In [144…
```
import pandas as pd
import numpy as np
from math import log2, e
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score
```

In [145…
```
df_inalmost = pd.read_csv("insertion_almost_sorted.csv",header=None, names=["Siz
df_inrandom = pd.read_csv("insertion_random.csv",header=None, names=["Size","Ela
df_inreverse = pd.read_csv("insertion_reverse.csv",header=None, names=["Size","E

logx, logy = np.log(x), np.log(y)
```

In [146…
```
 m, b = np.polyfit(logx, logy, 1)
```

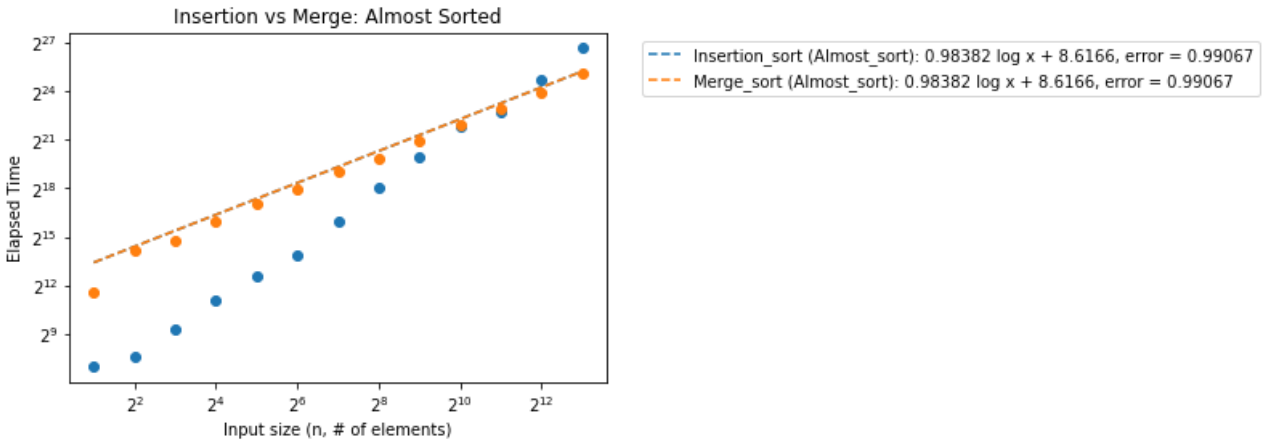
In [147…
```
fit = np.poly1d((m, b))
```

In [148…
```
expected_logy = fit(logx)
```

In [149…
```
r2 = r2_score(logy, expected_logy)
```

In [167…
```
plt.title("Insertion vs Merge: Almost Sorted")
x1 = df_inalmost['Size']
y1 = df_inalmost['Elapsed_Time']
p = plt.loglog(x1, y1, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x1[::len(x1)-1], (e ** expected_logy)[::len(y1)-1], '--', bas
label = f'{"Insertion_sort"} ({"Almost_sort"}): {m:0.5} log x + {b:.5}, error =
markersize = 6, color = p[-1].get_color())

x2 = df_malmost['Size']
y2 = df_malmost['Elapsed_Time']
p = plt.loglog(x2, y2, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x2[::len(x2)-1], (e ** expected_logy)[::len(y2)-1], '--', bas
label = f'{"Merge_sort"} ({"Almost_sort"}): {m:0.5} log x + {b:.5}, error = {r2:
markersize = 6, color = p[-1].get_color())

plt.xlabel('Input size (n, # of elements)')
plt.ylabel('Elapsed Time')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
```

