CS260P PROJECT 1 REPORT

SHELL SORT

Contents:

Theoretical Analysis

- Definition and Explanation
- Algorithm & Analysis
- Worst-case running time

Experimental Analysis

- Types of data
- Running the test cases
- Experimental Analysis
- Comparative Analysis
- General Conclusions

THEORETICAL ANALYSIS

Shell Sort

I. Definition and Explanation

Shell Sort is an in-place comparison sort. It can be seen as either a generalization of sorting by exchange (bubble sort) or sorting by insertion (insertion sort). The method starts by sorting pairs of elements far apart from each other, then progressively reducing the gap between elements to be compared. Starting with far apart elements, it can move some out-of-place elements into position faster than a simple nearest neighbor exchange.

The running time of Shell Sort is heavily dependent on the gap sequence it uses. For many practical variants, determining their time complexity remains an open problem.

The general algorithm for Shell Sort is:

i. Generate a gap sequence based on some series or formula (this in itself is a vast research topic; different gap sequences provide different complexities for the sorting algorithm).
ii. Do an insertion sort for each gap (number) in the gap sequence. Therefore, the idea here is to start from large gaps and progress to smaller gaps.
iii. The last gap is always of size 1. This means that the last step for shell sort with any gap sequence generator is the traditional insertion sort algorithm.

In this project we experimentally analyze running times using 5 gap sequences. They are:

a) $\frac{n}{2^k}$ for k = 1, 2, _ _ _ $\log n$ (Normal sequence)
b) $2^k - 1$ for k = $\log n$ _ _ _ 7, 3 ,1 (Hibbard sequence)
c) $2^p * 3^q$ ordered from the largest such number less than n down to 1. (Pratt sequence)
d) $4^k + 3 * 2^{k-1} + 1$ in reverse order, starting from the largest value less than n, down to 1. (A036562 sequence)
e) First gap = $\frac{n}{2}$ and then all subsequent gaps as a Geometric Progression of $r = \frac{1}{2.2}$. (Custom sequence)

Therefore, we take a gap sequence one by one and do a comparative theoretical and experimental analysis.

II. Algorithm and Analysis

a) Normal Sequence

$\frac{n}{2^k}$ for k = 1,2, _ _ _ $\log n$

The general algorithm for generating the gap sequence is as follows:

i. Find the size() of input array. This can be used to find the value of $\log_2 size()$.
ii. Once we have the limit for k, we can decrement k till it's value ≥ 1.
iii. We can store all gaps for the values of k using the formula $\frac{n}{2^k}$
iv. Now, all the gaps in the gap sequence can be used for the Shell Sort algorithm.

For an input of 100 numbers for sorting the gap sequence generated in this case is:

```
[50, 25, 12, 6, 3, 1]
```

This means that the sort algorithm will first sort the numbers in gaps of 50, then 25 and so on till 1. Therefore, as stated earlier the last iteration of the algorithm is a traditional insertion sort.

This gap sequence was the invented by Shell, it is the first gap sequence for shell sort.

Worst-case running time for this gap sequence is $O(n^2)$.

This gap sequence does not theoretically does not perform very well, and due to this better gap sequences were invented which we will explore now.

b) Hibbard Sequence

$2^k - 1$ for k = $\log n$ _ _ _ 7, 3 ,1

The general algorithm for generating the gap sequence is as follows:

i. Find the size() of input array. This can be used to find the value of $\log_2 size()$.
ii. Once we have the limit for k, we can decrement k till it's value ≥ 1.
iii. We can store all gaps for the values of k using the formula $2^k - 1$.
iv. Now, all the gaps in the gap sequence can be used for the Shell Sort algorithm.

For an input of 100 numbers for sorting the gap sequence generated in this case is:

```
[31, 15, 7, 3, 1]
```

This means that the sort algorithm will first sort the numbers in gaps of 31, then 15 and so on till 1. Therefore, as stated earlier the last iteration of the algorithm is a traditional insertion sort.

This gap sequence was the invented by Hibbard, it provided a significant improvement in running time over Shell's original sequence

Worst-case running time for this gap sequence is $O(n^{1.5})$.

c) Pratt Sequence

$2^p * 3^q$ ordered from the largest such number less than n down to 1

The general algorithm for generating the gap sequence is as follows:

i. Initialize the values of p and q to 0.
ii. Then the product of $2^p * 3^q$ is calculated within the nested for loop, for the values of p and q.
iii. Whenever the product is less than the size() of the array, we add it to to gap sequence.
iv. If the product is greater than the size() of the array for some value of q, we break from the loop and test all the values of q for an incremented value of p.
v. When the algorithm breaks out of both the loops, we have all the gaps in an array.
vi. The gaps array is then sorted in the reverse order (decreasing order).
vii. Now, all the gaps in the gap sequence can be used for the Shell Sort algorithm.

