Deuce Palmer
CS 3353 - 001

Program 1

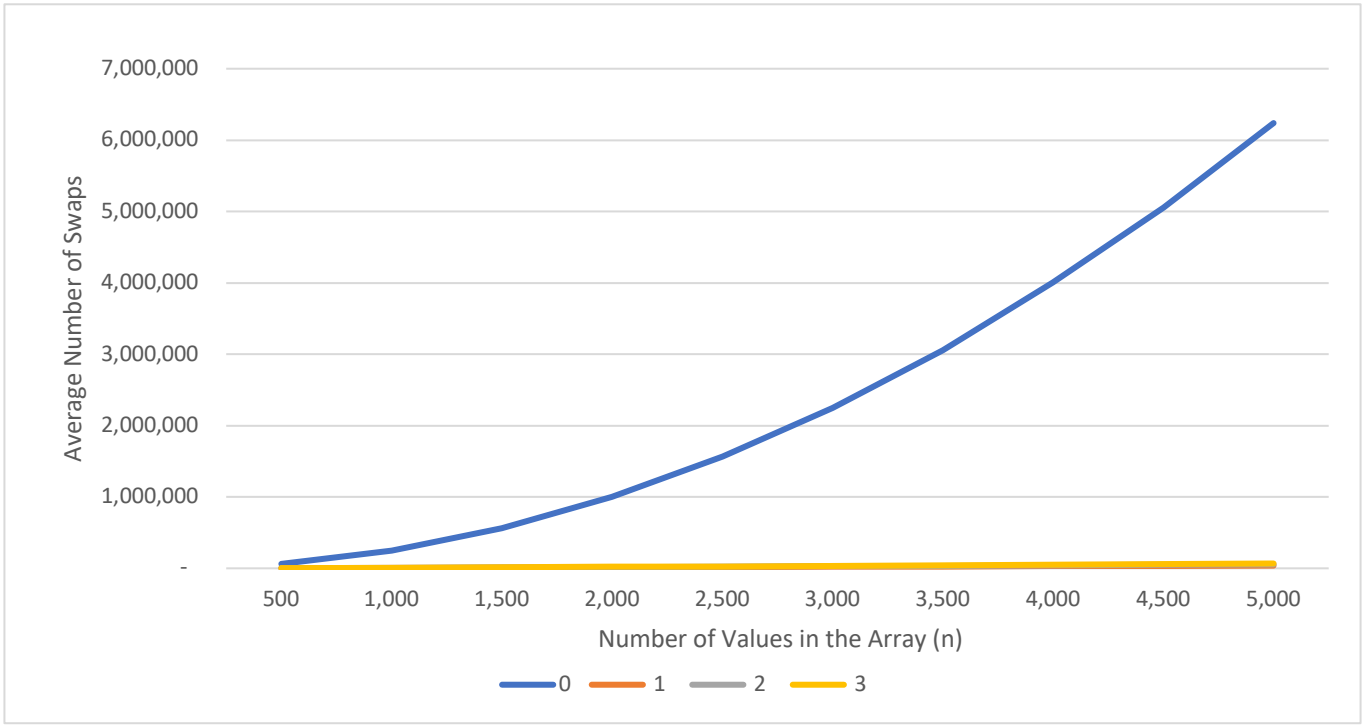
Table

Including average number of swaps and standard deviation for each array size and code value.

	n = 500				n = 1,000				n = 1,500			
	0	1	2	3	0	1	2	3	0	1	2	3
Average	62,364	2,662	3,037	3,436	249,516	5,247	7,167	8,900	560,817	8,601	12,059	14,774
Stan Dev	1,874	109	142	169	5,339	122	358	396	9,204	155	562	599
	n = 2,000				n = 2,500				n = 3,000			
	0	1	2	3	0	1	2	3	0	1	2	3
Average	997,463	12,178	17,455	21,317	1,565,579	15,965	23,198	28,236	2,250,901	20,075	29,206	35,835
Stan Dev	15,518	173	885	871	19,579	166	1,234	1,542	28,434	243	1,296	1,569
	n = 3,500				n = 4,000				n = 4,500			
	0	1	2	3	0	1	2	3	0	1	2	3
Average	3,055,077	24,307	35,264	43,803	4,003,906	28,835	42,384	51,612	5,057,453	33,504	49,173	59,643
Stan Dev	37,520	255	1,692	2,188	40,115	368	2,074	2,703	53,550	412	2,671	2,485
	n = 5,000											
	0	1	2	3								
Average	6,238,651	38,283	55,677	67,912								
Stan Dev	56,382	506	2,954	2,650								

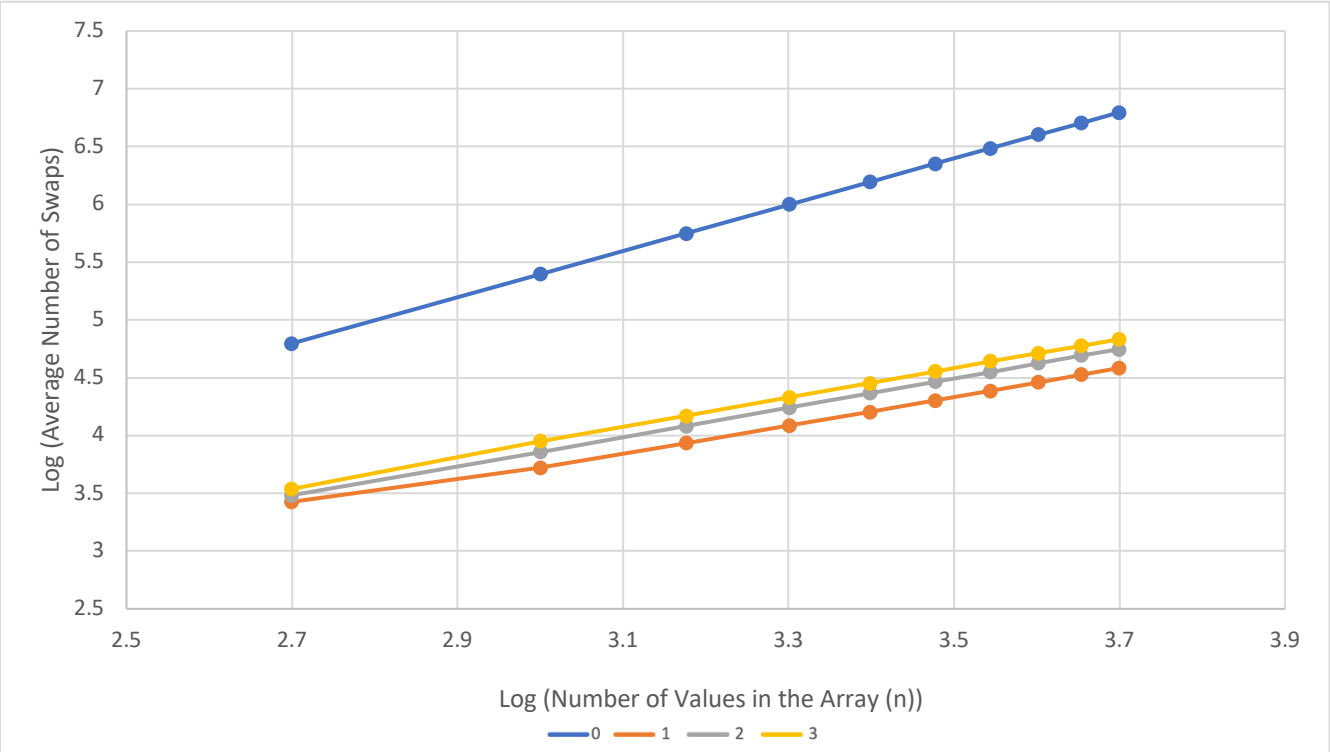
Graph 1 (X: n & Y: average number of swaps)

All 4 lines are plotted, but the difference between 1, 2, and 3 cannot be visualized since line 0 (insertion sort) has drastically higher average swap values



Graph 2 (X: Log(n) & Y: Log(average number of swaps))

Base 10 was used



Analysis

Based on the table and graphs collected, the most efficient shell sort case was case 1. Case 1 refers to the "code" value of 1, which creates an hlist[] containing k^2 values. This case was the most efficient because the hlist[] values incremented at the slowest rate. For example, let's take a list of 101 elements to be sorted. Case 1 would use 10 hlist[] values {1,4,9,16,25,36,49,64,81,100}, while case 2 would use 6 hlist[] value {1,3,7,15,31,63} and case 3 only 4 {1,4,13,40}. Case 1 is able to create more incremental hlist[] values, meaning that the sorting will begin with values that are much farther away in proximity. As a result, the necessary swaps are minimized as values can take larger jumps across the array to their correct location during the beginning of the sorting algorithm. The same reasoning can be applied to why case 2 outperformed case 3. Lastly, case 0 had, without a doubt, the worst performance. Case 0's hlist[] only consisted of 1 element {1}. As a result, the sorting algorithm mimicked insertion sort and could only swap adjacent elements in the array.

When analyzing the test cases asymptotically, we are concerned with the runtime as the size of the array approaches infinity. To begin, case 0 (insertion sort) has a worst and average case of O(n^2). This inefficiency was seen in the required number of swaps as compared to the 3 other test cases. Case 2 utilizes Hibbard's sequence for its hlist[] values and has an improved performance of O(n^3/2). In addition, Case 3 utilizes Knuth's sequence of ((3^k-1)/2) and also has a time complexity of O(n^3/2). There is a strong relationship between gap sequence and time complexity, as the smaller the gaps are the better the time complexity. This can further be shown as 2^p3^q or {1,2,3,4,6,8,9,12} gives the time complexity of O(nLog^2n). This further proves why case 1 is our best performer as its gap sequence is the smallest among the four test cases.