Karan Bharaj (T00693289). Assignment 1- COMP 2231

Question 1

ShellSort on Example Array

```
--------------QUESTION 1--------------
==============EXAMPLE ARRAY==============
Array: [9, 6, 8, 12, 3, 1, 7]
Pass #: 1 Gap Size: 3
[9, 3, 8, 12, 6, 1, 7]
[9, 3, 1, 12, 6, 8, 7]
[9, 3, 1, 7, 6, 8, 12]
Pass #: 2 Gap Size: 1
[3, 9, 1, 7, 6, 8, 12]
[3, 1, 9, 7, 6, 8, 12]
[3, 1, 7, 9, 6, 8, 12]
[3, 1, 7, 6, 9, 8, 12]
[3, 1, 7, 6, 8, 9, 12]
Pass #: 3 Gap Size: 1
[1, 3, 7, 6, 8, 9, 12]
[1, 3, 6, 7, 8, 9, 12]
Pass #: 4 Gap Size: 1
```

ShellSort executed on the Example Array, showing the array output each time a swap is done. The pass count at each pass is also shown, along with the gap size during the pass.

ShellSort on Random Array with 10 integers

```
==============RANDOM ARRAY OF SIZE 10===============
Array: [9, 14, 19, 16, 2, 19, 7, 12, 18, 8]
Pass #: 1 Gap Size: 5
[9, 7, 19, 16, 2, 19, 14, 12, 18, 8]
[9, 7, 12, 16, 2, 19, 14, 19, 18, 8]
Pass #: 2 Gap Size: 2
[9, 7, 2, 16, 12, 19, 14, 19, 18, 8]
[9, 7, 2, 16, 12, 19, 14, 8, 18, 19]
Pass #: 3 Gap Size: 1
[7, 9, 2, 16, 12, 19, 14, 8, 18, 19]
[7, 2, 9, 16, 12, 19, 14, 8, 18, 19]
[7, 2, 9, 12, 16, 19, 14, 8, 18, 19]
[7, 2, 9, 12, 16, 14, 19, 8, 18, 19]
[7, 2, 9, 12, 16, 14, 8, 19, 18, 19]
[7, 2, 9, 12, 16, 14, 8, 18, 19, 19]
Pass #: 4 Gap Size: 1
[2, 7, 9, 12, 16, 14, 8, 18, 19, 19]
[2, 7, 9, 12, 14, 16, 8, 18, 19, 19]
[2, 7, 9, 12, 14, 8, 16, 18, 19, 19]
Pass #: 5 Gap Size: 1
[2, 7, 9, 12, 8, 14, 16, 18, 19, 19]
Pass #: 6 Gap Size: 1
[2, 7, 9, 8, 12, 14, 16, 18, 19, 19]
Pass #: 7 Gap Size: 1
[2, 7, 8, 9, 12, 14, 16, 18, 19, 19]
Pass #: 8 Gap Size: 1
```

ShellSort executed on the Random Array with 10 integers, showing the array output each time a swap is done. The pass count at each pass is also shown, along with the gap size during the pass.