Out[167… `<matplotlib.legend.Legend at 0x7fa15fd95b50>`

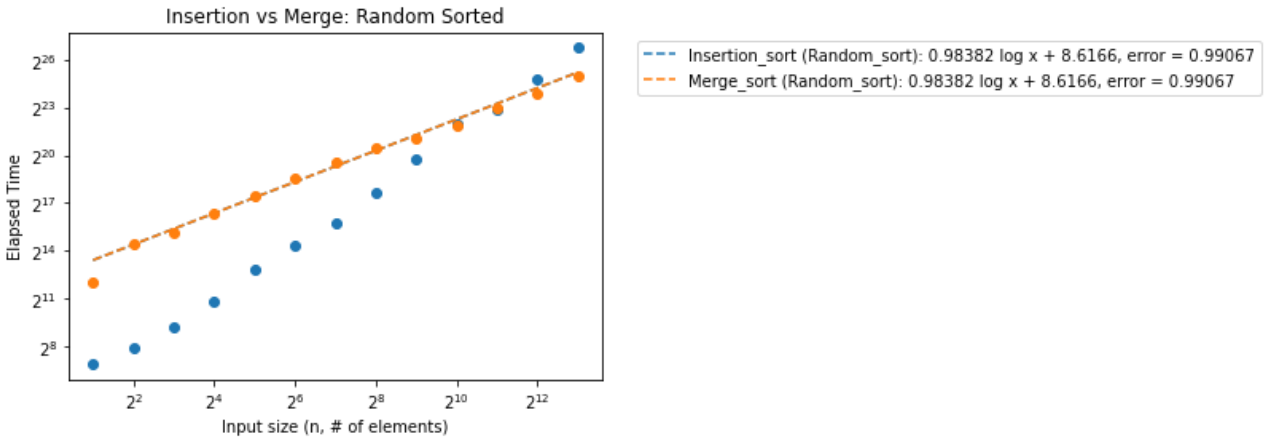


At low input sizes, insertion sort runs at significnatly faster rates. However as the number of

elements start to increase, merge sort starts to become faster. At asymptotically large input values, merge sort is the faster algorithm when it comes to almost sorted arrays.

In [151…
```
plt.title("Insertion vs Merge: Random Sorted")
x1 = df_inrandom['Size']
y1 = df_inrandom['Elapsed_Time']
p = plt.loglog(x1, y1, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x1[::len(x1)-1], (e ** expected_logy)[::len(y1)-1], '--', bas
label = f'{"Insertion_sort"} ({"Random_sort"}): {m:0.5} log x + {b:.5}, error =
markersize = 6, color = p[-1].get_color())

x2 = df_mrandom['Size']
y2 = df_mrandom['Elapsed_Time']
p = plt.loglog(x2, y2, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x2[::len(x2)-1], (e ** expected_logy)[::len(y2)-1], '--', bas
label = f'{"Merge_sort"} ({"Random_sort"}): {m:0.5} log x + {b:.5}, error = {r2:
markersize = 6, color = p[-1].get_color())

plt.xlabel('Input size (n, # of elements)')
plt.ylabel('Elapsed Time')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
```

Out[151… `<matplotlib.legend.Legend at 0x7fa15cd2ea30>`

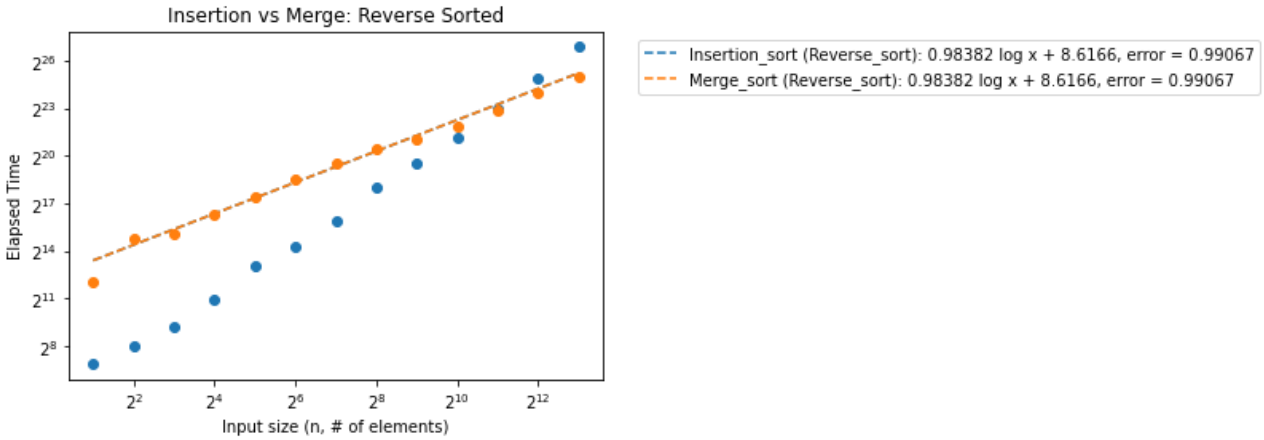


At low input sizes, again insertion sort runs at a faster rate. But again as the number of elements start to increase, merge sort starts to become faster. At asymptotically large input values, merge sort is the faster algorithm when it comes to randomly sorted arrays.

