Sorting Algorithm Experiments

The purpose of these experiments is to compare and contrast the efficiency of a variety of sorting algorithms on different types of data. I implemented insertion sort, merge sort, hybrid sort, and shell sort in C++. Then, I tested them on uniformly distributed numbers, almost sorted numbers, and reverse order numbers.

Andrew Self

In [6]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import math
%matplotlib inline
```

In [3]:
```
def getMeans(df):
    means = []
    vals = pd.unique(df["Size"])
    #print(vals)
    for i in vals:
        temp = df.loc[df['Size'] == i]
        #print(i)
        #print(temp["Time"].mean())
        means.append(temp["Time"].mean())
    return means, vals
```

For each algorithm and each type of data, I would average the results of 5-10 trials. Times were collected by using the Linux usr/bin/time program.

In [4]:
```
def getLog(times, sizes):
    logTime = [math.log2(x) if x > 0.0 else math.log2(0.01) for x in times]
    logSize = [math.log2(x) for x in sizes]
    return logTime, logSize
```

For many sorting results log-log plots are more meaningful because they represent functions of the form f(x) = a*(x^k) as a straight line with slope k.

Types of Data

Uniformly Distributed Permutations:

Permutations of the numbers, 1,2,3,...,n, where all permutations are equally likely.

Almost-Sorted Permutations

Start with a sorted array/vector of n numbers, say, the numbers 1,2,3,...,n, in this order. Then, independently choose 2log n pairs, (i,j), where i and j are uniformly-chosen random integers in the range from 0 to n-1, and swap the numbers at positions i and j in the array/vector.

Reverse-Sorted Permutations

This is the permutation (n,n-1,n-2,...,3,2,1).

Insertion Vs. Merge Sort:

Insertion Sort:

Insertion sort works by moving one element at a time. It iterates through an array and finds a smaller element too far to the right, finds its correct spot on the left, and shifts the other elements down to the right.

Merge Sort:

Merge sort works by utilizing the divide and conquer method. It recursively splits arrays in half until they are single elements, the zips them back up doubling the length of the sub arrays each time, making it run in O(nlogn) time.

Uniformly Distributed Permutations

In order to achieve true randomness, opposed to only psuedo-random numbers, I used the random number generator from the libsodium c++ library.

In [5]:
```
#read insertion data into pandas
df = pd.read_csv('uniform/insertion_uniform.csv', sep=' ')
times, sizes = getMeans(df)
logTime, logSize = getLog(times,sizes)
m, b = np.polyfit(logSize,logTime,1)
print(m,b)
```

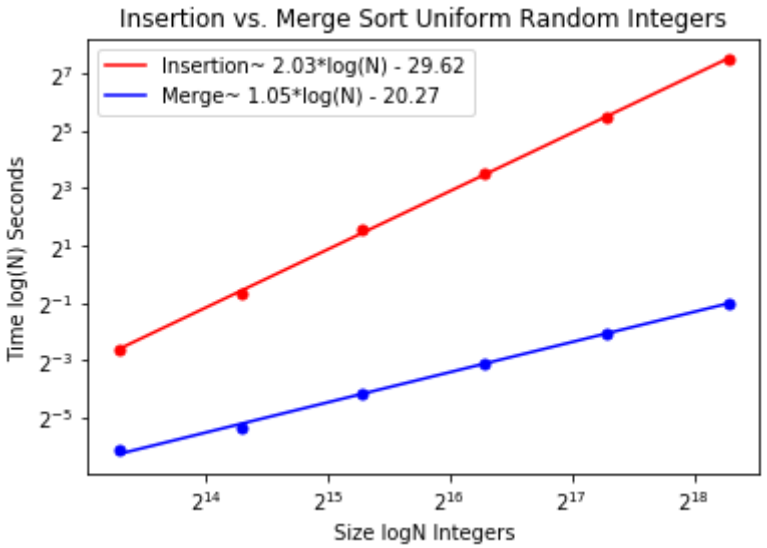
```
2.0324558345510253 -29.62280736583659
```

In [269]:
```
#read merge data into pandas
df2 = pd.read_csv('uniform/merge_uniform.csv', sep=' ')
times2, sizes2 = getMeans(df2)
logTime2, logSize2 = getLog(times2,sizes2)
m2, b2 = np.polyfit(logSize2,logTime2,1)
print(m2,b2)
```

```
1.052805156035237 -20.266674421482005
```

In [270]:
```
#plot loglog of insertion data
plt.loglog(sizes, times, '.', basex=2,basey=2, markersize=10, color='red')
fn = [2**(m * math.log2(x) + b) for x in sizes]
plt.loglog(sizes, fn,basex=2, basey=2, color='red', label='Insertion~ 2.03*log(N) - 29.62')
# plt.show()