As we can see, the generation of the gap sequence in Pratt's algorithm takes $O(n^2)$ time in the worst-case.

For an input of 100 numbers for sorting the gap sequence generated in this case is:

```
[96, 81, 72, 64, 54, 48, 36, 32, 27, 24, 18, 16, 12, 9, 8, 6, 4, 3, 2, 1]
```

This means that the sort algorithm will first sort the numbers in gaps of 96, then 81 and so on till 1. Therefore, as stated earlier the last iteration of the algorithm is a traditional insertion sort.

We observe that the number of gaps generated in Pratt's sequence is significantly higher than the previous sequences.

Worst-case running time for this gap sequence is $O(nlog^2 n)$.

d) A036562 Sequence

$4^k + 3 * 2^{k-1} + 1$ in reverse order, from the largest value less than n, down to 1.

The general algorithm for generating the gap sequence is as follows:

i. Add the value of 1 to the gap sequence.
ii. From k=1 calculate the gap using the formula $4^k + 3 * 2^{k-1} + 1$.
iii. Break from the loop when the value increases the size() of the array.
iv. Sort the array of gap sequences in reverse order (descending order).
v. Now, all the gaps in the gap sequence can be used for the Shell Sort algorithm.

For an input of 100 numbers for sorting the gap sequence generated in this case is:

```
[77, 23, 8, 1]
```

This means that the sort algorithm will first sort the numbers in gaps of 77, then 23 and so on till 1. Therefore, as stated earlier the last iteration of the algorithm is a traditional insertion sort.

The number of gaps generated in this algorithm will be lower than the other algorithms in this project.

This gap sequence was the invented by Sedgewick, it provided a a very good improvement in running time over Shell's original sequence.

Worst-case running time for this gap sequence is $O(n^{1.33})$.

e) Custom Sequence

First gap = $\frac{n}{2}$ and then all subsequent gaps as a Geometric Progression of $r = \frac{1}{2.2}$

The general algorithm for generating the gap sequence is as follows:

i. Using the formula given above generate gaps for the given input size.
ii. Now, all the gaps in the gap sequence can be used for the Shell Sort algorithm.

For an input of 100 numbers for sorting the gap sequence generated in this case is:

```
[50, 22, 10, 4, 2]
```

The idea behind selecting this gap sequence is that, the best gap sequence for Shell Sort proven experimentally is a sequence which is in the form of a Geometric Progression which starts with the ratio $r = 2.5$ and slowly the value of r decreases and reaches 2.25.

I have taken some motivation from this approach where the first gap is $\frac{n}{2}$ and all subsequent gaps are a Geometric Progression with the ratio $r = 2.2$.

The running time complexity of this sequence will be judged by doing a comparative experimental analysis on various inputs.

III. Worst-case running time

We will now create a table of all the theoretical complexities of the four gap sequences.

Gap Sequence	**Worst-case Theoretical Time Complexity**
Normal	$O(n^2)$
Hibbard	$O(n^{1.5})$
Pratt	$O(nlog^2 n)$
A036562	$O(n^{1.33})$

Now, we will move on to the experimental analysis of the sequences.

EXPERIMENTAL ANALYSIS

I. Types of data

There are two types of data on which the running times of the sequences are tested, they are:

a. Uniformly distributed data
b. Almost sorted data

Uniformly distributed data:

This data is true random numbers generated from www.random.org/

The maximum limit for the number of random numbers that can be generated is 10,000, so for databases of 100,000 and 1,000,000 I used Python's *random.randinit()* function

Therefore, this data was uniformly distributed and serves as a good analytical tool.

