Performance analysis of iterative and recursive Shellsort algorithms utilizing Shell and Hibbard increment sequences

Mathew Yamasaki

Design and Analysis of Algorithms (CMSC 451), University of Maryland University College

INTRODUCTION

The Shellsort algorithm, introduced by Donald Shell (Shell, 1959), performs in-place insertion sorting on progressively smaller subsets of a data set. Shellsort has a performance advantage over insertion sort by allowing exchanges of array elements that are far apart. Insertion sort only performs exchanges on adjacent elements. The sizes of the subsets are determined by a gap sequence equation. Since Shellsort's introduction, many computer scientists have attempted to develop a increment sequence with a time-complexity better than $\Theta\left(N^2\right)$given by Shell's original gap sequence of $\left\lfloor N / 2^k \right\rfloor$. For a comparative analysis of Shellsort's performance, Shell's iterative increment sequence is compared with that of Hibbard's recursive increment sequence, $2^k - 1$ (Hibbard, 1963). It is speculated that the iterative implementation will perform better than the recursive implementation on smaller data sets.

DEVELOPMENT AND TESTING ENVIRONMENT

Hardware and Software Used

The program was created and tested using the NetBeans integrated development environment (IDE) version 7.1.2 running on Microsoft Windows 7 Home Premium with Service Pack 1. The system processor is a 2.00GHz Intel Core i7-2630QM.

Critical Operation Selection

The performance of Shellsort depends on the sort increment sequence used. Therefore, the array element insertions at the defined increment sequences was chosen as the critical operation to evaluate the performance of both the iterative and recursive versions.

THE SHELLSORT ALGORITHM

Shellsort is a multi-pass algorithm with each pass resulting in an insertion sort of the increment sequences every n^{th} element for a fixed gap *n*. The basic structure follows:

Recursive Algorithm

The recursive version consists of rearranging the array to give it the property that every n^{th} element (starting anywhere) yields a sorted array. Such an array is said to be *n*-sorted. As the sort executes, *n* decreases with each recursive call until it becomes 1 in the last iteration where the algorithm is equivalent to an insertion sort.

Iterative Algorithm

Using the iterative shell sorting algorithm, it is not possible to sort the items in one pass. Therefore, on each pass, a fixed increment is used between elements whose value decreases after each iteration.

PROGRAM RESULTS

The results of running the Shellsort program are shown in Figure 2. This data was used to perform performance comparisons.

Data Set Size n	Iterative				Recursive			
	Average Critical Operation Count	Standard Deviation of Count	Average Execution Time	Standard Deviation of Time	Average Critical Operation Count	Standard Deviation of Count	Average Execution Time	Standard Deviation of Time
16	74.9600	9.7791	0.0040	0.0014	68.6200	9.8828	0.0044	0.0018
32	254.4600	26.1754	0.0110	0.0018	231.4400	24.9540	0.0117	0.0030
64	905.5000	76.8028	0.0360	0.0029	854.7600	72.8332	0.0383	0.0104
128	3483.6600	226.8230	0.0079	0.0004	3278.6000	206.0550	0.0075	0.0005
256	13143.1800	629.8152	0.0250	0.0021	12610.4200	637.8568	0.0251	0.0012
512	51991.0600	1685.4785	0.0884	0.0039	48946.4000	1939.9351	0.0941	0.0531
1024	205940.4000	4173.5495	0.4137	0.0955	195863.4600	4802.9846	0.3163	0.0196
2048	819514.6000	11698.0924	2.1712	0.0821	776011.5000	13108.4540	1.2065	0.0558
4096	3266887.6800	35399.6373	8.6501	0.2413	3096469.9600	44447.1504	4.6900	0.1702
8192	13048046.0800	116688.1087	34.3180	0.4746	12373738.3200	110135.3014	18.5291	0.3049

Figure 1. Screenshot of NetBeans output after running the `Shellsort` program.

Figures 2 and 3 show graphical comparisons of average critical counts and execution times.