#plot loglog of merge data
plt.loglog(sizes2, times2, '.', basex=2,basey=2, markersize=10, color='blue')
fn2 = [2**(m2 * math.log2(x) + b2) for x in sizes2]
plt.loglog(sizes2, fn2,basex=2, basey=2, label = 'Merge~ 1.05*log(N) - 20.27', color='blue')

plt.xlabel('Size logN Integers')
plt.ylabel('Time log(N) Seconds')
plt.title('Insertion vs. Merge Sort Uniform Random Integers')
plt.legend()
```

Out[270]: <matplotlib.legend.Legend at 0x120448a10>



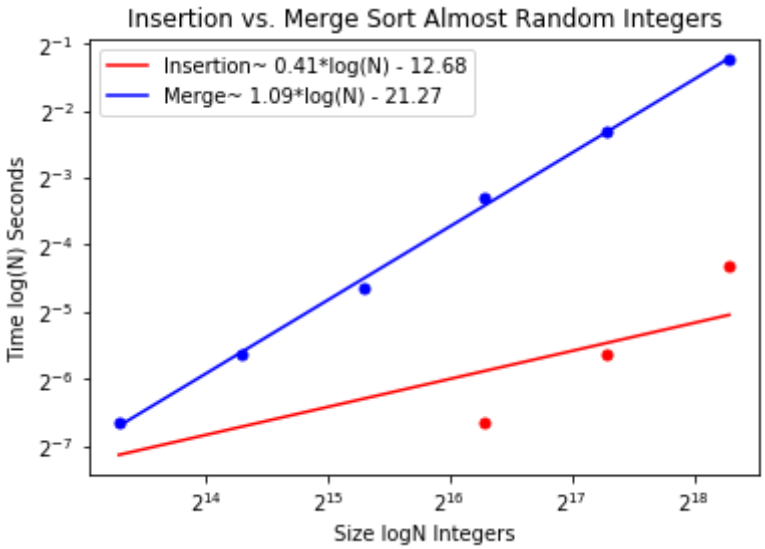
On uniformly random integers, we can see that merge Sort has a slope of 1.05 and there for is running in almost linear time while insertion Sort has a slope of 2.03 and is running in exponential time.

Almost-Sorted Permutations

In [273]:
```
#insertion almost sorted data
df = pd.read_csv('almost/insertion_almost.csv', sep=' ')
times, sizes = getMeans(df)
logTime, logSize = getLog(times,sizes)
m, b = np.polyfit(logSize,logTime,1)
print(m,b)

#merge data
df2 = pd.read_csv('almost/merge_almost.csv', sep=' ')
times2, sizes2 = getMeans(df2)
logTime2, logSize2 = getLog(times2,sizes2)
m2, b2 = np.polyfit(logSize2,logTime2,1)
print(m2, b2)

#insertion
plt.loglog(sizes, times, '.', basex=2,basey=2, markersize=10, color='red')
fn = [2**(m * math.log2(x) + b) for x in sizes]
plt.loglog(sizes, fn,basex=2, basey=2, color='red', label='Insertion~ 0.41*log(N) - 12.68')
# plt.show()
#merge
plt.loglog(sizes2, times2, '.', basex=2,basey=2, markersize=10, color='blue')
fn2 = [2**(m2 * math.log2(x) + b2) for x in sizes2]
plt.loglog(sizes2, fn2,basex=2, basey=2, label = 'Merge~ 1.09*log(N) - 21.27', color='blue')

plt.xlabel('Size logN Integers')
plt.ylabel('Time log(N) Seconds')
plt.title('Insertion vs. Merge Sort Almost Random Integers')
plt.legend()
```

```
0.41741829926962287 -12.680281558122998
1.0968374059730464 -21.278280849970887
```

Out[273]:
```
<matplotlib.legend.Legend at 0x1207e6790>
```



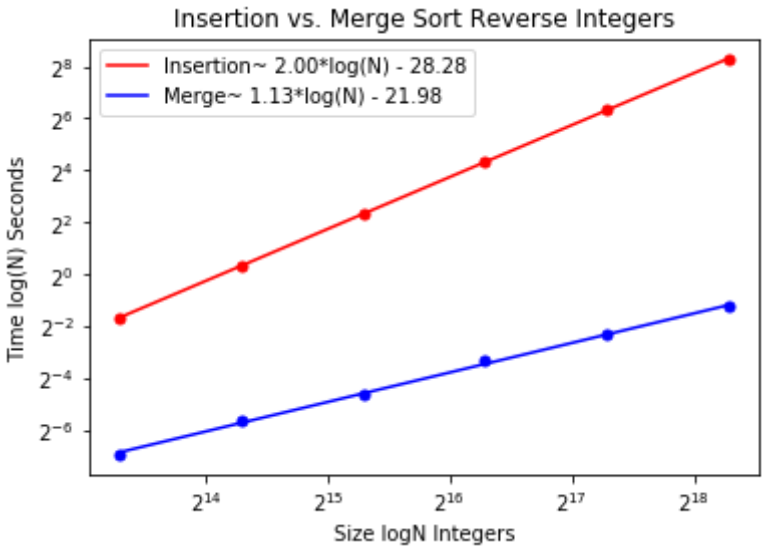
Here we can see on almost-sorted permutations, insertion sort, with a slope of 0.41, actually runs faster than merge, with a slope of 1.09. This graph is a little hindered by the fact that Insertion Sort was measured at taking 0.00 seconds for less than 2^10 integers.

Reverse-Sorted Permutations

In [272]:
```
#insertion data
df = pd.read_csv('reverse/insertion_reverse.csv', sep=' ')
times, sizes = getMeans(df)
logTime, logSize = getLog(times,sizes)
m, b = np.polyfit(logSize,logTime,1)
print(m,b)