Almost sorted data

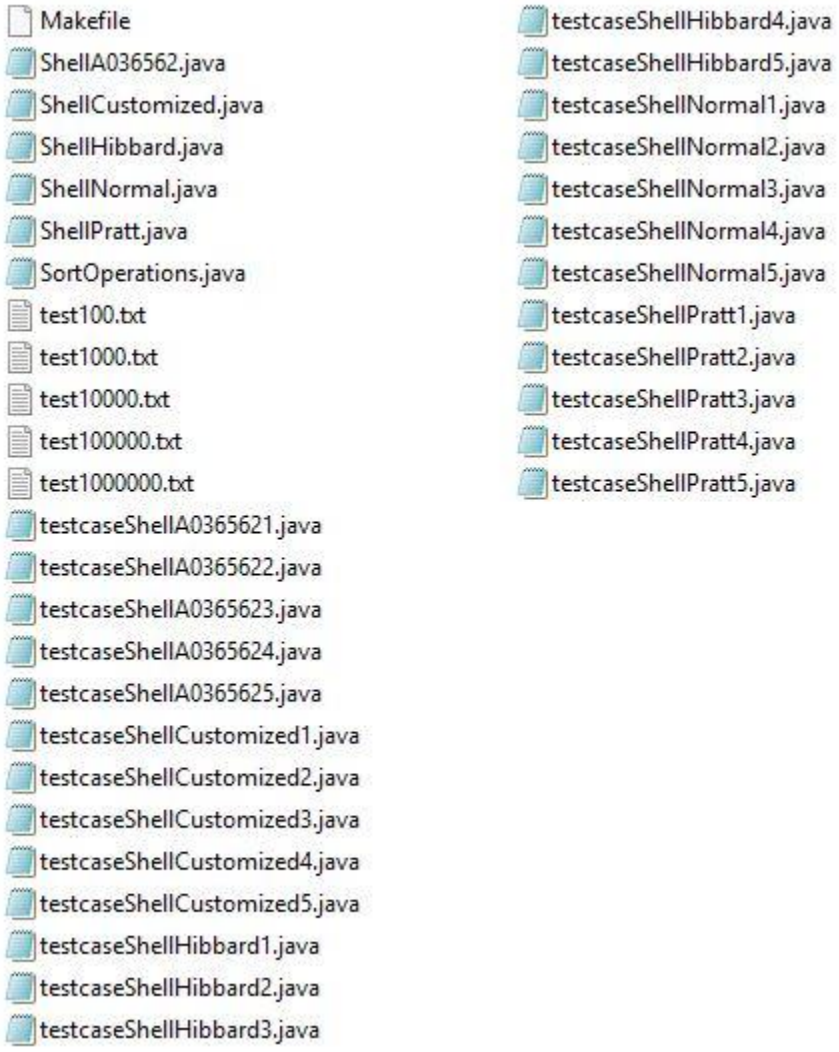
This data was generated by starting with a sorted array of n numbers, say, the numbers 1,2,3,…,n, in this order. Then, independently choose 2log n pairs, (i,j), where i and j are uniformly-chosen random integers in the range from 0 to n-1, and swap the numbers at positions i and j in the array.

Almost sorted data also serves as a good analytical tool.

So, there are two types of data on which experimental analysis is performed so that the behavior of the sequences can be tested properly, and we can find their varied behavior.

II. Running the testcases

The files in the code are as follows:



To compile all the files, the *make* command can be used on the terminal.

The *.txt* files contain the uniformly distributed data. The files are named as *test<input_size>.txt*

The testcases are named as *testcase<gap_sequence><testcase_number>*

Ex: test100.txt is uniformly distributed data of size 100.

Ex: testcaseShellPratt1.java is testcase 1 for Shell Sort using Pratt gap sequence.

The testcases are designed as follows:

i. Testcase 1 : This testcase computes the running time for sorting of 100 numbers (both uniformly distributed and almost sorted.
ii. Testcase 2 : This testcase computes the running time for sorting of 1000 numbers (both uniformly distributed and almost sorted.

iii. Testcase 3 : This testcase computes the running time for sorting of 10000 numbers (both uniformly distributed and almost sorted.

iv. Testcase 4 : This testcase computes the running time for sorting of 100000 numbers (both uniformly distributed and almost sorted.

v. Testcase 5 : This testcase computes the running time for sorting of 1000000 numbers (both uniformly distributed and almost sorted.