In [152…
```
plt.title("Insertion vs Merge: Reverse Sorted")
x1 = df_inreverse['Size']
y1 = df_inreverse['Elapsed_Time']
p = plt.loglog(x1, y1, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x1[::len(x1)-1], (e ** expected_logy)[::len(y1)-1], '--', bas
label = f'{"Insertion_sort"} ({"Reverse_sort"}): {m:0.5} log x + {b:.5}, error =
markersize = 6, color = p[-1].get_color())

x2 = df_mreverse['Size']
y2 = df_mreverse['Elapsed_Time']
p = plt.loglog(x2, y2, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x2[::len(x2)-1], (e ** expected_logy)[::len(y2)-1], '--', bas
label = f'{"Merge_sort"} ({"Reverse_sort"}): {m:0.5} log x + {b:.5}, error = {r2
markersize = 6, color = p[-1].get_color())

plt.xlabel('Input size (n, # of elements)')
```

```
plt.ylabel('Elapsed Time')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
```

Out[152… `<matplotlib.legend.Legend at 0x7fa15cc0c370>`



Similar to the other two arrays, at low input sizes insertion sort does run at a faster rate. And again as the number of elements start to increase, merge sort starts to become faster. At asymptotically large input values, merge sort is the faster algorithm when it comes to reverse sorted arrays.

In [153…
```
df_malmost = pd.read_csv("merge_almost_sorted.csv",header=None, names=["Size","E
df_mrandom = pd.read_csv("merge_random.csv",header=None, names=["Size","Elapsed_
df_mreverse = pd.read_csv("merge_reverse.csv",header=None, names=["Size","Elapse
```

In [154…
```
df_s1almost = pd.read_csv("shell1_almost_sorted.csv",header=None, names=["Size",
df_s1random = pd.read_csv("shell1_random.csv",header=None, names=["Size","Elapse
df_s1reverse = pd.read_csv("shell1_reverse.csv",header=None, names=["Size","Elap
```

In [155…
```
df_s2almost = pd.read_csv("shell2_almost_sorted.csv",header=None, names=["Size",
df_s2random = pd.read_csv("shell2_random.csv",header=None, names=["Size","Elapse
df_s2reverse = pd.read_csv("shell2_reverse.csv",header=None, names=["Size","Elap
```

In [156…
```
df_s3almost = pd.read_csv("shell3_almost_sorted.csv",header=None, names=["Size",
df_s3random = pd.read_csv("shell3_random.csv",header=None, names=["Size","Elapse
df_s3reverse = pd.read_csv("shell3_reverse.csv",header=None, names=["Size","Elap
```

In [157…
```
df_s4almost = pd.read_csv("shell4_almost_sorted.csv",header=None, names=["Size",
df_s4random = pd.read_csv("shell4_random.csv",header=None, names=["Size","Elapse
df_s4reverse = pd.read_csv("shell4_reverse.csv",header=None, names=["Size","Elap
```

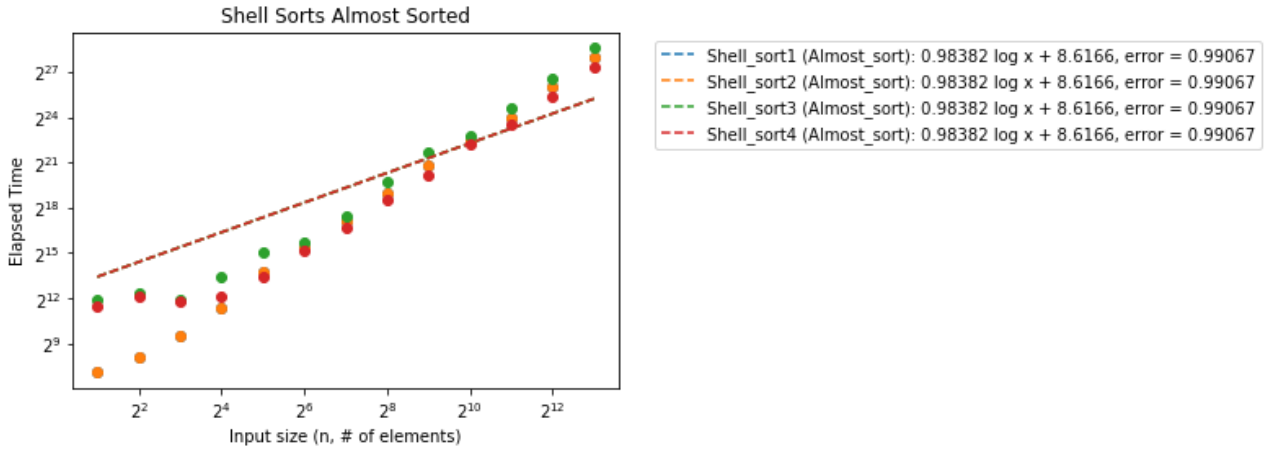
In [158…
```
df_h1almost = pd.read_csv("hybrid1_almost_sorted.csv",header=None, names=["Size"
df_h1random = pd.read_csv("hybrid1_random.csv",header=None, names=["Size","Elaps
df_h1reverse = pd.read_csv("hybrid1_reverse.csv",header=None, names=["Size","Ela
```