ShellSort on Random Array with 20 integers

```
==========RANDOM ARRAY OF SIZE 20==========
Array: [31, 14, 17, 16, 5, 33, 24, 15, 19, 21, 19, 28, 4, 21, 2, 6, 1, 33, 17, 19]
Pass #: 1 Gap Size: 10
[19, 14, 17, 16, 5, 33, 24, 15, 19, 21, 31, 28, 4, 21, 2, 6, 1, 33, 17, 19]
[19, 14, 4, 16, 5, 33, 24, 15, 19, 21, 31, 28, 17, 21, 2, 6, 1, 33, 17, 19]
[19, 14, 4, 16, 2, 33, 24, 15, 19, 21, 31, 28, 17, 21, 5, 6, 1, 33, 17, 19]
[19, 14, 4, 16, 2, 6, 24, 15, 19, 21, 31, 28, 17, 21, 5, 33, 1, 33, 17, 19]
[19, 14, 4, 16, 2, 6, 1, 15, 19, 21, 31, 28, 17, 21, 5, 33, 24, 33, 17, 19]
[19, 14, 4, 16, 2, 6, 1, 15, 17, 21, 31, 28, 17, 21, 5, 33, 24, 33, 19, 19]
[19, 14, 4, 16, 2, 6, 1, 15, 17, 19, 31, 28, 17, 21, 5, 33, 24, 33, 19, 21]
Pass #: 2 Gap Size: 5
[6, 14, 4, 16, 2, 19, 1, 15, 17, 19, 31, 28, 17, 21, 5, 33, 24, 33, 19, 21]
[6, 1, 4, 16, 2, 19, 14, 15, 17, 19, 31, 28, 17, 21, 5, 33, 24, 33, 19, 21]
[6, 1, 4, 16, 2, 19, 14, 15, 17, 5, 31, 28, 17, 21, 19, 33, 24, 33, 19, 21]
[6, 1, 4, 16, 2, 19, 14, 15, 17, 5, 31, 24, 17, 21, 19, 33, 28, 33, 19, 21]
[6, 1, 4, 16, 2, 19, 14, 15, 17, 5, 31, 24, 17, 19, 19, 33, 28, 33, 21, 21]
Pass #: 3 Gap Size: 2
[4, 1, 6, 16, 2, 19, 14, 15, 17, 5, 31, 24, 17, 19, 19, 33, 28, 33, 21, 21]
[4, 1, 2, 16, 6, 19, 14, 15, 17, 5, 31, 24, 17, 19, 19, 33, 28, 33, 21, 21]
[4, 1, 2, 16, 6, 15, 14, 19, 17, 5, 31, 24, 17, 19, 19, 33, 28, 33, 21, 21]
[4, 1, 2, 16, 6, 15, 14, 5, 17, 19, 31, 24, 17, 19, 19, 33, 28, 33, 21, 21]
[4, 1, 2, 16, 6, 15, 14, 5, 17, 19, 17, 24, 31, 19, 19, 33, 28, 33, 21, 21]
[4, 1, 2, 16, 6, 15, 14, 5, 17, 19, 17, 19, 31, 24, 19, 33, 28, 33, 21, 21]
[4, 1, 2, 16, 6, 15, 14, 5, 17, 19, 17, 19, 19, 24, 31, 33, 28, 33, 21, 21]
[4, 1, 2, 16, 6, 15, 14, 5, 17, 19, 17, 19, 19, 24, 28, 33, 31, 33, 21, 21]
[4, 1, 2, 16, 6, 15, 14, 5, 17, 19, 17, 19, 19, 24, 28, 33, 21, 33, 31, 21]
[4, 1, 2, 16, 6, 15, 14, 5, 17, 19, 17, 19, 19, 24, 28, 33, 21, 21, 31, 33]
Pass #: 4 Gap Size: 1
[1, 4, 2, 16, 6, 15, 14, 5, 17, 19, 17, 19, 19, 24, 28, 33, 21, 21, 31, 33]
[1, 2, 4, 16, 6, 15, 14, 5, 17, 19, 17, 19, 19, 24, 28, 33, 21, 21, 31, 33]
[1, 2, 4, 6, 16, 15, 14, 5, 17, 19, 17, 19, 19, 24, 28, 33, 21, 21, 31, 33]
[1, 2, 4, 6, 15, 16, 14, 5, 17, 19, 17, 19, 19, 24, 28, 33, 21, 21, 31, 33]
[1, 2, 4, 6, 15, 14, 16, 5, 17, 19, 17, 19, 19, 24, 28, 33, 21, 21, 31, 33]
[1, 2, 4, 6, 15, 14, 5, 16, 17, 19, 17, 19, 19, 24, 28, 33, 21, 21, 31, 33]
[1, 2, 4, 6, 15, 14, 5, 16, 17, 17, 19, 19, 19, 24, 28, 33, 21, 21, 31, 33]
[1, 2, 4, 6, 15, 14, 5, 16, 17, 17, 19, 19, 19, 24, 28, 21, 33, 21, 31, 33]
[1, 2, 4, 6, 15, 14, 5, 16, 17, 17, 19, 19, 19, 24, 28, 21, 21, 33, 31, 33]
[1, 2, 4, 6, 15, 14, 5, 16, 17, 17, 19, 19, 19, 24, 28, 21, 21, 31, 33, 33]
Pass #: 5 Gap Size: 1
[1, 2, 4, 6, 14, 15, 5, 16, 17, 17, 19, 19, 19, 24, 28, 21, 21, 31, 33, 33]
[1, 2, 4, 6, 14, 5, 15, 16, 17, 17, 19, 19, 19, 24, 28, 21, 21, 31, 33, 33]
[1, 2, 4, 6, 14, 5, 15, 16, 17, 17, 19, 19, 19, 24, 21, 28, 21, 31, 33, 33]
[1, 2, 4, 6, 14, 5, 15, 16, 17, 17, 19, 19, 19, 24, 21, 21, 28, 31, 33, 33]
Pass #: 6 Gap Size: 1
[1, 2, 4, 6, 5, 14, 15, 16, 17, 17, 19, 19, 19, 24, 21, 21, 28, 31, 33, 33]
[1, 2, 4, 6, 5, 14, 15, 16, 17, 17, 19, 19, 19, 21, 24, 21, 28, 31, 33, 33]
[1, 2, 4, 6, 5, 14, 15, 16, 17, 17, 19, 19, 19, 21, 21, 24, 28, 31, 33, 33]
Pass #: 7 Gap Size: 1
[1, 2, 4, 5, 6, 14, 15, 16, 17, 17, 19, 19, 19, 21, 21, 24, 28, 31, 33, 33]
Pass #: 8 Gap Size: 1
```