A sample of testcase 1 output for shell sort using Normal gap sequence is given below:

```
<terminated> testcaseShellNormal1 [Java Application] C:\Program Files\Java\jre1.8.0_20
NORMAL SHELL SORT

running on uniformly distributed input of 100 numbers
Time: 1392 microseconds
Is sorted array correct? Yes

running on almost sorted input of 100 numbers
Time: 432 microseconds
Is sorted array correct? Yes
```

An abstract class (SortOperations.java) is created which implements the same functions for various gap sequences. It is given below:

```
import java.util.*;

public abstract class SortOperations {

    public abstract ArrayList<Integer> readFile(String filename);
    public abstract ArrayList<Integer> generateGap(ArrayList<Integer> data);
    public abstract ArrayList<Integer> sortFunction(ArrayList<Integer> data, ArrayList<Integer> gaps);
    public abstract void checkCorrect(ArrayList<Integer> sort);
    public abstract ArrayList<Integer> generateAlmostSorted(int n);
}
```

readFile() – reads uniformly distributed data from a file

generateGap() – Generates the gap sequence according to the algorithm

sortFunction() – performs Shell Sort

checkCorrect() – checks correctness of the sorted array

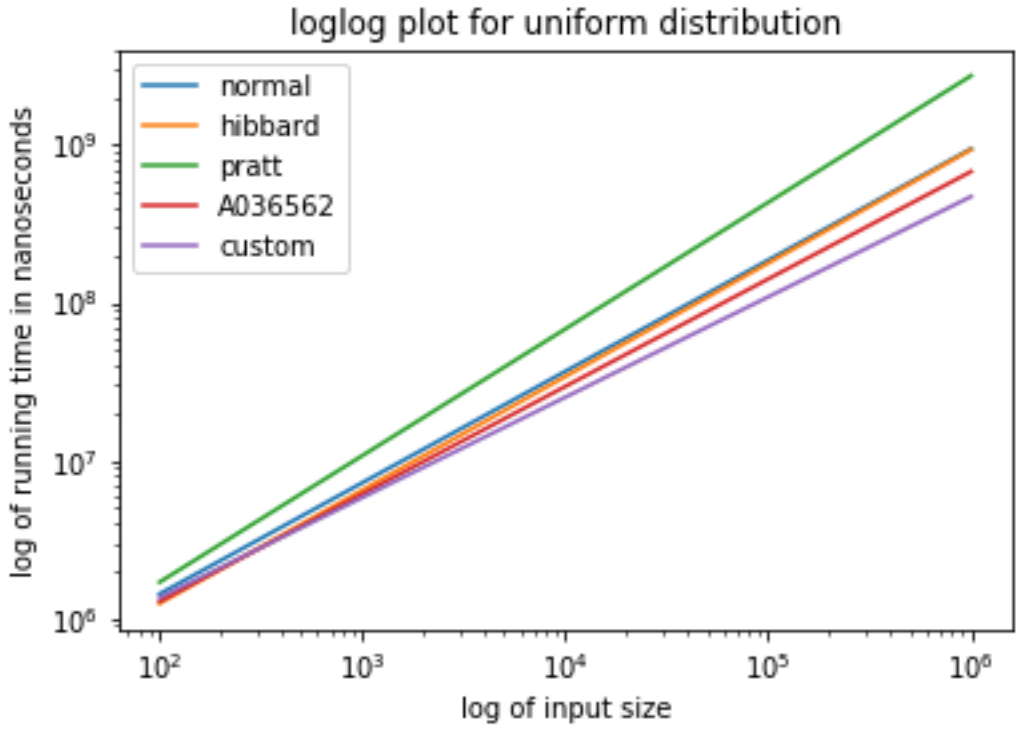
generateAlmostSorted() - Generates an almost sorted sequence for a given size

To calculate the running time, *System.nanoTime()* function is used.

To run a testcase (after compilation) – *java testcaseShellNormal1*

III. Experimental Analysis

i. Graph 1: Comparison of all sequences for uniformly distributed data



The graph shown above is a loglog plot for all the sequences run on uniformly distributed data.

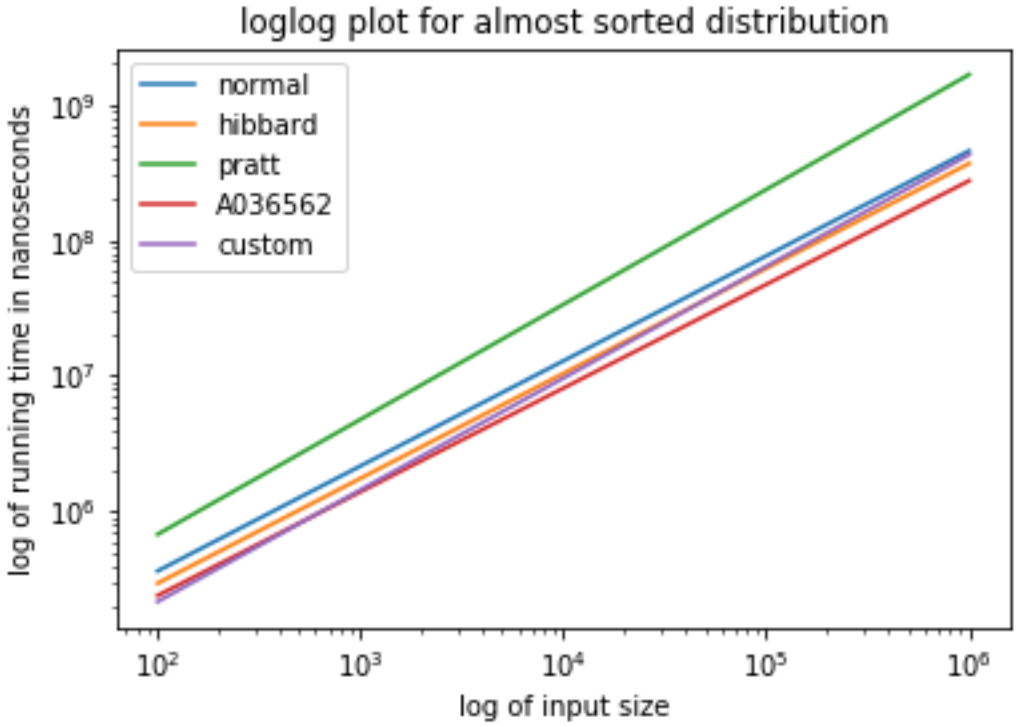
We observe from the graph that for uniformly distributed data and a maximum input size of 1,000,000, the Custom gap sequence performs the best.

The A036562 gap sequence is also good when it comes to sorting large amounts of data.

There is very little to separate between the Normal gap sequence and Hibbard gap sequence.

Surprisingly, the Pratt gap sequence performs very poorly, this can be due to the fact that the number of gaps in the Pratt gap sequence is far more than any other sequence, or the fact that Java's *system.nanoTime()* is not a good measure for calculating the time in this case.

ii. Graph 2: Comparison of all sequences for Almost sorted data



The graph shown above is a loglog plot for all the sequences run on almost sorted data.

We observe from the graph that for almost sorted data and a maximum input size of 1,000,000, the A036562 gap sequence performs the best.

The Custom gap sequence performs well in the beginning, but for inputs of higher size the sequence performs poorly.

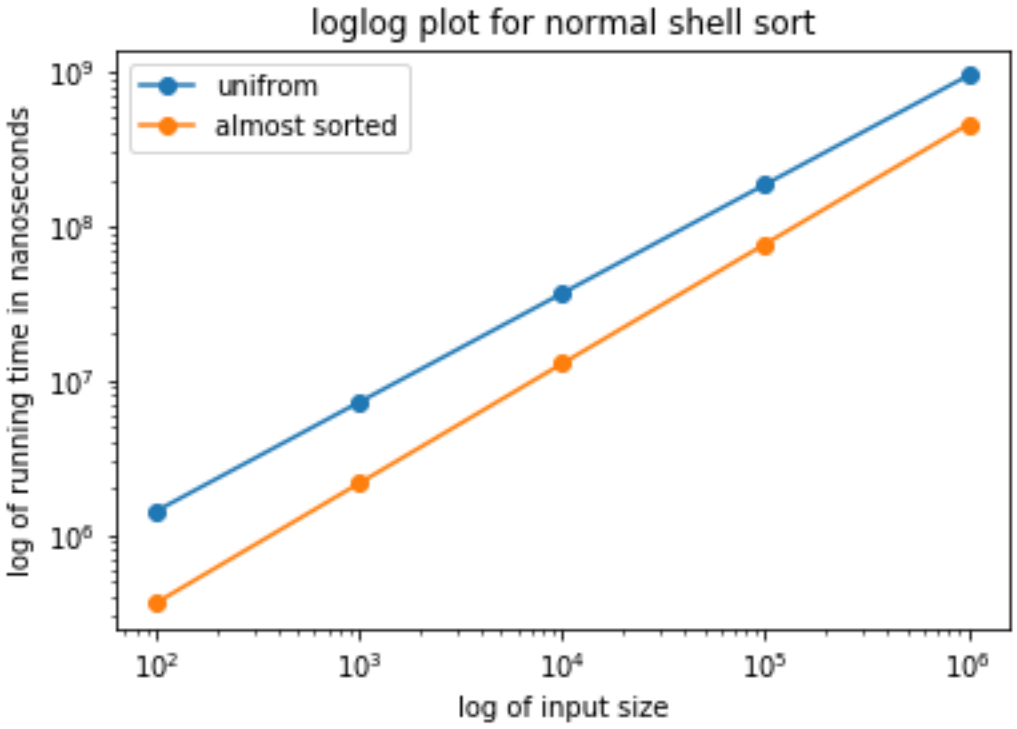
The Hibbard gap sequence performs better than the Normal gap sequence in the case of almost sorted data.

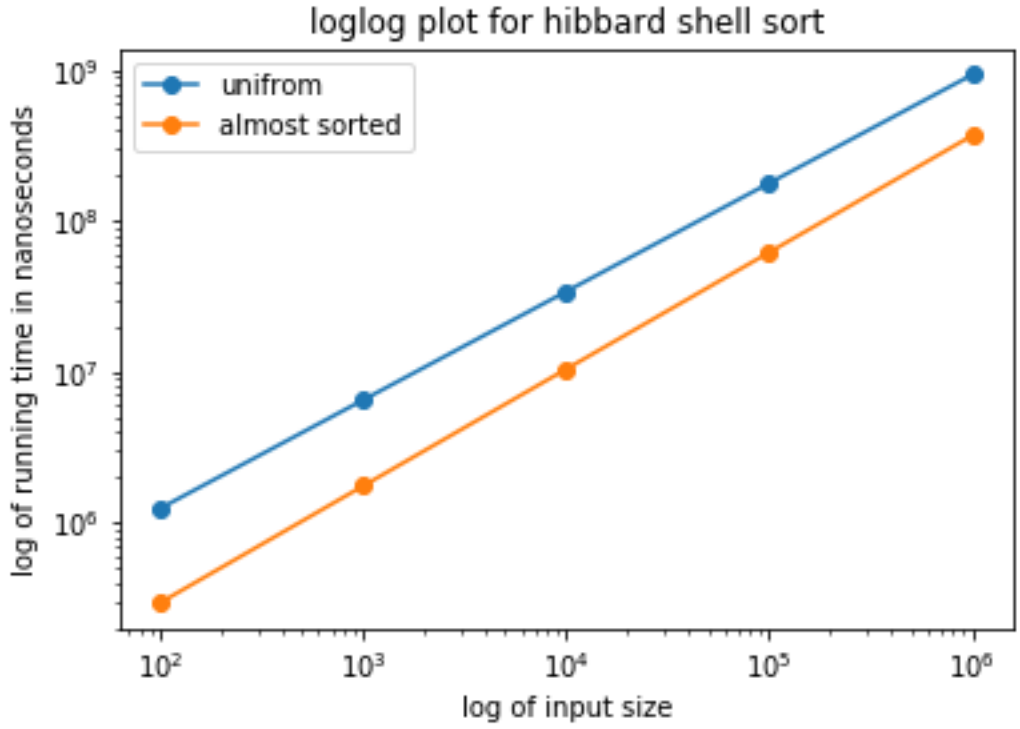
Surprisingly, the Pratt gap sequence performs very poorly, this can be due to the same facts stated above, ie: the number of gaps is high in the Pratt sequence or Java's *system.nanoTime()* is not a good measure for calculating the time in this case.

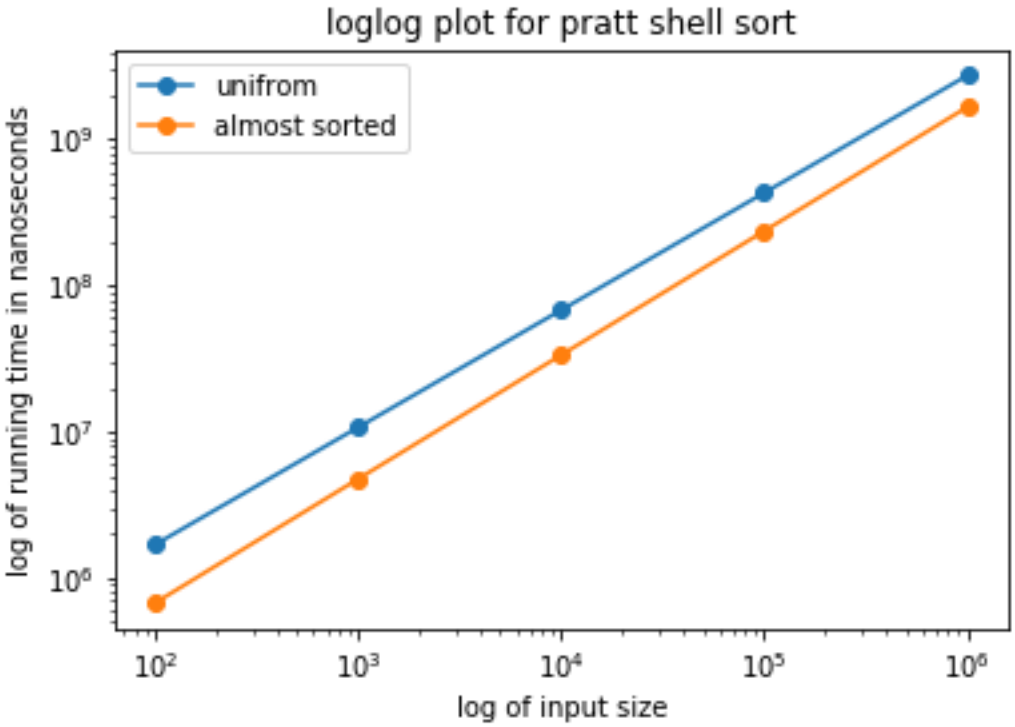
Again, we get the same result for Pratt sequence, this can mostly be attributed to the same reasons as mentioned above.

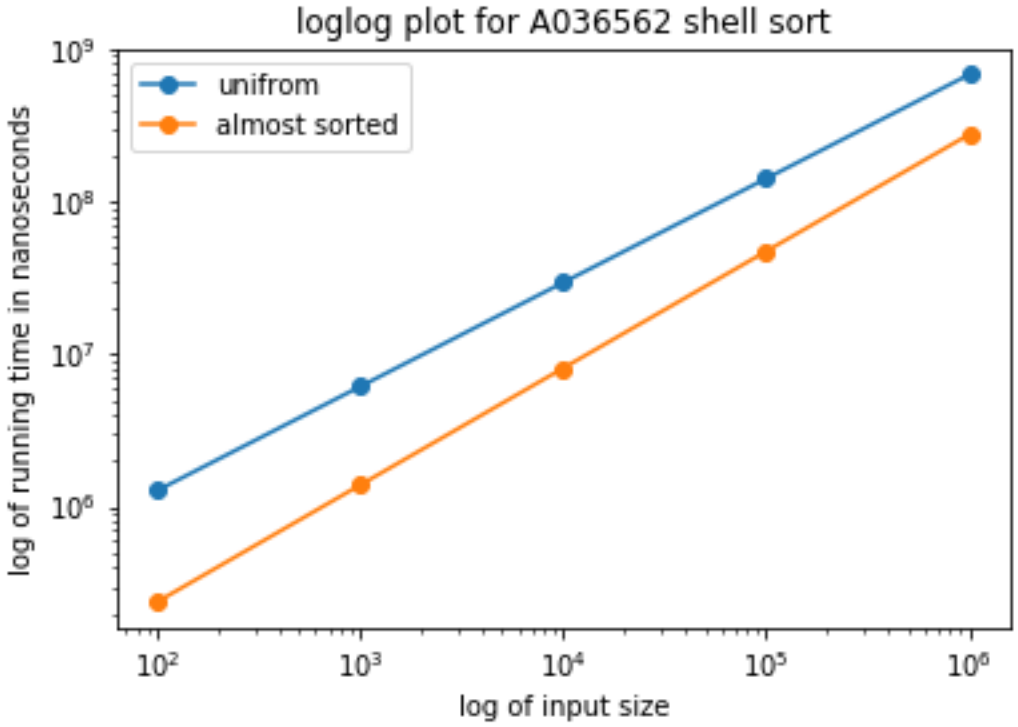
Because we have taken two different types of data, we can observe the differences in running times of A036562 and Custom sequence.

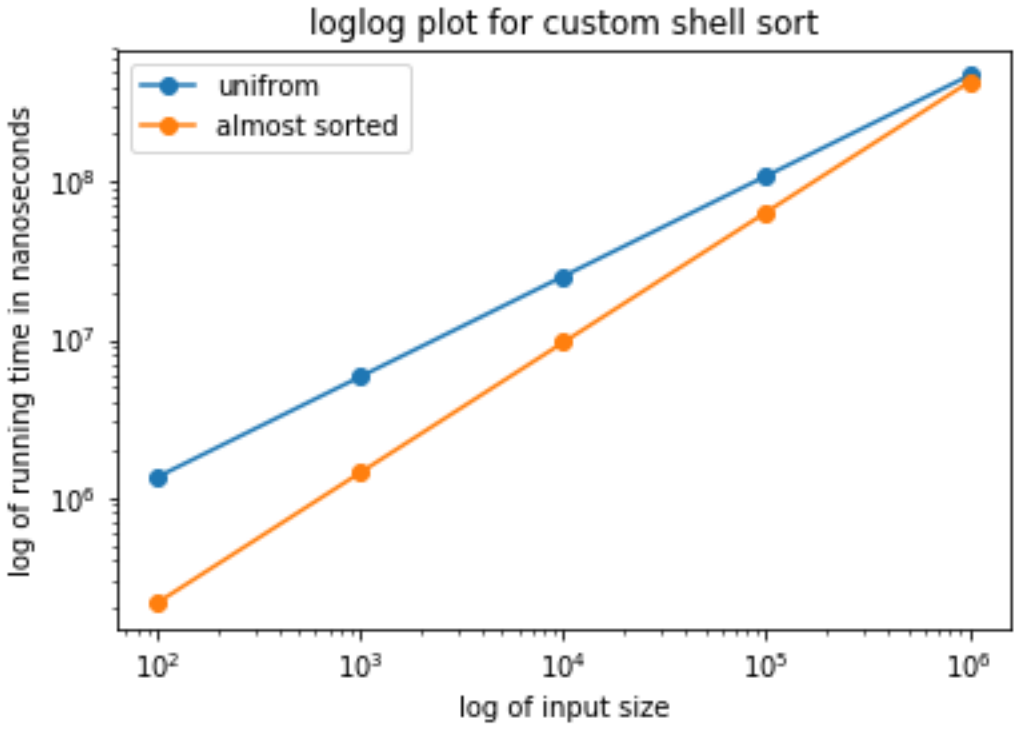
iii. Series of graphs comparing the running times for uniformly distributed and almost sorted data for each algorithm






loglog plot for pratt shell sort
unifrom
almost sorted
log of running time in nanoseconds
log of input size
10^6
10^7
10^8
10^9
10^2
10^3
10^4
10^5
10^6


loglog plot for A036562 shell sort
unifrom
almost sorted
log of running time in nanoseconds
log of input size
10^6
10^7
10^8
10^9
10^2
10^3
10^4
10^5
10^6



From the series of graphs above, we can come to the conclusion that the sorting algorithm almost always performs better in the case of almost sorted data.

One peculiar thing that we can observe in the case of Custom gap sequence is that for a large input size of 1,000,000 we the algorithm performs the same for uniform distribution and almost sorted sequence.

These graphs were plotted by taking the line estimate of the points on the loglog scale. The line was estimated using Python's *numpy.polyfit()* function.

To experimentally derive the complexity of the algorithms, we can calculate the slope of the lines on the loglog scale and n to the power of the slope.

What was observed was that the slope in itself does not totally conform to the theoretical running times.

This can be due to Java's *system.nanoTime()* which is not a good measure for calculating the time sometimes.

For the slope calculation only high input points have been considered as the low input points do not contribute much to the complexity.

IV. Comparative Analysis

i. Uniformly distributed data

From the graphs the derived complexity is as follows (from highest to lowest)

Gap Sequence	Slope	Complexity
Pratt	1.19	$O(n^{1.19})$
Normal	1.02	$O(n^{1.02})$
Hibbard	1.02	$O(n^{1.02})$
A036562	0.92	$O(n^{0.92})$
Custom	0.88	$O(n^{0.88})$

ii. Almost sorted data

From the graphs the derived complexity is as follows (from highest to lowest)

Gap Sequence	Slope	Complexity
Pratt	1.16	$O(n^{1.16})$
Normal	1.09	$O(n^{1.09})$
Custom	1.04	$O(n^{1.04})$
Hibbard	0.95	$O(n^{0.95})$
A036562	0.92	$O(n^{0.92})$

V. General Conclusions

i. The experimental complexities differ from the theoretical complexities. Some better measures of time have to be used in order to get a value close to the actual theoretical complexity.
ii. In general the A036562 sequence works very well for both uniformly distributed and almost sorted data, both theoretically and experimentally.
iii. The custom sequence which uses a Geometric Progression of $r = 2.2$ works well for small input size.
iv. Hibbard and Normal sequences are generally run in the same complexity on large input data
v. In general, Shell Sort gap sequences are still an active research topic and researchers are searching for a good gap sequence.