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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Ordered Data | | Reverse Order | | Random Order | |
|  | Comparisons | Exchanges | Comparisons | Exchanges | Comparisons | Exchanges |
| 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 |
| Heap Sort | 63 | 58 | 37 | 42 | 55 | 51 |

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Ordered Data | | Reverse Order | | Random Order | |
|  | Comparisons | Exchanges | Comparisons | Exchanges | Comparisons | Exchanges |
| 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 |
| Heap Sort | 29237 | 21300 | 24696 | 18708 | 27302 | 20162 |

**Report:**

For both the prototype and large data, heap sort (in bottom-up construction) does make approximately 1.5 N Log N comparisons and exchanges for ordered, reverse ordered, and random ordered data. Looking at the prototype data composed of 16 integers, based on its theoretical efficiency of 1.5N Log N, heap sort should be making around 96 comparisons and exchanges. Looking at my empirical results, this hypothesis is proven. For ordered data, heap sort made 121 comparisons and exchanges, 79 for reverse order, and 106 for random ordered data. This correlates to efficiencies all around 1.5 N Log N: 1.89 N Log N for ordered data, 1.23 N Log N for reverse order, and 1.65 N Log N for random ordered data. Given that the empirical results produce efficiencies for each of the prototype datasets all around 1.5 N Log N, it can be concluded that my results for heap sort are valid. The same can be said for the large datasets, which should be producing around 32897 comparisons and exchanges according to the theoretical efficiency of 1.5 N Log N. Referring to the empirical results, heapsort, for the ordered dataset made 50,537 comparisons and exchanges, 43,404 for reverse order, and 47,464 for random order. Again, these correlate to efficiencies similar to 1.5 N Log N: 2.3 N Log N for ordered data, 1.97 N Log N for reverse order, and 2.16 N Log N for random ordered data. Again, given that the empirical results produce efficiencies for each of the prototype datasets all around 1.5 N Log N, it can be concluded that my results for heap sort are valid.

When compared to merge sort and shell sort, heap sort (in this bottom-up construction) is proven to be less efficient, as it makes more comparisons and exchanges in *every* case of prototype and large datasets. In theory, this makes sense, since merge sort and heap sort are theorized to make around N Log N number of comparisons and exchanges, whereas heap sort does it in 1.5 N Log N comparisons and exchanges. As proven in my homework 2 report, shell sort is sensitive to data, so in the case of ordered data, it doesn’t even make any comparisons and exchanges. Therefore, shell sort beats out heap sort for ordered data immediately, especially since heap sort always makes comparisons and exchanges for ordered datasets. Merge sort also doesn’t make any exchanges if the data is ordered, which helps it to also beat out heap sort in this case. In comparison, for reverse ordered and data, heap sort makes ~ 1.5 times the number of comparisons and exchanges compared to shell sort and merge sort. Again, this makes sense since heap sort’s bottom-up implementation efficiency is 1.5 N Log N compared to merge and shell sort running at N log N.

Comparing all 3 advanced sorts, shell sort is the best for ordered data. As explained before, shell sort is data sensitive, meaning that if the list is already ordered, it won’t go through the process of comparing and swapping data entries. In total, it makes zero comparisons and swaps for prototype and large data sets, whereas all other sorts have more than zero comparisons and swaps for ordered data. For reverse-ordered data, shell sort operating at an incremental sequence of H = 3\*H + 1 would be the best. This is because this incremental sequence is relatively exponential, and since exponential functions gain speed with time, it makes shell sort operate very efficiently. Merge sort, due to its memory intensity, makes >= the number of comparisons and swaps compared to shell sort, making it slower. Finally, since heap sort operates at 1.5 N log N efficiency, it comes in last place for reverse ordered data, even though heap sort is seen to operate most efficiently when being ran against a reverse ordered list (due to the list being partially in max heap form in reverse order). Lastly, for random data, shell sort at an incremental sequence of H = H / 2 is the best for the prototype data and merge sort was best for the large data set. As explained in my homework 2 report, this is because of merge sort’s dynamic divide-and conqueror approach when compared to shell sort being limited being ran in place. And again, heap sort comes in last place for random data due to its 1.5 N log N efficiency.