#merge data
df2 = pd.read_csv('reverse/merge_reverse.csv', sep=' ')
times2, sizes2 = getMeans(df2)
logTime2, logSize2 = getLog(times2,sizes2)
m2, b2 = np.polyfit(logSize2,logTime2,1)
print(m2, b2)

#insertion
plt.loglog(sizes, times, '.', basex=2,basey=2, markersize=10, color='red')
fn = [2**(m * math.log2(x) + b) for x in sizes]
plt.loglog(sizes, fn,basex=2, basey=2, color='red', label='Insertion~ 2.00*log(N) - 28.28')
# plt.show()
#merge
plt.loglog(sizes2, times2, '.', basex=2,basey=2, markersize=10, color='blue')
fn2 = [2**(m2 * math.log2(x) + b2) for x in sizes2]
plt.loglog(sizes2, fn2,basex=2, basey=2, label = 'Merge~ 1.13*log(N) - 21.98', color='blue')

plt.xlabel('Size logN Integers')
plt.ylabel('Time log(N) Seconds')
plt.title('Insertion vs. Merge Sort Reverse Integers')
plt.legend()
```

```
2.0013058039871368 -28.281640391373543
1.1370103341226958 -21.981612853075696
```

Out[272]:
```
<matplotlib.legend.Legend at 0x120828f90>
```



Here we can see that merge Sort runs a lot better than insertion Sort when many swaps need to be made. Merge sort has a slope of 1.13 and insertion sort has a slope of 2.0.

Shell Sort Algorithms

The shell sort algorithms work similiarly to insertion sort. Except instead of moving one element at a time, they start sorting elements at some distance d apart, decreasing d at each pass through the array until it is 1. Different integer sequences, such as the Pratt sequence, can be used to determine d.

Shellsort1: the original Shell sequence, [n/2k], ..., 1, for k=1,2,...,log n, where [*] denotes the floor function

Shellsort2: 2k-1, for k=log n, ..., 3, 2, 1 https://oeis.org/A000225 (https://oeis.org/A000225)

Shellsort3: the Pratt sequence, 2p3q, ordered from the largest such number less than n down to 1

Shellsort4: 4^(n+1) + 3*2^n + 1 https://oeis.org/A036562 (https://oeis.org/A036562)

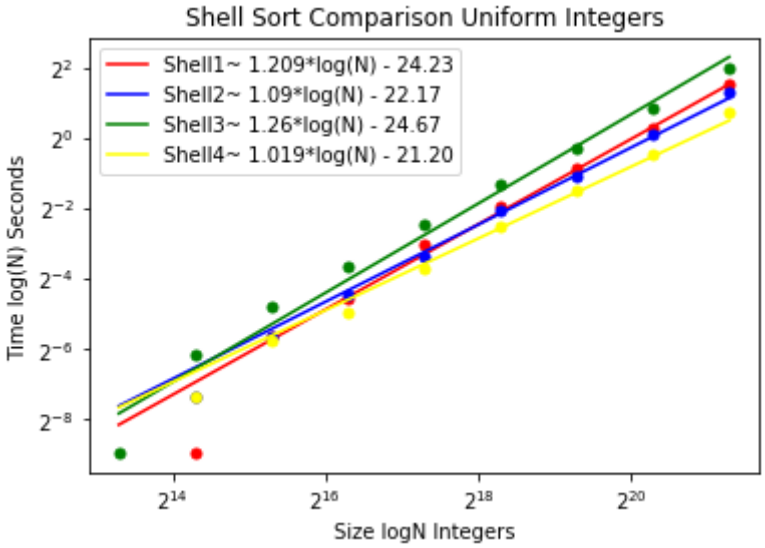
Uniformly Distributed Permutations

In [276]:
```
#shellsort1 data
df = pd.read_csv('uniform/shell_sort1_uniform.csv', sep=' ')
times, sizes = getMeans(df)
logTime, logSize = getLog(times,sizes)
m, b = np.polyfit(logSize,logTime,1)
print(m,b)

#shellsort2 data
df2 = pd.read_csv('uniform/shell_sort2_uniform.csv', sep=' ')
times2, sizes2 = getMeans(df2)
logTime2, logSize2 = getLog(times2,sizes2)
m2, b2 = np.polyfit(logSize2,logTime2,1)
print(m2,b2)

#shellsort3 data
df3 = pd.read_csv('uniform/shell_sort3_uniform.csv', sep=' ')
times3, sizes3 = getMeans(df3)
logTime3, logSize3 = getLog(times3,sizes3)
m3, b3 = np.polyfit(logSize3,logTime3,1)
print(m3,b3)

#shellsort14 data
df4 = pd.read_csv('uniform/shell_sort4_uniform.csv', sep=' ')
times4, sizes4 = getMeans(df4)
logTime4, logSize4 = getLog(times4,sizes4)
m4, b4 = np.polyfit(logSize4,logTime4,1)
print(m4,b4)