ShellSort executed on a Random Array with 20 integers, showing the array output each time a swap is done. The pass count at each pass is also shown, along with the gap size during the pass.

Question 2

Original/Old BubbleSort and New Bubble Sort (BubbleSort2) algorithms are executed on the same Random Array with 10 integers that was used with Shell Sort.

(Old) BubbleSort on Random Array with 10 integers

```
========OLD BUBBLE SORT RANDOM ARRAY OF SIZE 10========
Array: [9, 14, 19, 16, 2, 19, 7, 12, 18, 8]
Pass #: 1
[9, 14, 16, 2, 19, 7, 12, 18, 8, 19]
Pass #: 2
[9, 14, 2, 16, 7, 12, 18, 8, 19, 19]
Pass #: 3
[9, 2, 14, 7, 12, 16, 8, 18, 19, 19]
Pass #: 4
[2, 9, 7, 12, 14, 8, 16, 18, 19, 19]
Pass #: 5
[2, 7, 9, 12, 8, 14, 16, 18, 19, 19]
Pass #: 6
[2, 7, 9, 8, 12, 14, 16, 18, 19, 19]
Pass #: 7
[2, 7, 8, 9, 12, 14, 16, 18, 19, 19]
Pass #: 8
[2, 7, 8, 9, 12, 14, 16, 18, 19, 19]
Pass #: 9
[2, 7, 8, 9, 12, 14, 16, 18, 19, 19]
Pass #: 10
[2, 7, 8, 9, 12, 14, 16, 18, 19, 19]
```

(Old) BubbleSort takes 10 passes of this array, even though the array is sorted after the 8th pass is completed.

(New) BubbleSort2 on Random Array with 10 integers

```
========NEW BUBBLE SORT RANDOM ARRAY OF SIZE 10========
Array: [9, 14, 19, 16, 2, 19, 7, 12, 18, 8]
Pass #: 1
[9, 14, 16, 2, 19, 7, 12, 18, 8, 19]
Pass #: 2
[9, 14, 2, 16, 7, 12, 18, 8, 19, 19]
Pass #: 3
[9, 2, 14, 7, 12, 16, 8, 18, 19, 19]
Pass #: 4
[2, 9, 7, 12, 14, 8, 16, 18, 19, 19]
Pass #: 5
[2, 7, 9, 12, 8, 14, 16, 18, 19, 19]
Pass #: 6
[2, 7, 9, 8, 12, 14, 16, 18, 19, 19]
Pass #: 7
[2, 7, 8, 9, 12, 14, 16, 18, 19, 19]
Pass #: 8
[2, 7, 8, 9, 12, 14, 16, 18, 19, 19]
```

Alternatively, the new Bubble sort takes 8 passes of this array after analysing that there are no swaps possible at the 8th pass.