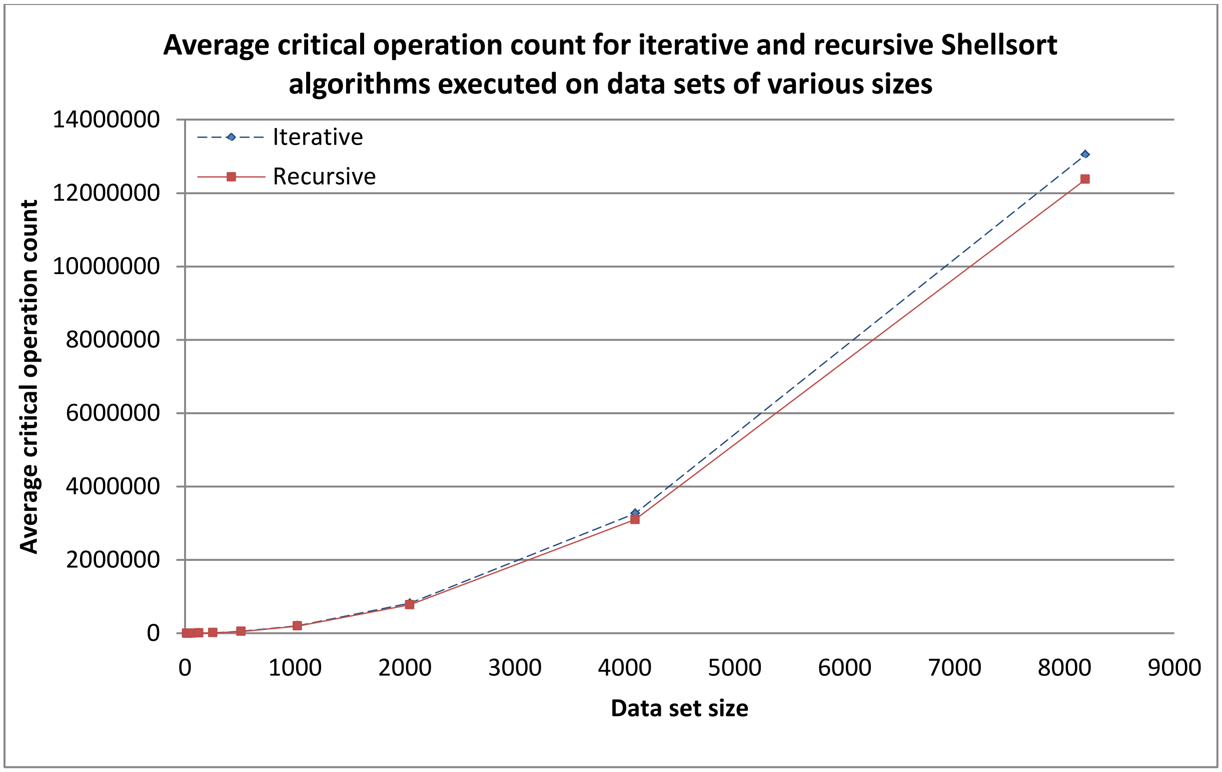

Figure 2. A comparison of the average critical operation counts.

