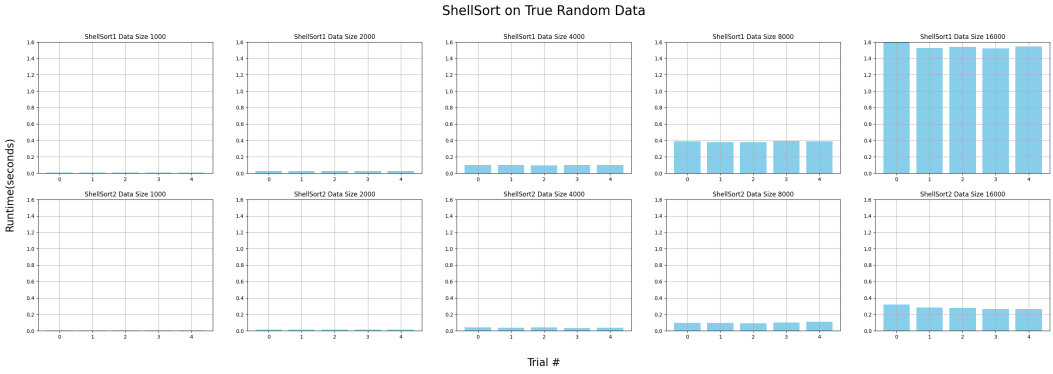
0.2.3 InsertionSort Time Analysis

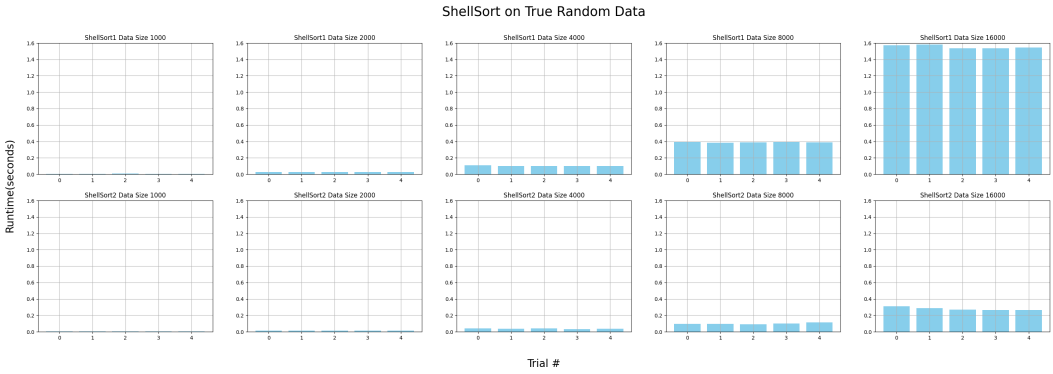


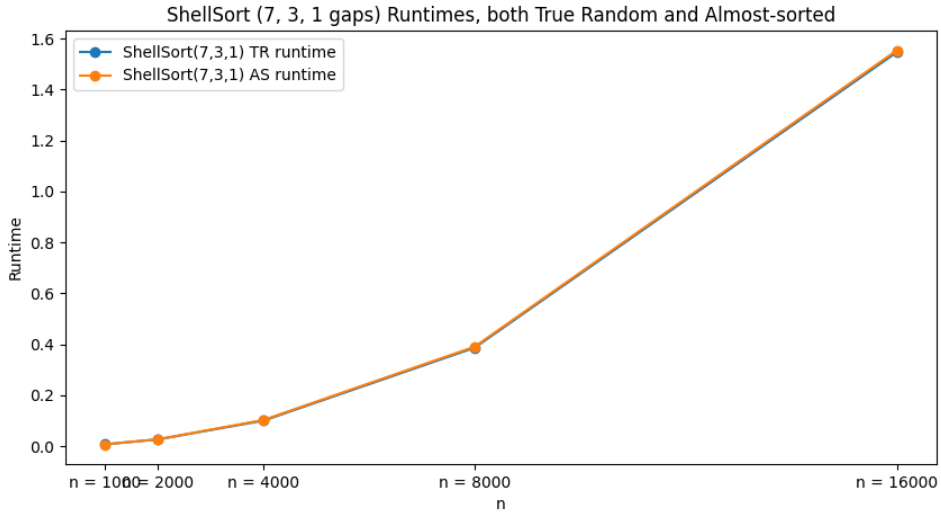


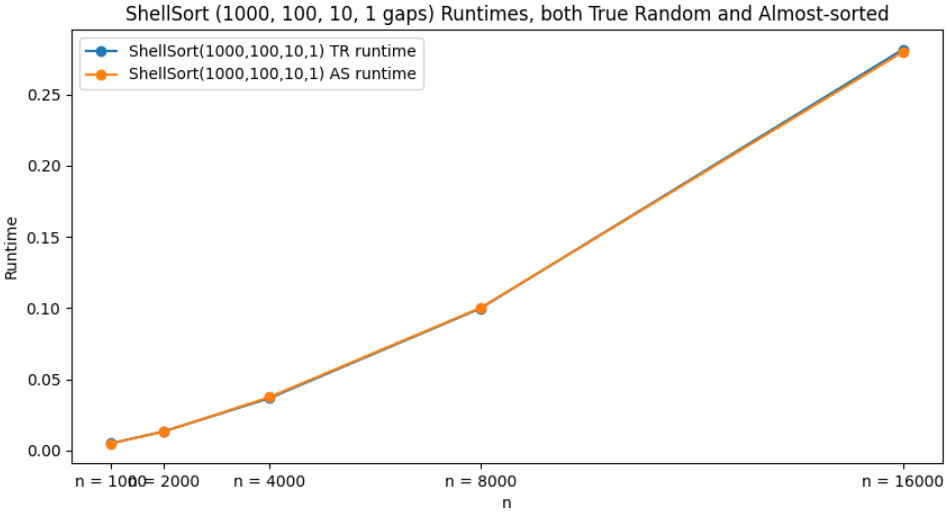


0.2.4 ShellSort Time Analysis

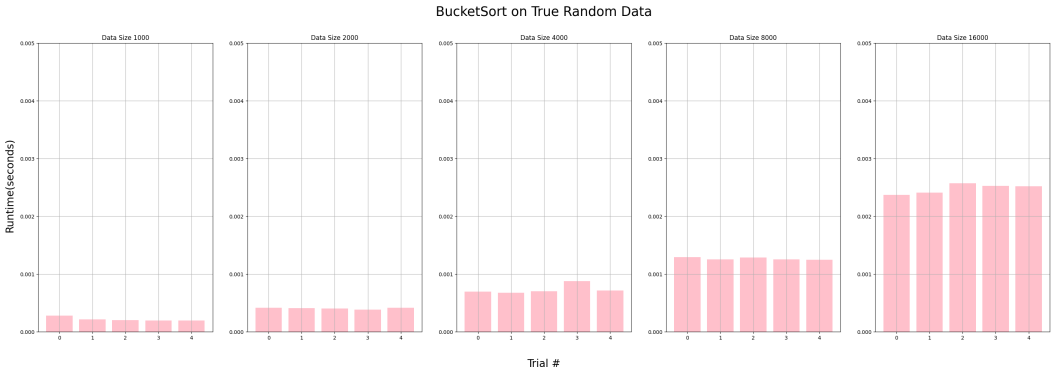


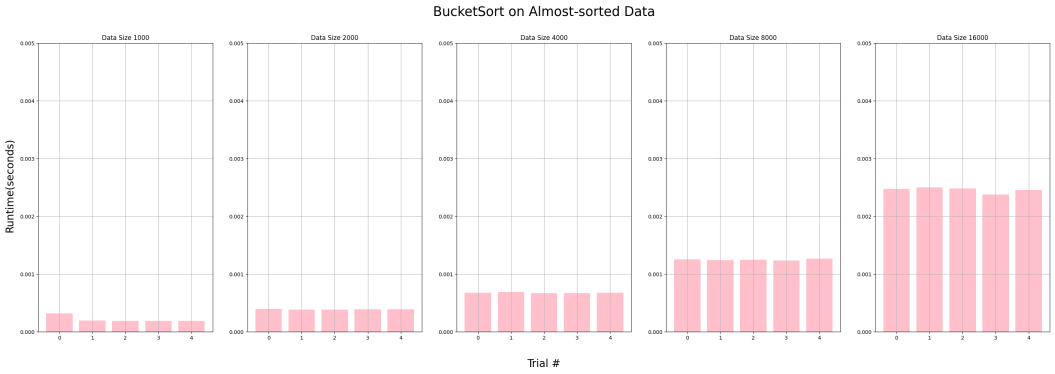

ShellSort on True Random Data
ShellSort1 Data Size 1000
ShellSort1 Data Size 2000
ShellSort1 Data Size 4000
ShellSort1 Data Size 8000
ShellSort1 Data Size 16000
ShellSort2 Data Size 1000
ShellSort2 Data Size 2000
ShellSort2 Data Size 4000
ShellSort2 Data Size 8000
ShellSort2 Data Size 16000
Runtime(seconds)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

