**Part #1:**

Outer loop: executes until X > N 2 . Taking the Log of both sides gets us the time complexity of the outer loop, which is O(Log 2 N2). The inner loop executes N times and decrements by 1 after each iteration. Therefore, the inner loop has a time complexity of O(N). To get the total time complexity = inner \* outer = N \* (log 2 N)2. Since the statements take 6 milliseconds to execute, we multiply 6 by N \* (log 2 N)2 to get the time to execute the construct:

= 6 ms. \* 2000 log 2 (2000)2 millisec.

**= 264000 millisec.**

**Part #2:**

**Table 1:** (Prototype integers 1-16):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Ordered Data | | Reverse Order | | Random Order | |
|  | Comparisons | Exchanges | Comparisons | Exchanges | Comparisons | Exchanges |
| Bubble Sort | 15 | 0 | 120 | 120 | 120 | 66 |
| Selection Sort | 120 | 15 | 120 | 15 | 120 | 15 |
| Insertion Sort | 0 | 0 | 120 | 120 | 66 | 66 |
| Shell Sort  (H/2) | 0 | 0 | 32 | 32 | 17 | 17 |
| Shell sort  (3 \* H + 1) | 0 | 0 | 14 | 14 | 22 | 22 |
| Merge Sort | 32 | 0 | 32 | 32 | 20 | 20 |

**Table 2:** (Large Data – Integers 1-2000):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Ordered Data | | Reverse Order | | Random Order | |
|  | Comparisons | Exchanges | Comparisons | Exchanges | Comparisons | Exchanges |
| Bubble Sort | 1999 | 0 | 1999000 | 1999000 | 1998724 | 974006 |
| Selection Sort | 1999000 | 1999 | 1999000 | 1999 | 1999000 | 1999 |
| Insertion Sort | 0 | 0 | 1999000 | 1999000 | 974006 | 974006 |
| Shell Sort  (H/2) | 0 | 0 | 10400 | 10400 | 18707 | 18707 |
| Shell Sort  (3 \* H + 1) | 0 | 0 | 4566 | 4566 | 17199 | 17199 |
| Merge Sort | 11088 | 0 | 10864 | 11088 | 2535 | 9792 |

**Report:**

Since bubble sort is sensitive to data, when running against the best case (ordered data), the algorithm makes n-1 comparisons and no exchanges (since the Boolean condition determined that no exchanges need to be made in an already ordered data set). This gives bubble sort a best running efficiency of O(n). Also, in the worst case, bubble sort will exchange n-1 times for the first iteration of the outer loop, n -2 times for the second loop, and so on, to get a total of (n\*(n-1)/2) exchanges. The same goes for the number of comparisons in the worst case, with the algorithm achieving (n\*(n-1)/2) comparisons – in all, this means a worst runtime efficiency of O(n2). In the average case, bubble sort makes < n^2 swaps and <n^2 number of comparisons, as exemplified by my data, landing at an average efficiency of O(n2).

For selection sort, the algorithm is **not** sensitive to input. This means that regardless of the case, selection sort will always execute n-1 exchanges (O(n) efficiency) and (n\*(n-1)/2) comparisons (O(n2) efficiency). Overall, the algorithm runs at an O(n2) efficiency for the best, worst, and average sorting scenarios.

Since insertion sort is sensitive to data, when running against the best case (ordered data), the algorithm makes no comparisons and no exchanges. This is because in the best case, the inner loop will never run, which represents zero comparisons. If no comparisons are made, this means no swaps happened either (again, the data is already in order). Overall, this equates to a best time complexity of O(n). In the worst case, insertion sort makes (n\*(n-1)/2) comparisons (the maximum because the data is in reverse order) and (n\*(n-1)/2) exchanges, which equates to an O(N)^2 efficiency. The average time complexity of insertion sort is O(N)^2. This is because on average, the bad and good cases tend to cancel each other out, which leaves the algorithm to examine about half the number of items in the list. From my data, is shown by ~ n2/4 comparisons (about half as many as selection sort) and exchanges.

Looking at my shell sort empirical data, it behaves like insertion sort (since it is an extension of insertion sort); it is sensitive to data like bubble sort and insertion sort. However, when compared to insertion sort (as my prototype and large data indicates), it makes much less comparisons and swaps because the logic of shell sort is to sort the furthest elements first, and shell sort divides the list into smaller subsets, and each of those sub-lists are sorted like insertion sort. This means for the ordered and prototype datasets; shell sort makes 0 comparisons and exchanges. For all trials of the random shell sort datasets, the efficiency of N\*log(N) is proven. This is proven when solving for the unknown constant A in O(A\*N\* Log N) the number of comparisons in terms of N(3/2) for the large and prototype data is .392 \* 10 (3/2)  and .171 \* 10 (3/2) respectively when H = 3 \* H + 1 and .426 \* 10 (3/2)  and .132 \* 10 (3/2) respectively for large and prototype data when H = (H / 2). Since these values of A correlate with the results of Table 1.4 in Lecture 5, my empirical results are valid. The N \* Log (N) efficiency of shell sort is proven for reverse ordered prototype and large data; for when H = H /2, shell sort makes N log N comparisons and exchanges, and it makes about half of N Log N comparisons and exchanges for when H = 3 \* H + 1.

When comparing the two incremental sequences tested, H = 3\*H +1 vs. H = H / 2, it can be seen that H = 3\*H +1 *generally* requires less comparisons and swaps. This is because it decreases relatively exponentially when compared to H = H / 2, and it is proven in my empirical data that exponential functions gain speed with time; the large random dataset required less comparisons and exchanges when set to H = 3\*H +1, but required slightly more in the small random dataset (where the algorithm didn’t have enough time to gain as much speed as H = H / 2).

Looking at my merge sort empirical data, the results are valid based on a number of factors. It is known that merge sort is memory intensive (continuously calls merge and has to divided into a source and destination array), and this is proven since it makes a more comparisons in the case of ordered data compared to shell sort, and it makes >= the number of comparisons and exchanges when the data is in reverse order when compared to shell sort (which proves the N\* log (N) efficiency) . However, in the case of randomly generated prototype data, merge sort makes a comparable number of comparisons and exchanges when comparing to shell sort (which proves the N\* log (N) efficiency). Also, when looking at randomly generated large data, merge sort makes significantly less comparisons and exchanges than shell sort. This is because merge sort works on large data sets well because of its dynamic divide and conqueror technique, and shell sort is limited to being ran in-place. However, this recursive divide-and-conquer approach is why both shell sort and merge sort complete less comparisons and exchanges, generally, than the elementary sorts; they compare and exchange *parts* of the dataset instead of the entirety of it.