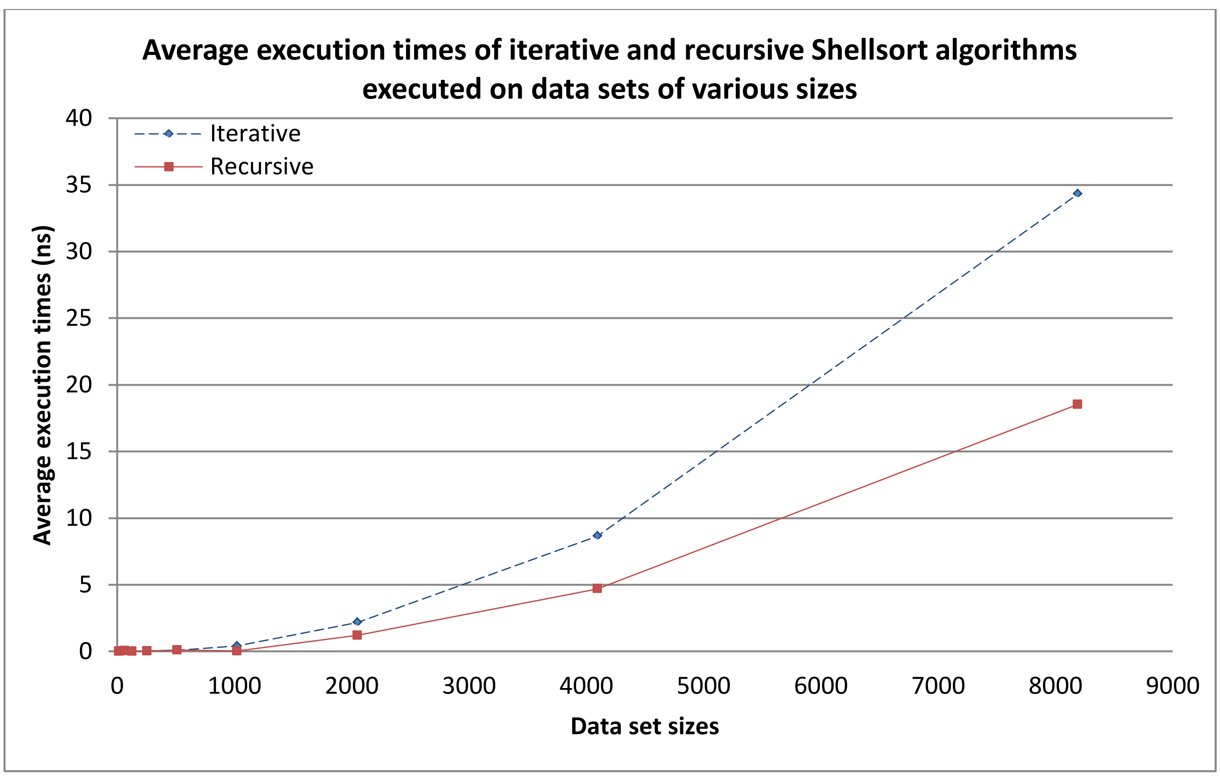

Figure 3. A comparison of average execution times.