#shell1
plt.loglog(sizes, times, '.', basex=2,basey=2, markersize=10, color='red')
fn = [2**(m * math.log2(x) + b) for x in sizes]
plt.loglog(sizes, fn,basex=2, basey=2, color='red', label='Shell1~ 1.209*log(N) - 24.23')

#shell2
plt.loglog(sizes2, times2, '.', basex=2,basey=2, markersize=10, color='blue')
fn2 = [2**(m2 * math.log2(x) + b2) for x in sizes2]
plt.loglog(sizes2, fn2,basex=2, basey=2, label = 'Shell2~ 1.09*log(N) - 22.17', color='blue')

#shell3
plt.loglog(sizes3, times3, '.', basex=2,basey=2, markersize=10, color='green')
fn3 = [2**(m3 * math.log2(x) + b3) for x in sizes3]
plt.loglog(sizes3, fn3,basex=2, basey=2, label = 'Shell3~ 1.26*log(N) - 24.67', color='green')
```

```
#shell4
plt.loglog(sizes4, times4, '.', basex=2,basey=2, markersize=10, color='yellow')
fn4 = [2**(m4 * math.log2(x) + b4) for x in sizes4]
plt.loglog(sizes4, fn4,basex=2, basey=2, label = 'Shell4~ 1.019*log(N) - 21.20', color='yellow')

plt.xlabel('Size logN Integers')
plt.ylabel('Time log(N) Seconds')
plt.title('Shell Sort Comparison Uniform Integers')
plt.legend()
```

```
1.2099492525483317 -24.23860016347638
1.0947181142746247 -22.174996086858854
1.2667953936527365 -24.672343380963227
1.0191974475499743 -21.20895716360359
```

Out[276]: <matplotlib.legend.Legend at 0x1206f31d0>



This chart demonstrates that all shell sort variations run at very similiar speeds on uniformly random integers.

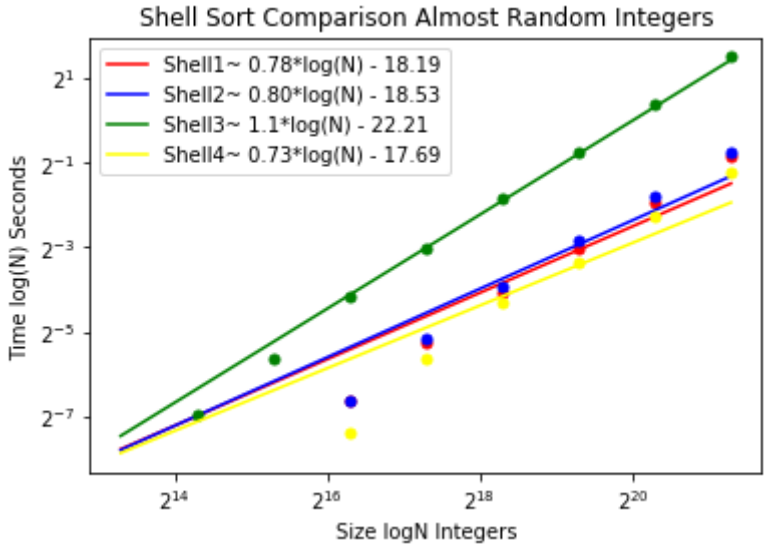
Almost-Sorted Permutations

In [275]:
```
#shellsort1 data
df = pd.read_csv('almost/shell_sort1_almost.csv', sep=' ')
times, sizes = getMeans(df)
logTime, logSize = getLog(times,sizes)
m, b = np.polyfit(logSize,logTime,1)
print(m,b)

#shellsort2 data
df2 = pd.read_csv('almost/shell_sort2_almost.csv', sep=' ')
times2, sizes2 = getMeans(df2)
logTime2, logSize2 = getLog(times2,sizes2)
m2, b2 = np.polyfit(logSize2,logTime2,1)
print(m2,b2)

#shellsort3 data
df3 = pd.read_csv('almost/shell_sort3_almost.csv', sep=' ')
times3, sizes3 = getMeans(df3)
logTime3, logSize3 = getLog(times3,sizes3)
m3, b3 = np.polyfit(logSize3,logTime3,1)
print(m3,b3)

#shellsort14 data
df4 = pd.read_csv('almost/shell_sort4_almost.csv', sep=' ')
times4, sizes4 = getMeans(df4)
logTime4, logSize4 = getLog(times4,sizes4)
m4, b4 = np.polyfit(logSize4,logTime4,1)
print(m4,b4)

#shell1
plt.loglog(sizes, times, '.', basex=2,basey=2, markersize=10, color='red')
fn = [2**(m * math.log2(x) + b) for x in sizes]
plt.loglog(sizes, fn,basex=2, basey=2, color='red', label='Shell1~ 0.78*log(N) - 18.19')