In [159…
```
df_h2almost = pd.read_csv("hybrid2_almost_sorted.csv",header=None, names=["Size"
df_h2random = pd.read_csv("hybrid2_random.csv",header=None, names=["Size","Elaps
df_h2reverse = pd.read_csv("hybrid2_reverse.csv",header=None, names=["Size","Ela
```

In [160…
```
df_h3almost = pd.read_csv("hybrid3_almost_sorted.csv",header=None, names=["Size"
df_h3random = pd.read_csv("hybrid3_ranomd.csv",header=None, names=["Size","Elaps
df_h3reverse = pd.read_csv("hybrid3_reverse.csv",header=None, names=["Size","Ela
```

In [161…
```
plt.title("Shell Sorts Almost Sorted")
x = df_s1almost['Size']
y = df_s1almost['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Shell_sort1"} ({"Almost_sort"}): {m:0.5} log x + {b:.5}, error = {r2
markersize = 6, color = p[-1].get_color())

x2 = df_s2almost['Size']
y2 = df_s2almost['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x2[::len(x2)-1], (e ** expected_logy)[::len(y2)-1], '--', bas
label = f'{"Shell_sort2"} ({"Almost_sort"}): {m:0.5} log x + {b:.5}, error = {r2
markersize = 6, color = p[-1].get_color())

x = df_s3almost['Size']
y = df_s3almost['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x2[::len(x2)-1], (e ** expected_logy)[::len(y2)-1], '--', bas
label = f'{"Shell_sort3"} ({"Almost_sort"}): {m:0.5} log x + {b:.5}, error = {r2
markersize = 6, color = p[-1].get_color())

x = df_s4almost['Size']
y = df_s4almost['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x2[::len(x2)-1], (e ** expected_logy)[::len(y2)-1], '--', bas
label = f'{"Shell_sort4"} ({"Almost_sort"}): {m:0.5} log x + {b:.5}, error = {r2
markersize = 6, color = p[-1].get_color())

plt.xlabel('Input size (n, # of elements)')
plt.ylabel('Elapsed Time')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
```