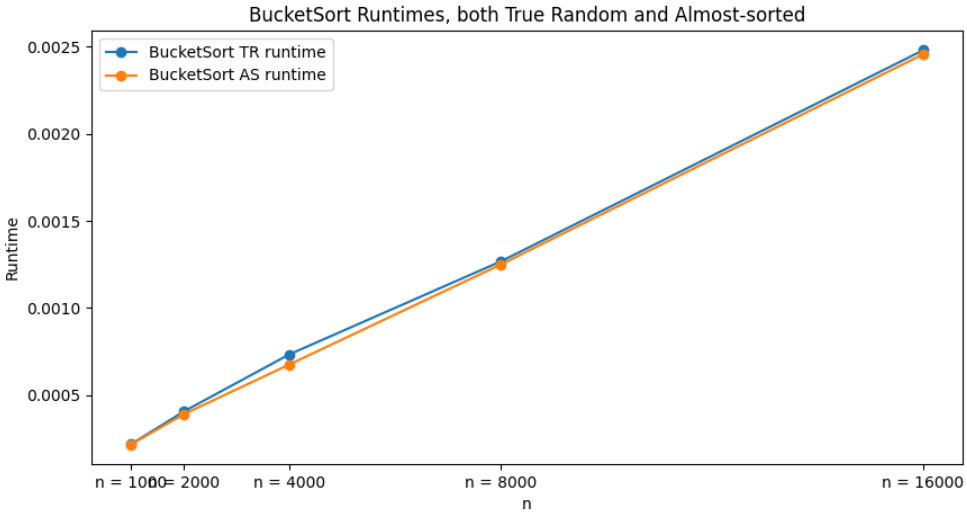

ShellSort (7, 3, 1 gaps) Runtimes, both True Random and Almost-sorted
ShellSort(7,3,1) TR runtime
ShellSort(7,3,1) AS runtime
Runtime
n = 1000
n = 2000
n = 4000
n = 8000
n = 16000
n


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
n = 1000
n = 2000
n = 4000
n = 8000
n = 16000
n