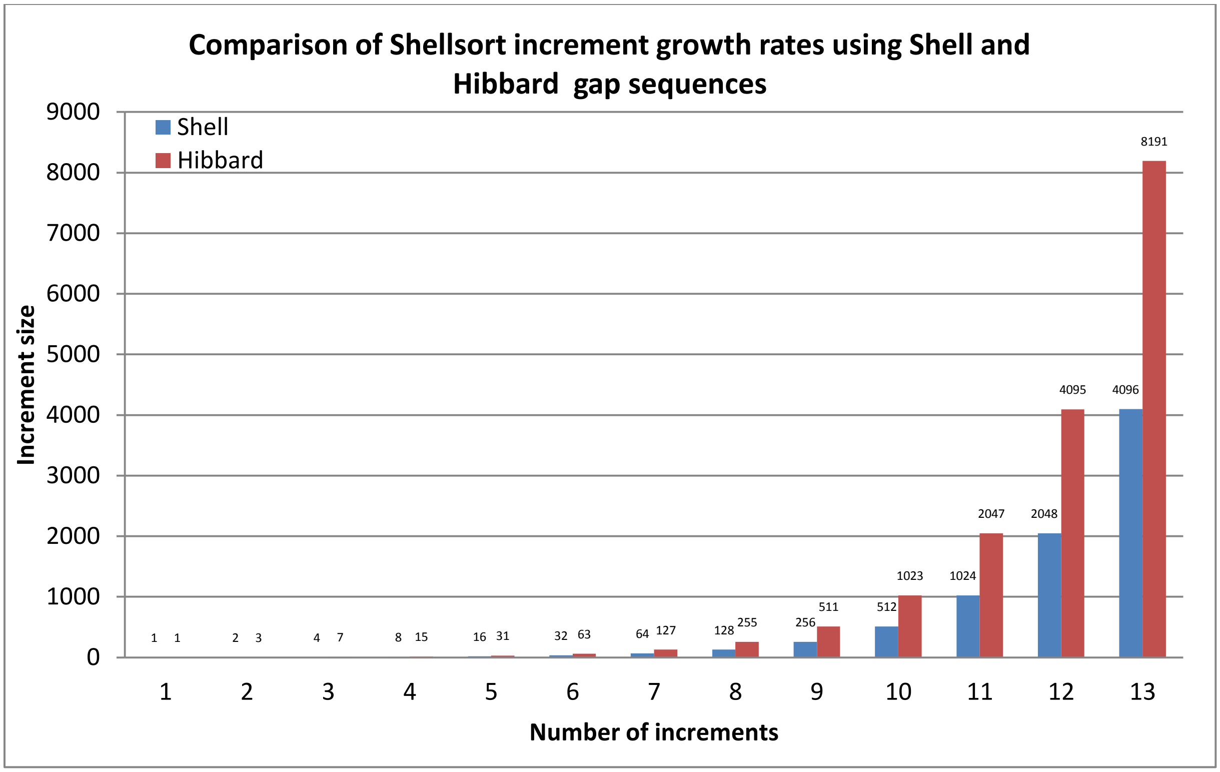

Figure 4. Graph shows that the Hibbard increment sequence sorts elements that are twice as far apart compared to Shell.

COMPLEXITY ANALYSIS

Iterative Algorithm

```
static class IterativeShellSort implements ShellSortI {
        public String getName() {
            return "Iterative Shell Sort";
        }
        public long sort(int[] arr) {
            long cmp_count = 0;
            for (int gap = arr.length/2; gap > 0; gap /= 2) {
                // insertion sort with given gap
                for (int i = gap; i < arr.length; i += gap) {
                    int item = arr[i];
                    int slot = i;
                    while (slot >= gap && arr[slot-gap] > item) {
                        ++cmp_count;
                        arr[slot] = arr[slot-gap];
                        slot -= gap;
                    }
                    ++cmp_count;
                    arr[slot] = item;
                }
            }
            return cmp_count;
        }
    }
```

Figure 5. The iterative Shellsort algorithm using Shell's increment sequence.

On line 13, the `while` loop begins with `slot` at `i` and decreases `slot` by `gap` each time around until `slot` has been reduced to `gap`. This loop repeats $\frac{i - \text{gap}}{\text{gap}}$ times. Therefore, the loop can be replaced by

$$\frac{i - \text{gap}}{\text{gap}}\Theta(1) = \Theta\left(\frac{i - \text{gap}}{\text{gap}}\right) = \Theta\left(\frac{i}{gap} - \frac{gap}{gap}\right) = \Theta\left(\frac{i}{gap} - 1\right) = \Theta\left(\frac{i}{gap}\right). \quad (1)$$

Since lines 14, 15 and 16 are $O(1)$, the entire loop body is $\Theta\left(\frac{i}{gap}\right)$.

On line 5, the `for` loop repeats `arr.length - gap` times. Since the value of `i` changes with each iteration, a summation of all the iterations is required:

$$\Theta\left(\sum_{i=gap}^{n-1} \frac{i}{gap}\right) = \Theta\left(\frac{1}{gap} \sum_{i=gap}^{n-1} i\right). \quad (2)$$

`When` $i = 1$:

$$\Theta\left(\frac{1}{gap}\sum_{i=1}^{n-1} i\right) = \Theta\left(\frac{1}{gap} O\left(n^2\right)\right) = \Theta\left(\frac{n^2}{gap}\right). \quad (3)$$

On line 10, `gap` will always become values that are powers of 2. Since `gap` is reduced by ½ after each iteration, it is factored into a summation:

$$\Theta\left(\sum_{i=0}^{\log(n-1)} \frac{n^2}{2^i}\right) = \Theta\left(n^2 \sum_{i=0}^{\log(n-1)} \frac{1}{2^i}\right). \quad (4)$$