Out[161… <matplotlib.legend.Legend at 0x7fa15f2e8100>

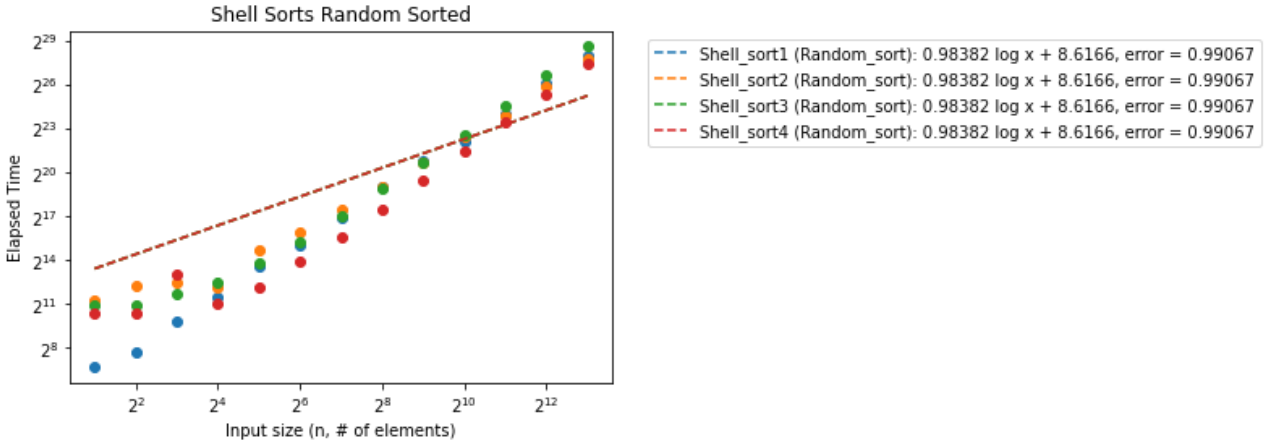


The various shell sorts have interesting patterns. At very low input sizes, shell sort 2 is the fastest with shell sort 3 being the clear slowest one. However, as the input sizes get asymptotically large, shell sort 3 actually becomes the fastest algorithm when it comes to almost sorted arrays.

In [162…
```
plt.title("Shell Sorts Random Sorted")
x = df_s1random['Size']
y = df_s1random['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
```

```
label = f'{"Shell_sort1"} ({"Random_sort"}): {m:0.5} log x + {b:.5}, error = {r2
markersize = 6, color = p[-1].get_color())



x = df_s2random['Size']
y = df_s2random['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Shell_sort2"} ({"Random_sort"}): {m:0.5} log x + {b:.5}, error = {r2
markersize = 6, color = p[-1].get_color())

x = df_s3random['Size']
y = df_s3random['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Shell_sort3"} ({"Random_sort"}): {m:0.5} log x + {b:.5}, error = {r2
markersize = 6, color = p[-1].get_color())

x = df_s4random['Size']
y = df_s4random['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Shell_sort4"} ({"Random_sort"}): {m:0.5} log x + {b:.5}, error = {r2
markersize = 6, color = p[-1].get_color())

plt.xlabel('Input size (n, # of elements)')
plt.ylabel('Elapsed Time')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
```

Out[162… `<matplotlib.legend.Legend at 0x7fa15f666d00>`



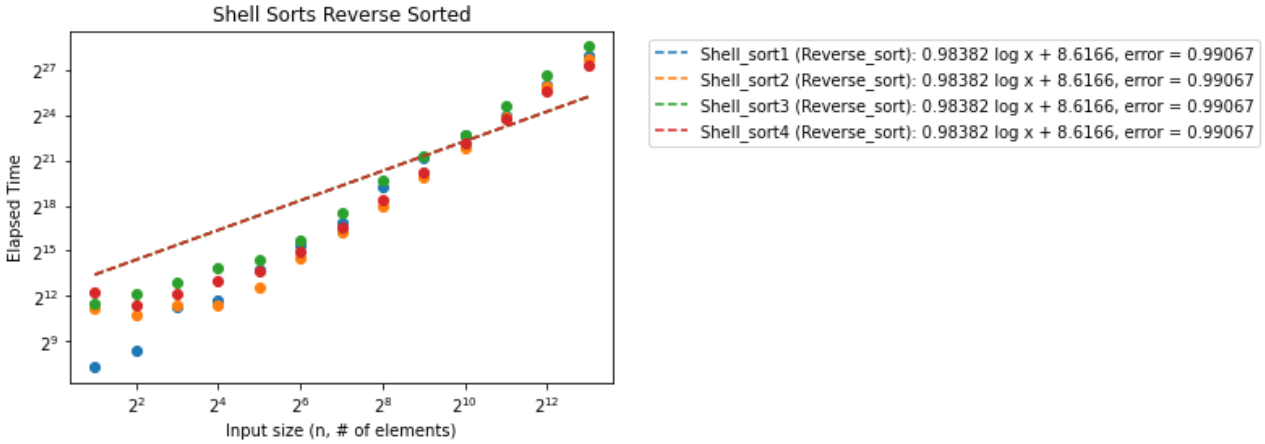
Unlike the almost sorted array, the random sorted array is sorted fastest by shell sort 1. The intermediate values are all roughly the same speed, however shell sort 4 has a slight advantage. As input values get significantly larger and larger, shell sort 4 does have the fastest speed in randomly sorted arrays.

In [163…

```
plt.title("Shell Sorts Reverse Sorted")
x = df_s1reverse['Size']
y = df_s1reverse['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Shell_sort1"} ({"Reverse_sort"}): {m:0.5} log x + {b:.5}, error = {r
markersize = 6, color = p[-1].get_color())
```

```
x = df_s2reverse['Size']
y = df_s2reverse['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Shell_sort2"} ({"Reverse_sort"}): {m:0.5} log x + {b:.5}, error = {r
markersize = 6, color = p[-1].get_color())

x = df_s3reverse['Size']
y = df_s3reverse['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Shell_sort3"} ({"Reverse_sort"}): {m:0.5} log x + {b:.5}, error = {r
markersize = 6, color = p[-1].get_color())

x = df_s4reverse['Size']
y = df_s4reverse['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Shell_sort4"} ({"Reverse_sort"}): {m:0.5} log x + {b:.5}, error = {r
markersize = 6, color = p[-1].get_color())

plt.xlabel('Input size (n, # of elements)')
plt.ylabel('Elapsed Time')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
```

Out[163… `<matplotlib.legend.Legend at 0x7fa15f294c70>`



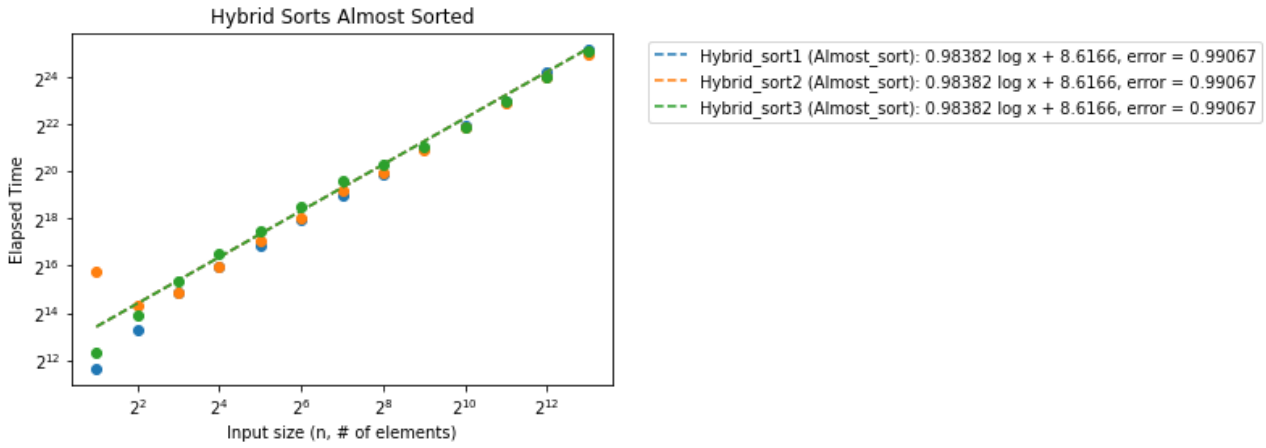
For small input values, shell sort 1 has the fastest speed for sorting reverse sorted arrays. For a small period of time in between 64-128 values, shell sort 2 is the fastest algorithm. As the input values asympototically grow, shell sort 4 becomes the fasted in reverse sorted algorithms.

In [164…

```
plt.title("Hybrid Sorts Almost Sorted")
x = df_h1almost['Size']
y = df_h1almost['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Hybrid_sort1"} ({"Almost_sort"}): {m:0.5} log x + {b:.5}, error = {r
markersize = 6, color = p[-1].get_color())

x = df_h2almost['Size']
y = df_h2almost['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Hybrid_sort2"} ({"Almost_sort"}): {m:0.5} log x + {b:.5}, error = {r
markersize = 6, color = p[-1].get_color())
```

```
x = df_h3almost['Size']
y = df_h3almost['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Hybrid_sort3"} ({"Almost_sort"}): {m:0.5} log x + {b:.5}, error = {r
markersize = 6, color = p[-1].get_color())

plt.xlabel('Input size (n, # of elements)')
plt.ylabel('Elapsed Time')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
```