This (new) BubbleSort2 algorithm also makes the same number of passes as the shell sort algorithm did

(Old) BubbleSort on Sorted Array with 10 integers

```
========OLD BUBBLE SORT SORTED ARRAY OF SIZE 10========
Array: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Pass #: 1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Pass #: 2
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Pass #: 3
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Pass #: 4
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Pass #: 5
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Pass #: 6
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Pass #: 7
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Pass #: 8
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Pass #: 9
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Pass #: 10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

(Old) BubbleSort takes 10 passes of this array, even though the array is sorted after the 1st pass is completed.

(New) BubbleSort2 on Sorted Array with 10 integers

```
========NEW BUBBLE SORT SORTED ARRAY OF SIZE 10======
Array: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Pass #: 1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Alternatively, the (new) BubbleSort2 method takes only a single pass and terminates after analysing that the array is sorted.

Question 3

Since it is possible to generate random arrays that could be more biased towards one algorithm over the other, using 20 iterations and then evaluating the average of the runtimes, and average of the number of comparisons, swaps and passes would give a more reliable comparison. This is the strategy adopted for the subsequent analysis.

Print outputs of the code in the array sizes instructed (10, 100 and 1000 integers) are shown below:

Algorithm analysis on a random array with 10 integers

```
================================================
Iteration #: 1
================RANDOM ARRAY================
Size of Array: 10
================================================
--------------Shell Sort----------------
Number of Comparisons: 58
Number of Swaps: 13
Execution Time: 0 milliseconds
------------------------------------------------
--------------Bubble Sort----------------
Number of Comparisons: 45
Number of Swaps: 20
Execution Time: 0 milliseconds
------------------------------------------------
---------------Bubble2 Sort-----------------
Number of Comparisons: 42
Number of Swaps: 20
Execution Time: 0 milliseconds
------------------------------------------------
```

For this random array, Shell sort performed the most comparisons (58), but the least swaps.

As expected, BubbleSort2 performed fewer comparisons than its predecessor (Old Bubble Sort), for the same number of swaps. At worst, the BubbleSort2 algorithm is expected to perform the same number of comparison as the Old BubbleSort algorithm (when the array is unsorted until its last pass).

Algorithm analysis on a sorted array with 10 integers

```
================SORTED ARRAY================
Size of Array: 10
================================================
--------------Shell Sort----------------
Number of Comparisons: 22
Number of Swaps: 0
Execution Time: 0 milliseconds
------------------------------------------------
--------------Bubble Sort----------------
Number of Comparisons: 45
Number of Swaps: 0
Execution Time: 0 milliseconds
------------------------------------------------
---------------Bubble2 Sort-----------------
Number of Comparisons: 9
Number of Swaps: 0
Execution Time: 0 milliseconds
------------------------------------------------
================================================
```

As expected, the sorted arrays incurs no swaps for all algorithms. The (old) BubbleSort makes the same number of comparisons it will always make for an array with 10 integers (45 comparisons). Alternatively, (new) BubbleSort2 stops after the first pass when no swaps were made. Shell sort on the other hand also terminates when gap size is reduced to one and pass is made with no swaps.

However, this array list size is not large enough to reliably distinguish between the various algorithms, especially from a runtime perspective.

Algorithm analysis on a random array with 100 integers

```
================================================
Iteration #: 4
================RANDOM ARRAY=================
Size of Array: 100
================================================
---------------Shell Sort----------------
Number of Comparisons: 2483
Number of Swaps: 531
Execution Time: 0 milliseconds
------------------------------------------------
---------------Bubble Sort----------------
Number of Comparisons: 4950
Number of Swaps: 2203
Execution Time: 0 milliseconds
------------------------------------------------
----------------Bubble2 Sort-----------------
Number of Comparisons: 4697
Number of Swaps: 2203
Execution Time: 0 milliseconds
------------------------------------------------
================================================
```

For a random array with 100 integers, Bubble Sort performed the most comparisons (4950), more than BubbleSort2 (4697). As expected, the number of Swaps for both were the same. However, For this array, Shell Sort performed the least number of comparisons and swaps. However, this isn't always the case for the number of comparisons, as shown in the example below.

Algorithm analysis on another random array with 100 integers

```
================================================
Iteration #: 6
================RANDOM ARRAY=================
Size of Array: 100
================================================
---------------Shell Sort----------------
Number of Comparisons: 5354
Number of Swaps: 730
Execution Time: 0 milliseconds
------------------------------------------------
---------------Bubble Sort----------------
Number of Comparisons: 4950
Number of Swaps: 2442
Execution Time: 0 milliseconds
------------------------------------------------
----------------Bubble2 Sort-----------------
Number of Comparisons: 4905
Number of Swaps: 2442
Execution Time: 1 milliseconds
------------------------------------------------
```

The iteration here shows an example where Shell Sort has made the most comparisons (5354), but as with the smaller arrays, it makes the least swaps.

This is due to the approach adopted for this type of ShellSort algorithm, and does not mean that ShellSort is inferior to BubbleSort. Later, the impact of this on the actual runtime is analysed further.

This array list size is still not large enough to reliably distinguish between the various algorithms, especially from a runtime perspective.

Algorithm analysis on a sorted array with 100 integers

```
================SORTED ARRAY=================
Size of Array: 100
=============================================
--------------Shell Sort---------------
Number of Comparisons: 503
Number of Swaps: 0
Execution Time: 0 milliseconds
---------------------------------------------
--------------Bubble Sort--------------
Number of Comparisons: 4950
Number of Swaps: 0
Execution Time: 0 milliseconds
---------------------------------------------
--------------Bubble2 Sort-----------------
Number of Comparisons: 99
Number of Swaps: 0
Execution Time: 0 milliseconds
---------------------------------------------
=============================================
```

Similar to the smaller array, a sorted array with 100 integers incurs the same number of comparison by the (Old) BubbleSort algorithm. BubbleSort2 terminates after the 1st pass, after no swaps were made, and ShellSort terminates after the first pass at a gap size of 1.

Algorithm analysis on a random array with 1,000 integers

```
=============================================
Iteration #: 5
================RANDOM ARRAY=================
Size of Array: 1000
=============================================
--------------Shell Sort---------------
Number of Comparisons: 409604
Number of Swaps: 37005
Execution Time: 0 milliseconds
---------------------------------------------
--------------Bubble Sort--------------
Number of Comparisons: 499500
Number of Swaps: 250932
Execution Time: 1 milliseconds
---------------------------------------------
----------------Bubble2 Sort-----------------
Number of Comparisons: 499455
Number of Swaps: 250932
Execution Time: 1 milliseconds
---------------------------------------------
=============================================
```

For a large random array, as expected, (Old) BubbleSort performed the most comparisons (499,500), more than (New) BubbleSort2 (499,455), and the same number of Swaps. For this array, Shell sort makes the least comparisons and the least swaps. However, there are some arrays at this size where ShellSort makes the most comparisons (also seen with an array size of 100 integers). However, it always performs the least swaps.

This array list size is still not large enough to reliably distinguish between the various algorithms, especially from a runtime perspective. The latter analysis at with larger arrays will address this.

Algorithm analysis on a sorted array with 1,000 integers

```
================SORTED ARRAY=================
Size of Array: 1000
=============================================
--------------Shell Sort---------------
Number of Comparisons: 8006
Number of Swaps: 0
Execution Time: 0 milliseconds
---------------------------------------------
--------------Bubble Sort---------------
Number of Comparisons: 499500
Number of Swaps: 0
Execution Time: 1 milliseconds
---------------------------------------------
---------------Bubble2 Sort-----------------
Number of Comparisons: 999
Number of Swaps: 0
Execution Time: 0 milliseconds
---------------------------------------------
=============================================
```

As expected , (New) BubbleSort2 makes the fewest comparisons for the sorted array, because it realises the array is sorted after a single pass. (Old) BubbleSort makes the same number of comparisons it always will for an array of this size, whereas Shell Sort terminates once it reaches a gap size of 1 and passes through the array without swapping any numbers.

Comparison of algorithm performance on multiple Random arrays

The table below shows average runtime, average number of swaps, average number of comparisons and average number of passes for each algorithm at increasing array sizes across the 20 iterations.

Sorting Method	Array Size	Comparisons	Swaps	Passes	Runtime (milliseconds)
ShellSort Random	10	46	12	6	0.00
	100	3,681	630	38	0.07
	1,000	474,339	40,751	476	0.57
	10,000	54,108,606	3,800,474	5,412	66.80
	100,000	6,209,977,921	389,907,346	62,101	11,682.60
(Old) BubbleSort Random	10	45	23	10	0.00
	100	4,950	2,465	100	0.09
	1,000	499,500	250,847	1,000	0.79
	10,000	49,995,000	25,012,622	10,000	98.15
	100,000	4,999,950,000	2,499,575,685	100,000	20,526.15
(New) BubbleSort2 Random	10	43	23	8	0.00
	100	4,863	2,421	88	0.08
	1,000	498,378	250,731	956	0.75
	10,000	49,986,399	25,003,813	9,880	91.55
	100,000	4,999,882,505	2,499,239,010	99,657	15,354.55

At larger array sizes (10,000 and 100,000 integers), the runtimes can be reliably calculated for comparisons between algorithms. At large array sizes, ShellSort makes more comparisons than both BubbleSort algorithms, however, the number of swaps and passes are less. Furthermore, its runtime is much less than the BubbleSort algorithms (up to twice as fast as the Old Bubble Sort algorithm on the largest array). As expected, the Old BubbleSort algorithm has the highest runtime, making the most swaps and passes. In summary, ShellSort the most efficient algorithm at larger array sizes.

Comparison of algorithm performance on multiple Sorted arrays

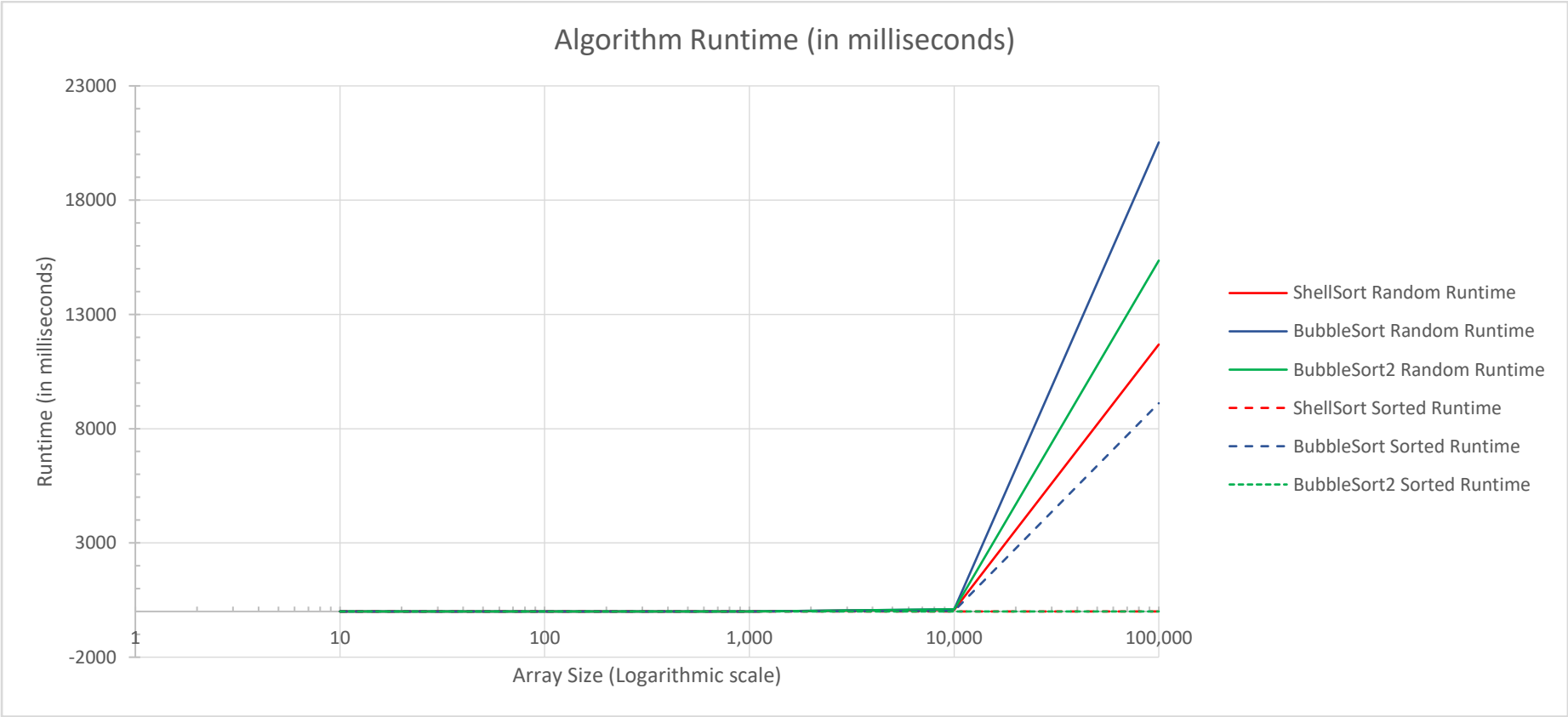
The table below repeats the information in the table above, but does so for the sorted arrays of varying sizes.

Sorting Method	Array Size	Comparisons	Swaps	Passes	Runtime (milliseconds)
ShellSort Sorted	10	22	0	3	0.00
	100	503	0	6	0.02
	1,000	8,006	0	9	0.06
	10,000	120,005	0	13	0.10
	100,000	1,500,006	0	16	1.09
(Old) BubbleSort Random	10	45	0	10	0.00
	100	4,950	0	100	0.09
	1,000	499,500	0	1,000	0.35
	10,000	49,995,000	0	10,000	24.39
	100,000	4,999,950,000	0	100,000	9,111.85
(New) BubbleSort2 Random	10	9	0	1	0.00
	100	99	0	1	0.00
	1,000	999	0	1	0.00
	10,000	9,999	0	1	0.00
	100,000	99,999	0	1	0.00

As expected, (old) BubbleSort takes the most time since it continues to compare elements even though the list is sorted. At large array sizes, additional runtime is accumulated due to these comparisons made. (New) BubbleSort2 has next to no runtime since it terminates after the first pass at all array sizes. ShellSort terminates after no swaps were made at a gap size of 1. Comparisons made prior to this contribute to the runtime incurred in the larger arrays.

The previous two tables show that the runtime is a function of the number of comparisons and swaps an algorithm makes across all its passes. Thus, in order to empirically analyse algorithm efficiency, the runtime at varying array sizes are analysed graphically for each sorting algorithm.

Runtime comparison of algorithm efficiency on multiple Random and Sorted arrays



The simple graph above shows the runtime differences between the sorting algorithms at varying array sizes. The differences are most discernible at the largest array size (100,000 integers). All algorithms have a quadratic time ($O(n^2)$) complexity, evident from their shape of each of them. When compared with (Old) BubbleSort, (New) BubbleSort2 has a shorter runtime due to its early termination. Furthermore, the runtime analysis shows ShellSort's main advantage, which is to move out-of-place integers into position faster than the neighbouring exchange that BubbleSort uses. Despite having quadratic shape, this makes ShellSort's growth function is more time efficient than the BubbleSort algorithms. Even though it has not been analysed, all three algorithms have a worst case space complexity of O(1). Further increases in array size would extrapolate the quadratic curves, and ShellSort would be expected to continue to perform better than the BubbleSort algorithms.