#shell2
plt.loglog(sizes2, times2, '.', basex=2,basey=2, markersize=10, color='blue')
fn2 = [2**(m2 * math.log2(x) + b2) for x in sizes2]
plt.loglog(sizes2, fn2,basex=2, basey=2, label = 'Shell2~ 0.80*log(N) - 18.53', color='blue')

#shell3
plt.loglog(sizes3, times3, '.', basex=2,basey=2, markersize=10, color='green')
fn3 = [2**(m3 * math.log2(x) + b3) for x in sizes3]
plt.loglog(sizes3, fn3,basex=2, basey=2, label = 'Shell3~ 1.1*log(N) - 22.21', color='green')
```

```
#shell4
plt.loglog(sizes4, times4, '.', basex=2,basey=2, markersize=10, color='yellow')
fn4 = [2**(m4 * math.log2(x) + b4) for x in sizes4]
plt.loglog(sizes4, fn4,basex=2, basey=2, label = 'Shell4~ 0.73*log(N) - 17.69', color='yellow')

plt.xlabel('Size logN Integers')
plt.ylabel('Time log(N) Seconds')
plt.title('Shell Sort Comparison Almost Random Integers')
plt.legend()
```

```
0.784573250196301 -18.199460282168364
0.8086568861868654 -18.53753914656672
1.1100022867194084 -22.209350845843364
0.7397647367960118 -17.694217667036213
```

Out[275]: <matplotlib.legend.Legend at 0x120e31b90>



The third variation of shell sort runs significantly slower than the others on almost-sorted permutations.

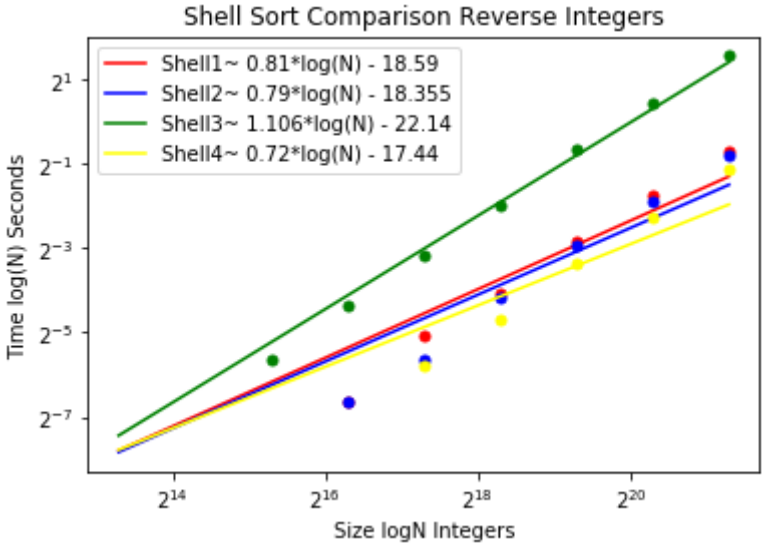
Reverse-Order Permutations

In [277]:
```
#shellsort1 data
df = pd.read_csv('reverse/shell_sort1_reverse.csv', sep=' ')
times, sizes = getMeans(df)
logTime, logSize = getLog(times,sizes)
m, b = np.polyfit(logSize,logTime,1)
print(m,b)

#shellsort2 data
df2 = pd.read_csv('reverse/shell_sort2_reverse.csv', sep=' ')
times2, sizes2 = getMeans(df2)
logTime2, logSize2 = getLog(times2,sizes2)
m2, b2 = np.polyfit(logSize2,logTime2,1)
print(m2,b2)

#shellsort3 data
df3 = pd.read_csv('reverse/shell_sort3_reverse.csv', sep=' ')
times3, sizes3 = getMeans(df3)
logTime3, logSize3 = getLog(times3,sizes3)
m3, b3 = np.polyfit(logSize3,logTime3,1)
print(m3,b3)

#shellsort14 data
df4 = pd.read_csv('reverse/shell_sort4_reverse.csv', sep=' ')
times4, sizes4 = getMeans(df4)
logTime4, logSize4 = getLog(times4,sizes4)
m4, b4 = np.polyfit(logSize4,logTime4,1)
print(m4,b4)

#shell1
plt.loglog(sizes, times, '.', basex=2,basey=2, markersize=10, color='red')
fn = [2**(m * math.log2(x) + b) for x in sizes]
plt.loglog(sizes, fn,basex=2, basey=2, color='red', label='Shell1~ 0.81*log(N) - 18.59')

#shell2
plt.loglog(sizes2, times2, '.', basex=2,basey=2, markersize=10, color='blue')
fn2 = [2**(m2 * math.log2(x) + b2) for x in sizes2]
plt.loglog(sizes2, fn2,basex=2, basey=2, label = 'Shell2~ 0.79*log(N) - 18.355', color='blue')

#shell3
plt.loglog(sizes3, times3, '.', basex=2,basey=2, markersize=10, color='green')
fn3 = [2**(m3 * math.log2(x) + b3) for x in sizes3]
plt.loglog(sizes3, fn3,basex=2, basey=2, label = 'Shell3~ 1.106*log(N) - 22.14', color='green')
```