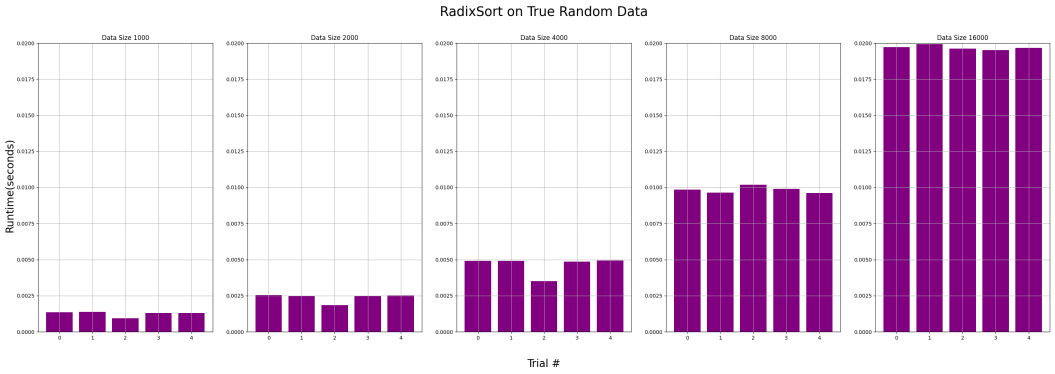
0.2.5 BucketSort Time Analysis

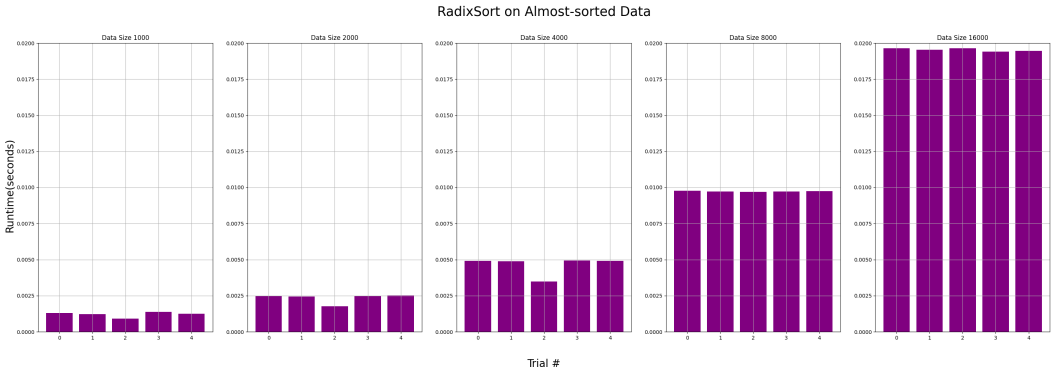


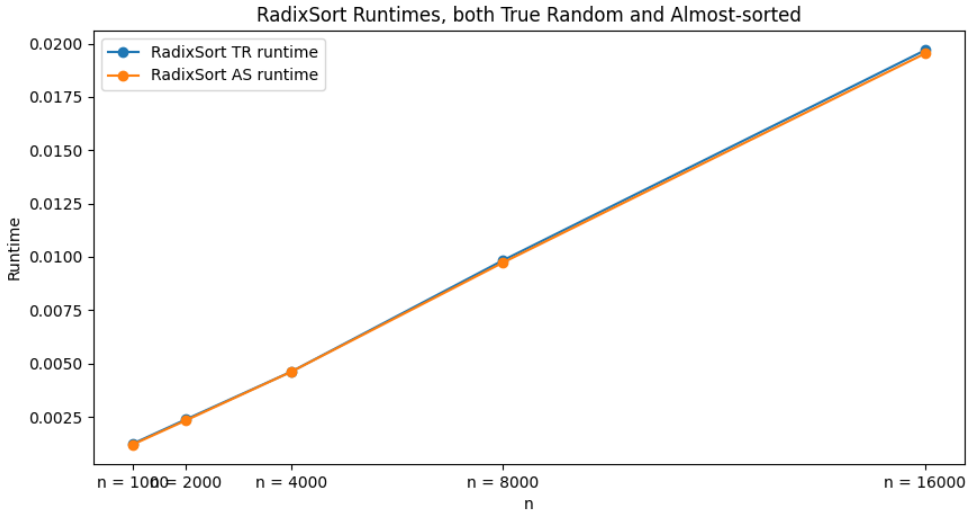




0.2.6 RadixSort Time Analysis







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 binary_search call to find the position to insert "current" into sorted suba
        1.bi (binary_search()) While the "start" index is less than the "end" index:
            1.bi(a) Calculate the "midpoint" index or halfway point of "start" and "end"
            1.bi(b) If the value of the "midpoint" is less than the target "value":
                1.bi(bi) Set the "start" index to the midpoint plus 1 "mid + 1"
            1.bi(c) Else:
                1.bi(ci) Set the "end" index to the "midpoint" index
        1.bii Return the "start" index as the spot for which the "value" should be inserted
    1.c Shift elements: "data" index "i - 1" to "j + 1" to make room for the "current" element
    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 index)
    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.

[3]:
```
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)
```

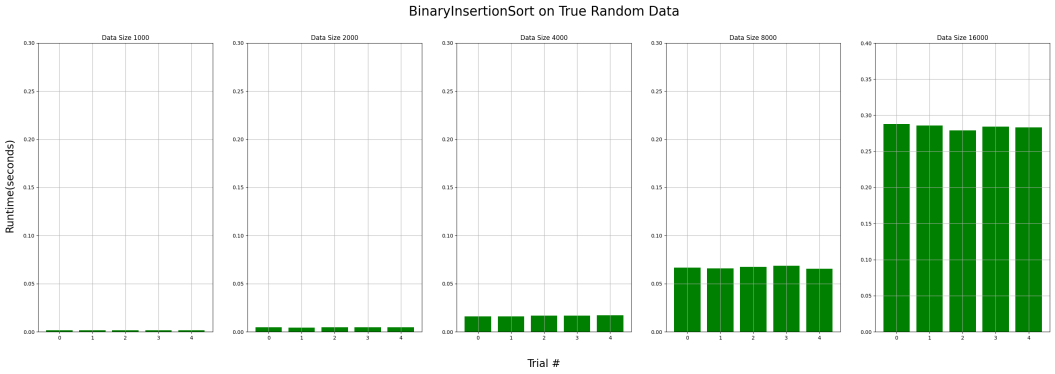
```
Comparison of BinaryInsertionSort to InsertionSort runtime on True Random data:
           BIS Runtime  Insertion Runtime  Runtime Ratio (BIS / Insertion)
Data Size
1000          0.001555           0.018981                         0.081904
2000          0.004537           0.076569                         0.059247
4000          0.016499           0.299134                         0.055157
8000          0.066804           1.224779                         0.054544
16000         0.284079           4.817695                         0.058966
```