Recursive Algorithm

```
1   static class RecursiveShellSort implements ShellSortI {
2           public String getName() {
3               return "Recursive Shell Sort";
4           }
5           /** Wrapper around sort_helper */
6           public long sort(int[] arr) {
7               return sort_helper(arr, 1);
9           }
10          /** Recursive function where sorting is actually done */
11          public long sort_helper(int[] arr, int gap) {
12              if (gap > arr.length) return 0;
13              // recurse
14              long cmp_count = sort_helper(arr, (gap+1)*2 - 1);
15              // insertion sort with given gap
16              for (int i = gap; i < arr.length; i += gap) {
17                  int item = arr[i];
18                  int slot = i;
19                  while (slot >= gap && arr[slot-gap] > item) {
20                      ++cmp_count;
21                      arr[slot] = arr[slot-gap];
22                      slot -= gap;
23                  }
24                  ++cmp_count;
25                   arr[slot] = item;
26              }
27              return cmp_count;
28          }
29      }
```

Figure 6. Recursive Shellsort algorithm using Hibbard's increment sequence.

The shaded area in Figure 6 shows the code equivalent with the iterative algorithm shown in Figure 5. Recursion occurs on line 14 and continues until `gap` becomes less than `arr.length`.

For an increment h_k, there are h_k insertion sorts of N/h_k elements. The complexity of a single pass can be expressed by $O\left(h_k\left(N/h_k\right)^2\right)$. The summation over all passes can be expressed by $O\left(\sum_{i=1}^{t} N^2/h_i\right)$, which is $O\left(N^2\right)$.

For $h_{t/2} = \Theta\left(\sqrt{N}\right)$:

$$O\left(\sum_{k=1}^{t/2} Nh_k + \sum_{k=t/2+1}^{t} N^2 / h_k\right) = O\left(N\sum_{k=1}^{t/2} h_k + N^2 \sum_{k=t/2+1}^{t} 1/h_k\right)$$
$$O\left(Nh_{t/2}\right) + O\left(\frac{N^2}{h_{t/2}}\right) = O\left(N^{3/2}\right) \tag{5}$$

SENSITIVITY ANALYSIS

Standard Deviation of Critical Operation Count

From comparing Figures 10 and 11, the recursive algorithm has the appearance of having an overall lower standard deviation that the iterative version. However, this only occurs for data set sizes of 32, 64, 128, 2048, and 8192. This is attributed to the algorithm performing significantly better with larger data set sizes.