```
#shell4
plt.loglog(sizes4, times4, '.', basex=2,basey=2, markersize=10, color='yellow')
fn4 = [2**(m4 * math.log2(x) + b4) for x in sizes4]
plt.loglog(sizes4, fn4,basex=2, basey=2, label = 'Shell4~ 0.72*log(N) - 17.44', color='yellow')

plt.xlabel('Size logN Integers')
plt.ylabel('Time log(N) Seconds')
plt.title('Shell Sort Comparison Reverse Integers')
plt.legend()
```

```
0.8121127387088943 -18.590831744751394
0.7917716246845341 -18.355043043340398
1.1061984028538605 -22.141565046119183
0.7270473959598489 -17.444248132478123
```

Out[277]: <matplotlib.legend.Legend at 0x11f5f4150>



Similiar to the previous experiment, shell sort 3 also runs considerably slower than the other 3 algorithms.

Hybrid Sorts

The Hybrid sorting algorithms work by beginning to sort the integers using Merge sort and then switching to Insertion sort at some threshold.

Hybrid Sort1: n^1/2

Hybrid Sort2: n^1/3

Hybrid Sort3: n^1/4

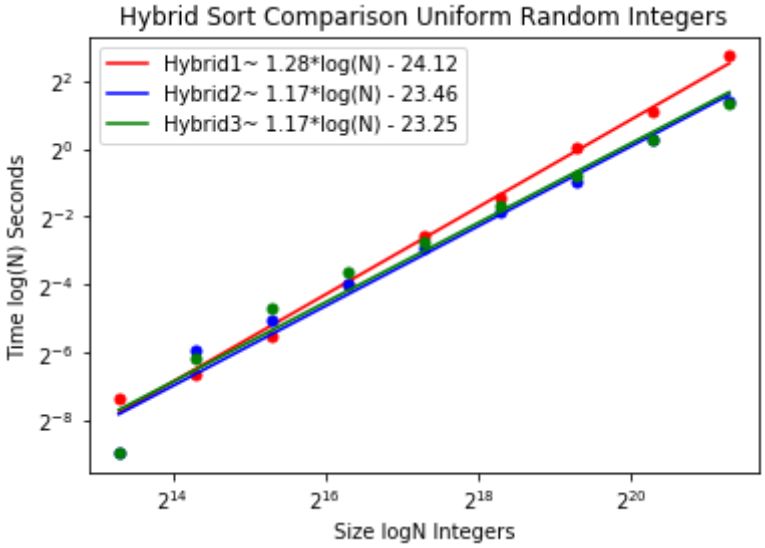
Uniformly Distributed Permutations

In [278]:
```
#hybridsort1 data
df = pd.read_csv('uniform/hybrid_sort1_uniform.csv', sep=' ')
times, sizes = getMeans(df)
logTime, logSize = getLog(times,sizes)
m, b = np.polyfit(logSize,logTime,1)
print(m,b)

#hybridsort2 data
df2 = pd.read_csv('uniform/hybrid_sort2_uniform.csv', sep=' ')
times2, sizes2 = getMeans(df2)
logTime2, logSize2 = getLog(times2,sizes2)
m2, b2 = np.polyfit(logSize2,logTime2,1)
print(m2,b2)

#hybridsort3 data
df3 = pd.read_csv('uniform/hybrid_sort3_uniform.csv', sep=' ')
times3, sizes3 = getMeans(df3)
logTime3, logSize3 = getLog(times3,sizes3)
m3, b3 = np.polyfit(logSize3,logTime3,1)
print(m3,b3)

#hybrid1
plt.loglog(sizes, times, '.', basex=2,basey=2, markersize=10, color='red')
fn = [2**(m * math.log2(x) + b) for x in sizes]
plt.loglog(sizes, fn,basex=2, basey=2, color='red', label='Hybrid1~ 1.28*log(N) - 24.12')

#hybrid2
plt.loglog(sizes2, times2, '.', basex=2,basey=2, markersize=10, color='blue')
fn2 = [2**(m2 * math.log2(x) + b2) for x in sizes2]
plt.loglog(sizes2, fn2,basex=2, basey=2, label = 'Hybrid2~ 1.17*log(N) - 23.46', color='blue')

#hybrid3
plt.loglog(sizes3, times3, '.', basex=2,basey=2, markersize=10, color='green')
fn3 = [2**(m3 * math.log2(x) + b3) for x in sizes3]
plt.loglog(sizes3, fn3,basex=2, basey=2, label = 'Hybrid3~ 1.17*log(N) - 23.25', color='green')

plt.xlabel('Size logN Integers')
plt.ylabel('Time log(N) Seconds')
plt.title('Hybrid Sort Comparison Uniform Random Integers')
plt.legend()
```