Out[164… <matplotlib.legend.Legend at 0x7fa15daf5d00>

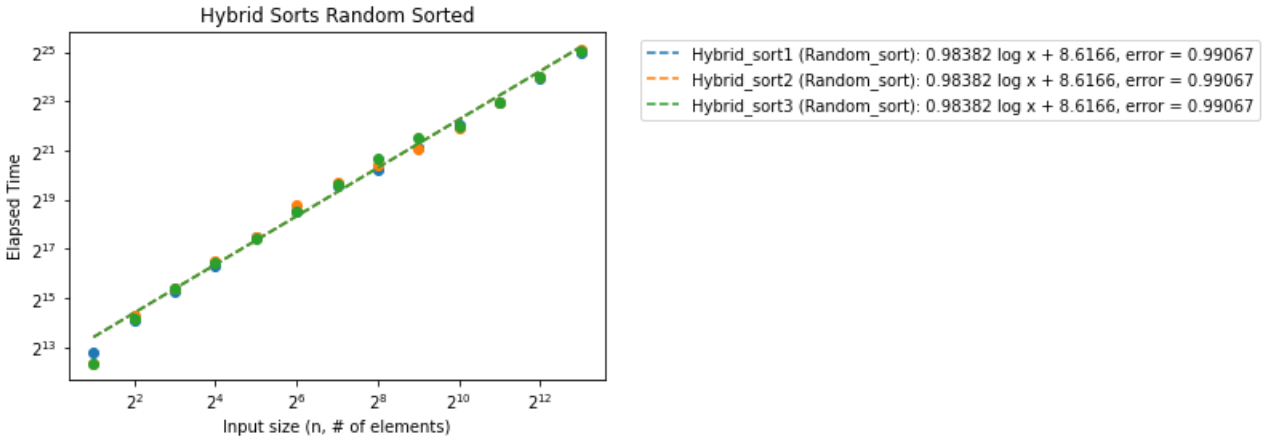


For extremely low input values, hybrid sort 1 is the fastest algorithm for sorting almost sorted arrays. Hybrid sort 2 os by far the slowest amongst the 3 at low input values. However, as the input values increase, all 3 algorithms tend to converge reaching very similar run times. It does seem that hybrid sort 2 has a slight advantage over the other 2 at extremely large input values.

In [165…

```
plt.title("Hybrid Sorts Random Sorted")
x = df_h1random['Size']
y = df_h1random['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Hybrid_sort1"} ({"Random_sort"}): {m:0.5} log x + {b:.5}, error = {r
markersize = 6, color = p[-1].get_color())

x = df_h2random['Size']
y = df_h2random['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Hybrid_sort2"} ({"Random_sort"}): {m:0.5} log x + {b:.5}, error = {r
markersize = 6, color = p[-1].get_color())

x = df_h3random['Size']
y = df_h3random['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Hybrid_sort3"} ({"Random_sort"}): {m:0.5} log x + {b:.5}, error = {r
markersize = 6, color = p[-1].get_color())


plt.xlabel('Input size (n, # of elements)')
plt.ylabel('Elapsed Time')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
```

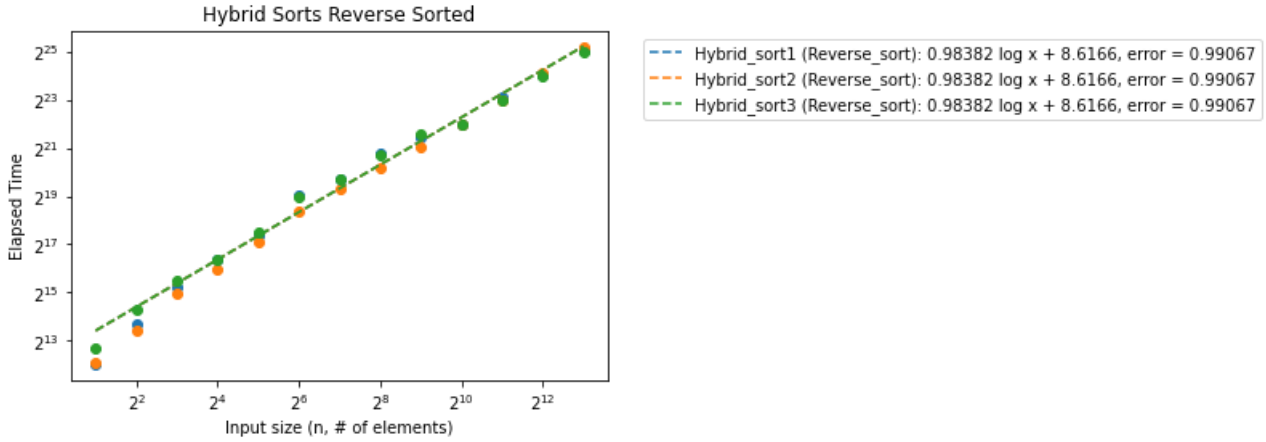
Out[165… <matplotlib.legend.Legend at 0x7fa15eed45b0>



Both at low input values and high input values, hybrid sort 3 is the fastest at sorting randomly sorted arrays. However, there is a slight jump in runtime between 128-256 elements where hybrid sort 1 and 2 start to take a slight advantage. As the numbers increase above 256, hybrid sort 3 is the most optimal.