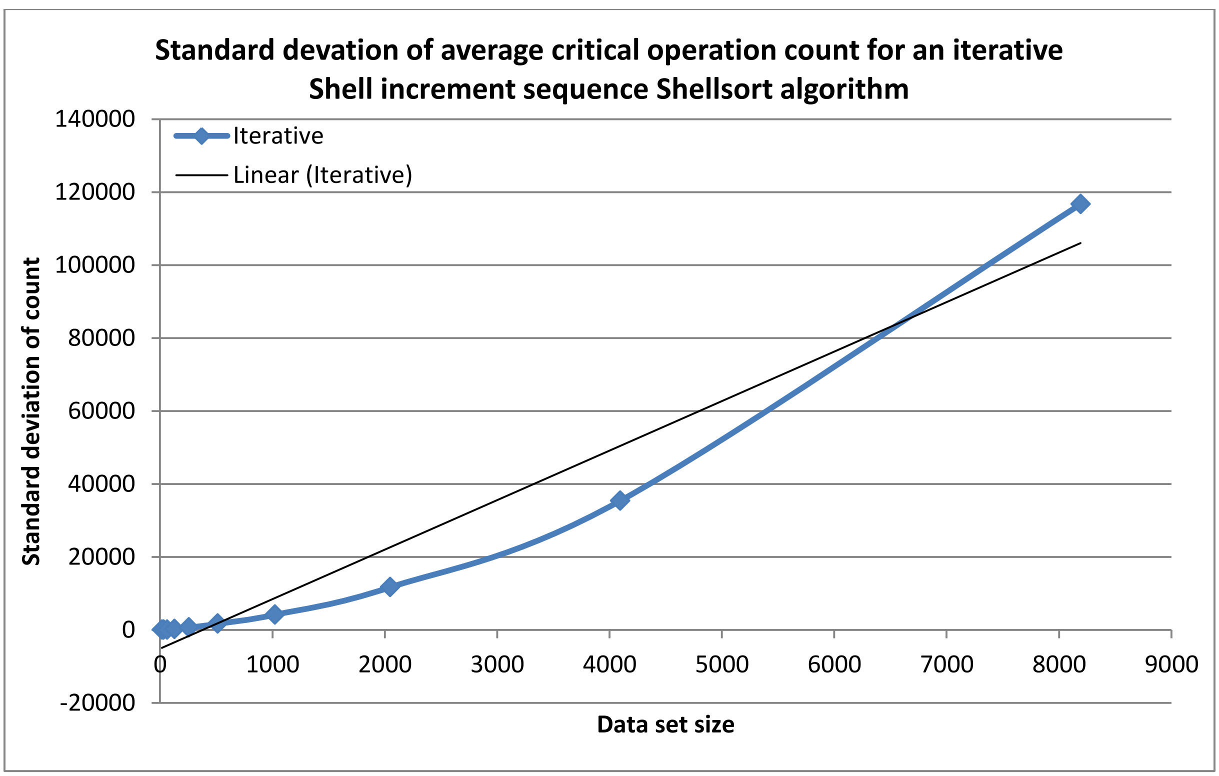

Figure 7. Standard deviation of iterative algorithm with linear regression plot.

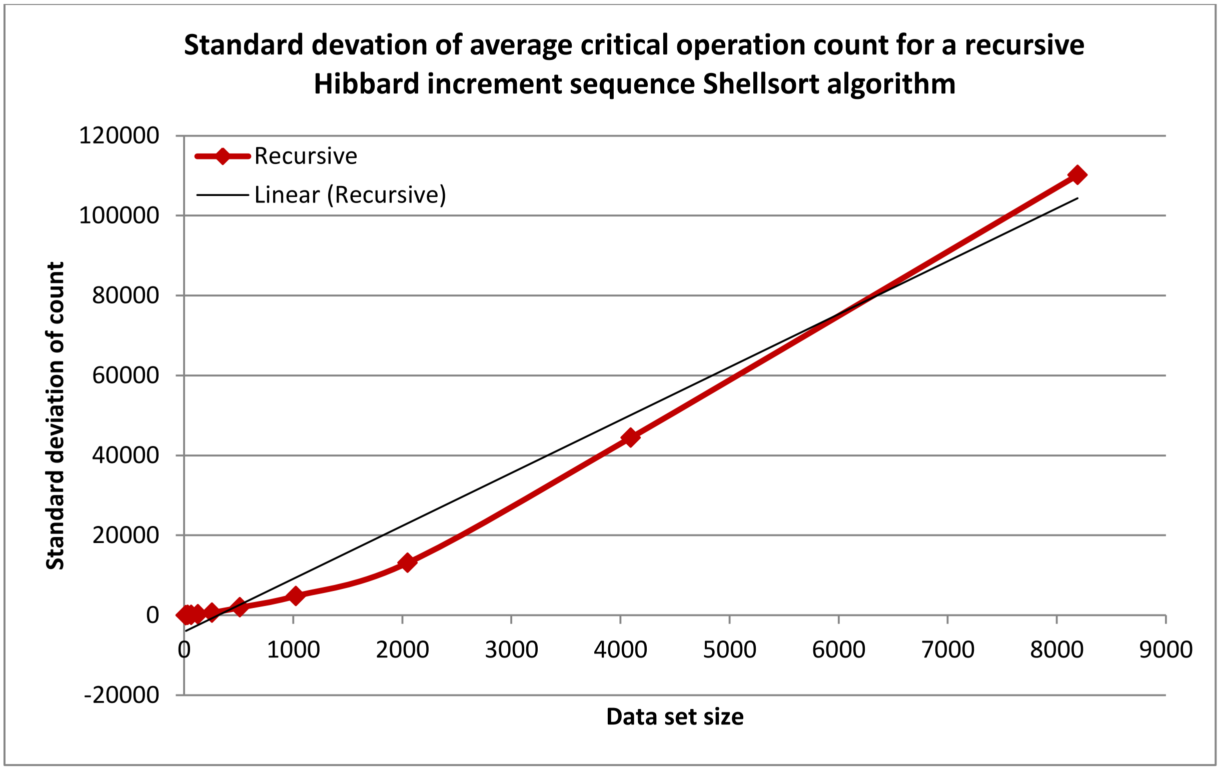

Figure 8. Standard deviation of recursive algorithm with linear regression plot.

Standard Deviation of Execution Time

Here, the recursive algorithm showed a greater deviation from the mean compared to the iterative version. At a data set size of 512 elements, recursion showed a large increase in execution time (Figure 12). For data set sizes of 1,024 elements and larger, recursion had significantly faster execution times. For data set sizes of 512 elements and smaller, recursion had a minimal advantage over iteration.

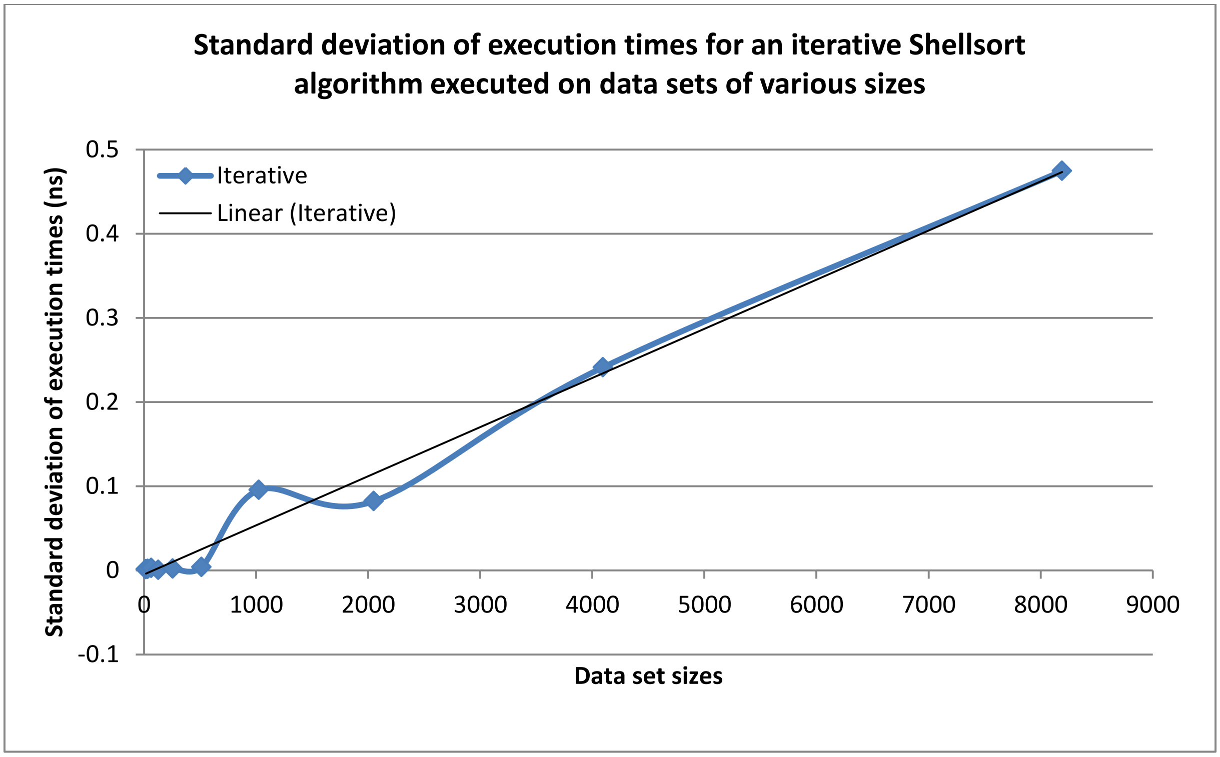

Figure 9. Iterative Shellsort standard deviation of execution times.

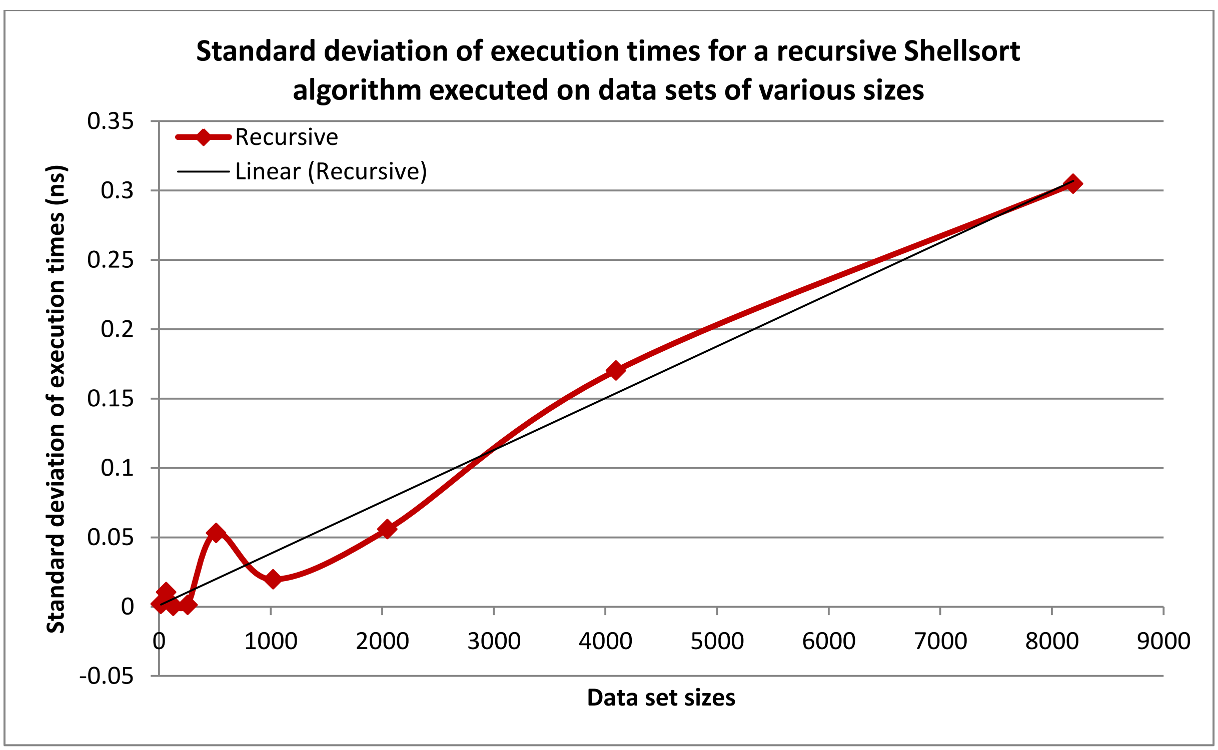

Figure 10. Recursive Shellsort standard deviation of execution times.

Algorithm Data Sensitivity

Up until a data set size of 64 elements, the recursive algorithm performs less critical operations with longer executions times compared to the iterative algorithm. For data set sizes larger than 64 elements, the recursive version performs less critical operations in less time.

CONCLUSIONS

The complexity analysis appears to match the results achieved.

For smaller data set sizes, the iterative algorithm with Shell increment sequences may be the better choice. This may depend on the type and complexity of the data being sorted. The recursive Hibbard increment sequence algorithm performed better for larger data sets. If testing was conducted on data sets larger than 8,192 elements, it is speculated that performance would greatly exceed that of the iterative version.

REFERENCES

Hibbard, T.N. (1963). A simple sorting algorithm. *Communications of the ACM, 10*(2), 142-150. Retrieved from the ACM Digital Library.

Oracle Corporation (n.d.). NetBeans (version 7.1.2) [software]. Available from http://netbeans.org/downloads/7.1.2/

Shell, D.L. (1959). A high-speed sorting procedure. *Communication of the ACM, 2*(7), 30-32. Retrieved from the ACM Digital Library.