```
1.2886365632975414 -24.907574330970707
```

```
1.17687964947346 -23.46022008928897
1.1711668345846384 -23.257777167196323
```

Out[278]: <matplotlib.legend.Legend at 0x120d5aad0>



All 3 algorithms run basically the same, with hybrid1 taking slightly longer.

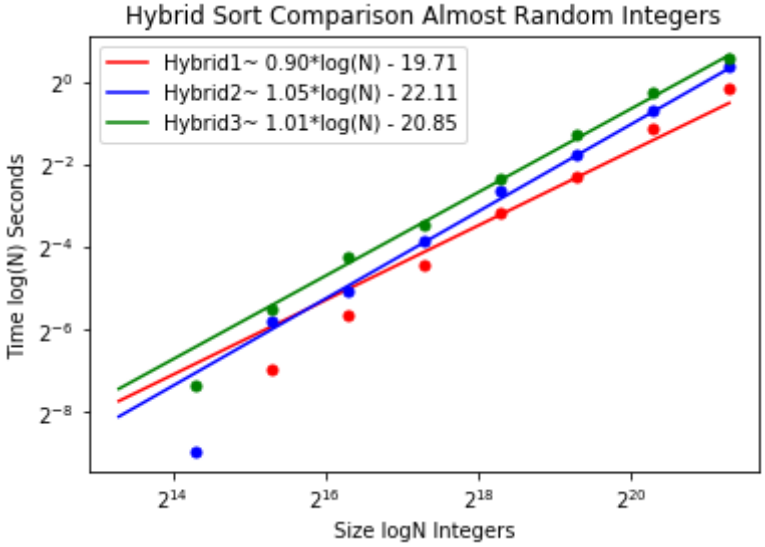
Almost-Sorted Permutations

In [280]:
```
#hybridsort1 data
df = pd.read_csv('almost/hybrid_sort1_almost.csv', sep=' ')
times, sizes = getMeans(df)
logTime, logSize = getLog(times,sizes)
m, b = np.polyfit(logSize,logTime,1)
print(m,b)

#hybridsort2 data
df2 = pd.read_csv('almost/hybrid_sort2_almost.csv', sep=' ')
times2, sizes2 = getMeans(df2)
logTime2, logSize2 = getLog(times2,sizes2)
m2, b2 = np.polyfit(logSize2,logTime2,1)
print(m2,b2)

#hybridsort3 data
df3 = pd.read_csv('almost/hybrid_sort3_almost.csv', sep=' ')
times3, sizes3 = getMeans(df3)
logTime3, logSize3 = getLog(times3,sizes3)
m3, b3 = np.polyfit(logSize3,logTime3,1)
print(m3,b3)

#hybrid1
plt.loglog(sizes, times, '.', basex=2,basey=2, markersize=10, color='red')
fn = [2**(m * math.log2(x) + b) for x in sizes]
plt.loglog(sizes, fn,basex=2, basey=2, color='red', label='Hybrid1~ 0.90*log(N) - 19.71')

#hybrid2
plt.loglog(sizes2, times2, '.', basex=2,basey=2, markersize=10, color='blue')
fn2 = [2**(m2 * math.log2(x) + b2) for x in sizes2]
plt.loglog(sizes2, fn2,basex=2, basey=2, label = 'Hybrid2~ 1.05*log(N) - 22.11', color='blue')

#hybrid3
plt.loglog(sizes3, times3, '.', basex=2,basey=2, markersize=10, color='green')
fn3 = [2**(m3 * math.log2(x) + b3) for x in sizes3]
plt.loglog(sizes3, fn3,basex=2, basey=2, label = 'Hybrid3~ 1.01*log(N) - 20.85', color='green')

plt.xlabel('Size logN Integers')
plt.ylabel('Time log(N) Seconds')
plt.title('Hybrid Sort Comparison Almost Random Integers')
plt.legend()
```

```
0.9018179040053695 -19.719771133656362
```

```
1.0537529128906957 -22.11457475920659
1.0095727630185958 -20.854877020902297
```

Out[280]: <matplotlib.legend.Legend at 0x12095ca50>



On almost random integers, the algorithms are still perform very similiarly, but hybrid1 run slightly faster.

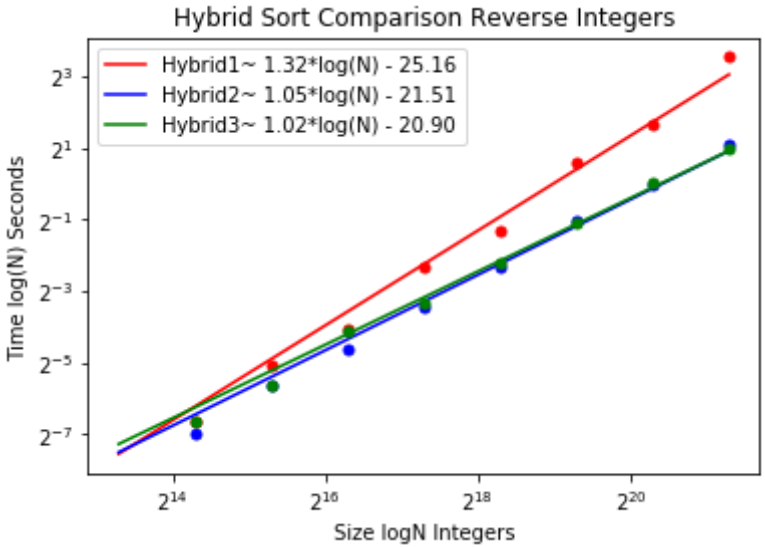
Reverse-Sorted Permutations

In [281]:
```
#hybridsort1 data
df = pd.read_csv('reverse/hybrid_sort1_reverse.csv', sep=' ')
times, sizes = getMeans(df)
logTime, logSize = getLog(times,sizes)
m, b = np.polyfit(logSize,logTime,1)
print(m,b)