In [166…
```
plt.title("Hybrid Sorts Reverse Sorted")
x = df_h1reverse['Size']
y = df_h1reverse['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Hybrid_sort1"} ({"Reverse_sort"}): {m:0.5} log x + {b:.5}, error = {
markersize = 6, color = p[-1].get_color())

x = df_h2reverse['Size']
y = df_h2reverse['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Hybrid_sort2"} ({"Reverse_sort"}): {m:0.5} log x + {b:.5}, error = {
markersize = 6, color = p[-1].get_color())

x = df_h3reverse['Size']
y = df_h3reverse['Elapsed_Time']
p = plt.loglog(x, y, '.', base= 2, markersize = 12)
fit_p = plt.loglog(x[::len(x)-1], (e ** expected_logy)[::len(y)-1], '--', base =
label = f'{"Hybrid_sort3"} ({"Reverse_sort"}): {m:0.5} log x + {b:.5}, error = {
markersize = 6, color = p[-1].get_color())

plt.xlabel('Input size (n, # of elements)')
plt.ylabel('Elapsed Time')
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
```

Out[166… <matplotlib.legend.Legend at 0x7fa15fa8ea60>



At very low input values, hybrid sort 1 and 2 are almost identical in terms of runtime but hybrid sort 1 has a slight advantage. Hybrid sort 3 is the slowest throughout until the input size reaches above 1,000. Once the number of elements exceeds this size, hybrid sort 3 becomes the fastest algorithm to sort reverse sorted arrays.

Similarities: It seems as though merge sort stays pretty consistent in terms of runtime no matter what type of array it is being passed.

Shell sort 2 for almost sorted arrays seems to have extremely similar patterns to shell sort 1 for randomly sorted arrays.

Hybrid sort 2 for almost sorted arrays also has a similar pattern to shell sort 4 for reverse sorted arrays however the shell sort definitely has an advantage between the two.

Almost all hybrid sorts are very similar in run time and have a very linear pattern between runtime and input size.

Differences: Hybrid sort 2 for almost sorted arrays jump out to be quite different than the rest when the input values are low. The runtime is signifantly slower than the others but it catches up as the input values increase.

Shell sort 1 for reverse sorted arrays on the other hand are significantly faster than the rest for very small values but evens out as the number of input values increase.

For almost sorted arrays, shell sort 1 and 2 differ greatly from shell sort 3 and 4. Although at low input sizes 1 & 2 are very similar to each other, and 3 & 4 are very similar to each other, the pairs are opposites as 1 & 2 are very fast in relation to 3 & 4. However as the input values increase, they merge together.

Winners: Shell sort wins as the fastest algorithm for small input sizes i.e. less than ~30, as its runtime can be as low as 2^9 nanoseconds compared to the other sorting algorithms which tend to be well over 2^13 nanoseconds.

As far as asymptotically large numbers, hybrid sort is the clear winner. For about 2^12 input values, hybrid sort sorts the array in roughly 2^25 nanoseconds, more than half the time of all the other sorting algorithms. Additionally, it grows in a linear pattern which means the runtime does not increase as fast as the other sorting algorithms as input values increase. This was my

guess to be the fastest algorithm as well for sorting arrays because instertion sort excels at low input values and merge sort excels at high input values. Using a hybrid of the two to switch from insertion sort to merge sort as values increase is an exceptional way to get the best of both worlds. In particular, hybrid sort 3 seemed to be the fastest at large values. This makes sense becasue the switching point is the 6th root of the input size. As input sizes increase, the 6th root is the smallest of the other hybrid sort H values so it switches to merge sort quicker. As values increase, there will always be a faster hybrid sort. In other words, as n increases and H = n^(1/x), we would need a larger and larger x value to obtain the optimal hybrid sort.