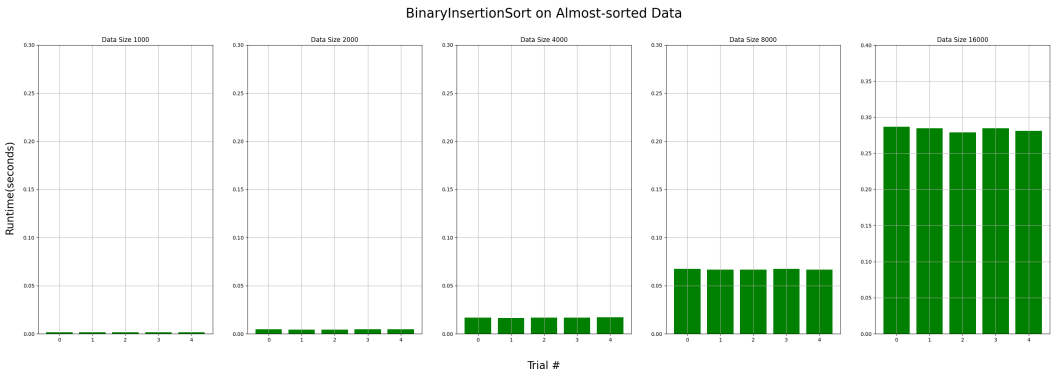
[4]:
```
bis_df = as_df.loc[as_df['Algo'] == 'Binary Insertion', ['Data Size', 'Observed␣
 ↪Runtime']].copy()
bis_df.rename(columns={'Observed Runtime': 'BIS Runtime'}, inplace=True)

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)
```

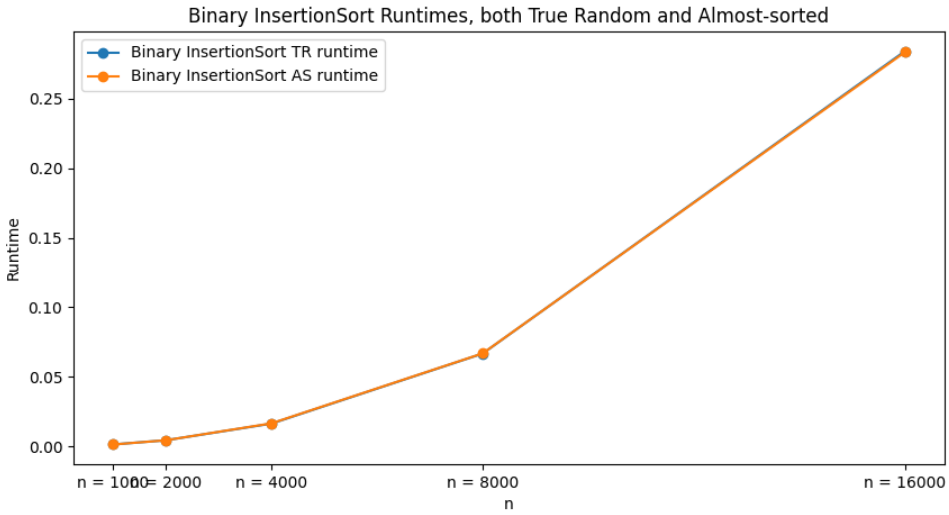
```
print("Comparison of BinaryInsertionSort to InsertionSort runtime on True␣
  ↪Random data:")
print(comparison_df)
```

```
Comparison of BinaryInsertionSort to InsertionSort runtime on True Random data:
           BIS Runtime  Insertion Runtime  Runtime Ratio (BIS / Insertion)
Data Size
1000          0.001490           0.018644                         0.079927
2000          0.004535           0.077796                         0.058288
4000          0.016692           0.311041                         0.053665
8000          0.066929           1.244236                         0.053791
16000         0.283342           4.964963                         0.057068
```

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.

[5]:
```
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                 0.001462       0.001916
2000                 0.003102       0.004056
4000                 0.006723       0.008482
8000                 0.016296       0.018181
16000                0.031620       0.038370

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

```
1000                                0.762998
2000                                0.764863
4000                                0.792637
8000                                0.896354
16000                               0.824069
```

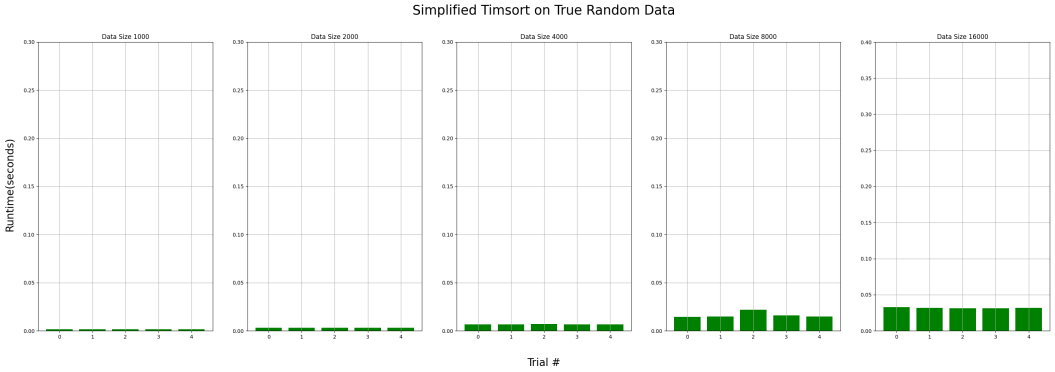
[6]:
```
simple_tim_df = as_df.loc[as_df['Algo'] == 'Simple Tim', ['Data Size',␣
 ↪'Observed Runtime']].copy()
simple_tim_df.rename(columns={'Observed Runtime': 'Simple Tim Runtime'},␣
 ↪inplace=True)

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)
```

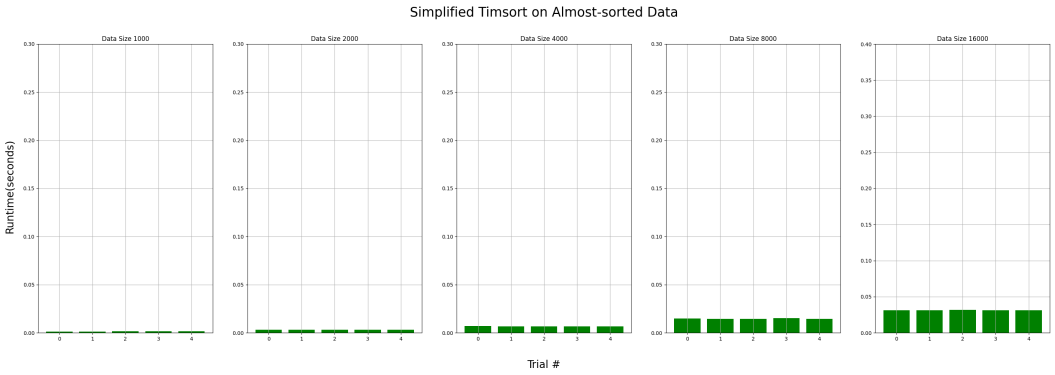
```
Comparison of Simple Timsort to MergeSort runtime on Almost-sorted data:
           Simple Tim Runtime  Merge Runtime  \
Data Size
1000                 0.001088       0.001372
2000                 0.002509       0.003013
4000                 0.005462       0.006403
8000                 0.011643 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

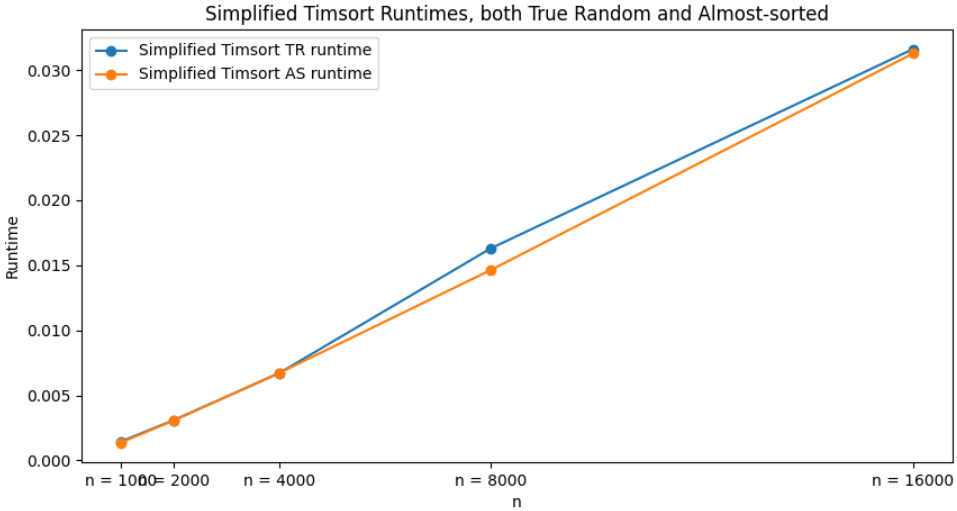
2 Simple Timsort Time Analysis

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

[6]:
```
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 (0.000217s)            Bucket (0.000405s)
2             Radix (0.001246s)             Radix (0.002378s)
3             Quick (0.001324s)             Quick (0.003008s)
4        Simple Tim (0.001462s)        Simple Tim (0.003102s)
```

```
5  Binary Insertion (0.001555s)             Merge (0.004056s)
6             Merge (0.001916s)  Binary Insertion (0.004537s)
7          Shell1000 (0.005101s)         Shell1000 (0.013417s)
8           Shell731 (0.007089s)          Shell731 (0.026349s)
9          Insertion (0.018981s)         Insertion (0.076569s)

                          4000                          8000   \
1             Bucket (0.000733s)            Bucket (0.001267s)
2              Radix (0.004622s)             Radix (0.009829s)
3         Simple Tim (0.006723s)             Quick (0.016247s)
4              Quick (0.006822s)        Simple Tim (0.016296s)
5              Merge (0.008482s)             Merge (0.018181s)
6  Binary Insertion (0.016499s)  Binary Insertion (0.066804s)
7          Shell1000 (0.036741s)         Shell1000 (0.099846s)
8           Shell731 (0.100089s)          Shell731 (0.386434s)
9          Insertion (0.299134s)         Insertion (1.224779s)

                         16000
1             Bucket (0.002480s)
2              Radix (0.019688s)
3         Simple Tim (0.031620s)
4              Merge (0.038370s)
5              Quick (0.041129s)
6          Shell1000 (0.281685s)
7  Binary Insertion (0.284079s)
8           Shell731 (1.547525s)
9          Insertion (4.817695s)
```

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

[7]:
```
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 (0.000216s)           Bucket (0.000389s)
2             Radix (0.001208s)            Radix (0.002332s)
3             Quick (0.001256s)            Quick (0.002812s)
4       Simple Tim (0.001378s)       Simple Tim (0.003078s)
5  Binary Insertion (0.001490s)            Merge (0.004011s)
6             Merge (0.001846s)  Binary Insertion (0.004535s)
7         Shell1000 (0.005031s)        Shell1000 (0.013325s)
8          Shell731 (0.006848s)         Shell731 (0.026712s)
9         Insertion (0.018644s)        Insertion (0.077796s)

                         4000                         8000   \
1            Bucket (0.000675s)           Bucket (0.001249s)
2             Radix (0.004618s)            Radix (0.009725s)
3             Quick (0.006574s)       Simple Tim (0.014634s)
4       Simple Tim (0.006730s)            Quick (0.016028s)
5             Merge (0.008529s)            Merge (0.018039s)
6  Binary Insertion (0.016692s)  Binary Insertion (0.066929s)
7         Shell1000 (0.037454s)        Shell1000 (0.100196s)
8          Shell731 (0.102651s)         Shell731 (0.390457s)
9         Insertion (0.311041s)        Insertion (1.244236s)

                         16000
1            Bucket (0.002457s)
2             Radix (0.019534s)
3       Simple Tim (0.031283s)
4             Merge (0.038290s)
```

```
5             Quick (0.041123s)
6         Shell1000 (0.279907s)
7  Binary Insertion (0.283342s)
8          Shell731 (1.554301s)
9         Insertion (4.964963s)
```

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.

[8]:
```
# 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}:")
```

```
    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          0.001916        NaN           n log n
5              Quick          0.001324        NaN              n^2
10         Insertion          0.018981        NaN              n^2
15          Shell731          0.007089        NaN              n^2
20         Shell1000          0.005101        NaN              n^2
25            Bucket          0.000217        NaN                n
30             Radix          0.001246        NaN               nd
35  Binary Insertion          0.001555        NaN              n^2
40        Simple Tim          0.001462        NaN           n log n
True Random runtimes at Data Size 2000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
1              Merge          0.004056   1.082267           n log n
6              Quick          0.003008   1.184039              n^2
11         Insertion          0.076569   2.012195              n^2
16          Shell731          0.026349   1.894021              n^2
21         Shell1000          0.013417   1.395067              n^2
26            Bucket          0.000405   0.900418                n
31             Radix          0.002378   0.932593               nd
36  Binary Insertion          0.004537   1.545013              n^2
41        Simple Tim          0.003102   1.085790           n log n
True Random runtimes at Data Size 4000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
2              Merge          0.008482   1.064340           n log n
7              Quick          0.006822   1.181388              n^2
12         Insertion          0.299134   1.965963              n^2
17          Shell731          0.100089   1.925468              n^2
22         Shell1000          0.036741   1.453388              n^2
27            Bucket          0.000733   0.855978                n
32             Radix          0.004622   0.958776               nd
37  Binary Insertion          0.016499   1.862753              n^2
42        Simple Tim          0.006723   1.115800           n log n
True Random runtimes at Data Size 8000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
3              Merge          0.018181   1.099941           n log n
8              Quick          0.016247   1.251987              n^2
13         Insertion          1.224779   2.033659              n^2
18          Shell731          0.386434   1.948941              n^2
23         Shell1000          0.099846   1.442309              n^2
28            Bucket          0.001267   0.788679                n
33             Radix          0.009829   1.088580               nd
38  Binary Insertion          0.066804   2.017529              n^2
43        Simple Tim          0.016296   1.277349           n log n
True Random runtimes at Data Size 16000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
```

```
4              Merge          0.038370   1.077586          n log n
9              Quick          0.041129   1.339942              n^2
14         Insertion          4.817695   1.975822              n^2
19          Shell731          1.547525   2.001671              n^2
24         Shell1000          0.281685   1.496302              n^2
29            Bucket          0.002480   0.969497                n
34             Radix          0.019688   1.002185               nd
39  Binary Insertion          0.284079   2.088286              n^2
44        Simple Tim          0.031620   0.956282          n log n
```

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

[9]:
```
# 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          0.001846        NaN          n log n
5              Quick          0.001256        NaN              n^2
10         Insertion          0.018644        NaN              n^2
15          Shell731          0.006848        NaN              n^2
20         Shell1000          0.005031        NaN              n^2
25            Bucket          0.000216        NaN                n
30             Radix          0.001208        NaN               nd
35  Binary Insertion          0.001490        NaN              n^2
40        Simple Tim          0.001378        NaN          n log n
Almost-sorted runtimes at Data Size 2000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
```

```
1              Merge          0.004011   1.119827           n log n
6              Quick          0.002812   1.162428               n^2
11         Insertion          0.077796   2.061011               n^2
16          Shell731          0.026712   1.963683               n^2
21         Shell1000          0.013325   1.405248               n^2
26            Bucket          0.000389   0.850441                 n
31             Radix          0.002332   0.948410                nd
36  Binary Insertion          0.004535   1.605531               n^2
41        Simple Tim          0.003078   1.159702           n log n
Almost-sorted runtimes at Data Size 4000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
2              Merge          0.008529   1.088393           n log n
7              Quick          0.006574   1.225453               n^2
12         Insertion          0.311041   1.999328               n^2
17          Shell731          0.102651   1.942200               n^2
22         Shell1000          0.037454   1.490999               n^2
27            Bucket          0.000675   0.794762                 n
32             Radix          0.004618   0.985718                nd
37  Binary Insertion          0.016692   1.880111               n^2
42        Simple Tim          0.006730   1.128694           n log n
Almost-sorted runtimes at Data Size 8000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
3              Merge          0.018039   1.080607           n log n
8              Quick          0.016028   1.285622               n^2
13         Insertion          1.244236   2.000086               n^2
18          Shell731          0.390457   1.927417               n^2
23         Shell1000          0.100196   1.419643               n^2
28            Bucket          0.001249   0.887197                 n
33             Radix          0.009725   1.074418                nd
38  Binary Insertion          0.066929   2.003471               n^2
43        Simple Tim          0.014634   1.120645           n log n
Almost-sorted runtimes at Data Size 16000:
                Algo  Observed Runtime  Emp Big-O Theoretical Big-O
4              Merge          0.038290   1.085838           n log n
9              Quick          0.041123   1.359386               n^2
14         Insertion          4.964963   1.996523               n^2
19          Shell731          1.554301   1.993030               n^2
24         Shell1000          0.279907   1.482119               n^2
29            Bucket          0.002457   0.976495                 n
34             Radix          0.019534   1.006197                nd
39  Binary Insertion          0.283342   2.081837               n^2
44        Simple Tim          0.031283   1.096119           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)).