#hybridsort2 data
df2 = pd.read_csv('reverse/hybrid_sort2_reverse.csv', sep=' ')
times2, sizes2 = getMeans(df2)
logTime2, logSize2 = getLog(times2,sizes2)
m2, b2 = np.polyfit(logSize2,logTime2,1)
print(m2,b2)

#hybridsort3 data
df3 = pd.read_csv('reverse/hybrid_sort3_reverse.csv', sep=' ')
times3, sizes3 = getMeans(df3)
logTime3, logSize3 = getLog(times3,sizes3)
m3, b3 = np.polyfit(logSize3,logTime3,1)
print(m3,b3)

#hybrid1
plt.loglog(sizes, times, '.', basex=2,basey=2, markersize=10, color='red')
fn = [2**(m * math.log2(x) + b) for x in sizes]
plt.loglog(sizes, fn,basex=2, basey=2, color='red', label='Hybrid1~ 1.32*log(N) - 25.16')

#hybrid2
plt.loglog(sizes2, times2, '.', basex=2,basey=2, markersize=10, color='blue')
fn2 = [2**(m2 * math.log2(x) + b2) for x in sizes2]
plt.loglog(sizes2, fn2,basex=2, basey=2, label = 'Hybrid2~ 1.05*log(N) - 21.51', color='blue')

#hybrid3
plt.loglog(sizes3, times3, '.', basex=2,basey=2, markersize=10, color='green')
fn3 = [2**(m3 * math.log2(x) + b3) for x in sizes3]
plt.loglog(sizes3, fn3,basex=2, basey=2, label = 'Hybrid3~ 1.02*log(N) - 20.90', color='green')

plt.xlabel('Size logN Integers')
plt.ylabel('Time log(N) Seconds')
plt.title('Hybrid Sort Comparison Reverse Integers')
plt.legend()
```

```
1.3253662136053619 -25.168272759444843
1.0533548546453975 -21.510832809125485
```

```
1.0248587990651752 -20.901118539315004
```

Out[281]: <matplotlib.legend.Legend at 0x12045fdd0>



On reverse data, hybrid1 runs the slowest. This makes sense because it spends the least amount of time in merge sort, which is faster when more switches need to be made.

Shell Sort Comparison

Uniform:

All the 4 shell sorting algorithms performed similiarly on these vectors with shell sort 3 performing slightly worse at a slope of 1.26. This does not seem to be a significant difference.

Almost:

On the almost random vectors, Shell sort 3 sticks out as significantly slower than the other 3, with a slope of 1.1, while the other three slopes range from 0.73-0.8. This is potentially because the Pratt sequence increases slower than the others.

Reverse:

The reverse sequence has a similiar result to the "Almost" sorted integers with Shell Sort 3, the Pratt sequence, performing slower than the other 3 by a signficant margin. This is also potentially because the Pratt sequence increases slower than the others. This makes it have to go through more iterations before it terminates.

Hybrid Comparison

Uniform:

All 3 hybrid sorting algorithms perform similiarly on these vectors with slopes ranging from 1.17 - 1.28 with hybrid1 performing slightly worse. This does not seem to be a significant difference.

Almost:

All 3 hybrid algorithms perform similiarly on these vectors with slopes ranging from 0.90-1.05 with hybrid1 performing slightly better than the others. This does not seem to be a significant difference.

Reverse:

Hybrid1 performs significantly worse than 2 and 3 with a slope of 1.32 while the others have 1.05 and 1.02 respectively. The constant N=1/2 seems to have a bigger effect on reverse sorted order. This is probably because it spends more time in insertion sort, which does the worst on reverse sorted arrays.

Different Running Times for Different Distributions

Insertion vs Merge:

Merge sort out performed insertion sort on uniformly random and reverse vectors, but insertion sort outperform merge sort on almost sorted vectors. These is because insertion sort has to perform very few steps on almost sorted vectors.

Shell Sorts:

They all performed similiarly on uniformly random vectors, but shell3 performed significantly worse on almost sorted and reverse vectors.

Hybrid Sorts:

They all performed similiarly on uniformly distributed random vectors and almost sorted vectors, but hybrid1 did signficantly worse on reverse vectors.

Which Algorithm Is Best

It is hard to say which algorithm is best because they mostly tend to perform better in some situations and worse in others. For example, merge sort generally does better than insertion sort, except on an almost sorted array. Furthermore, many algorithms in the same family perform quite similiarly, such as shell 1,2, and 4 and hybrid 2 and 3. However, in short, I would argue Shell Sort 4 is best algorithm because it performed the best on average over all 3 types of vectors with slopes of 1.02, 0.74, and 0.72 on uniformly random, almost sorted, and reverse order vectors respectively.

In [ ]: