Assignment 6 Report

**Figure 1.**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Size 16** | | | | | | | |
| **InOrder** | **Runtime** | **Comparisons** | **Movements** | **ReverseOrder** | **Runtime** | **Comparisons** | **Movements** |
| Alpha | 0 | 15 | 30 | Alpha | 0 | 120 | 150 |
| Beta | 0 | 120 | 0 | Beta | 0 | 120 | 24 |
| Gamma | 0 | 85 | 136 | Gamma | 0 | 72 | 120 |
| Delta | 0 | 150 | 15 | Delta | 0 | 158 | 39 |
| Epsilon | 0 | 46 | 36 | Epsilon | 0 | 55 | 80 |
| Zeta | 0 | 32 | 128 | Zeta | 0 | 32 | 128 |
| **AlmostOrder** | **Runtime** | **Comparisons** | **Movements** | **RandomOrder** | **Runtime** | **Comparisons** | **Movements** |
| Alpha | 0 | 24 | 39 | Alpha | 0 | 65 | 83 |
| Beta | 0 | 120 | 3 | Beta | 0 | 120 | 45 |
| Gamma | 0 | 84 | 135 | Gamma | 0 | 85 | 133 |
| Delta | 0 | 131 | 18 | Delta | 0 | 100 | 66 |
| Epsilon | 0 | 46 | 39 | Epsilon | 0 | 55 | 77 |
| Zeta | 0 | 37 | 128 | Zeta | 0 | 47 | 128 |

**Alpha = Insertion Sort**

1. See Figure 1.

|  |  |
| --- | --- |
| **Size (Random)** | **Runtime** |
| 8192 | 81 |
| 16384 | 135 |
| 24576 | 288 |
| 32768 | 489 |
| 40960 | 743 |
| 49152 | 1063 |
| 57344 | 1409 |

1. O(n2).
2. Alpha’s growth rate is quadratic, like Insertion Sort. Also, as you can see in Figure 1, the comparison count changes based on the input type. This makes sense, since Insertion Sort will compare differently depending on what elements it has sorted thus far. In the case of a perfectly ordered set, it will only do N – 1 comparisons, since all values will be in the correct order. Alpha does this. However, in the case of a reverse order, it will make sum(N - 1) comparisons since every subsequent element will be of lower ordering than those behind it, which Alpha also does. Finally, in the reverse case Alpha makes a lot of movements, as it drags every element all the way back through the array to the beginning.

**Beta = Selection Sort**

1. See Figure 1.

|  |  |
| --- | --- |
| **Size (Random)** | **Runtime** |
| 8192 | 105 |
| 16384 | 305 |
| 24576 | 561 |
| 32768 | 960 |
| 40960 | 1481 |
| 49152 | 2289 |
| 57344 | 2948 |

2. O(n2).
3. Beta’s growth rate is quadratic, so this agrees with Selection Sort. Another big clue is that Beta always has a constant comparison count in Figure 1. This makes sense, since Selection Sort always does sum(N – 1) comparisons, regardless of input type. Another clue is that Beta makes 24 moves when sorting a reversed input. This agrees with Selection Sort, which has to swap N/2 elements in a reversed array. Finally, Beta is slower than Alpha, and Selection Sort is supposed to be slower than Insertion Sort.

**Gamma = Heap Sort**

1. See Figure 1.

|  |  |
| --- | --- |
| **Size (Random)** | **Runtime** |
| 16384 | 1 |
| 32768 | 3 |
| 65536 | 7 |
| 131072 | 15 |
| 262144 | 33 |
| 524288 | 77 |
| 1048576 | 186 |

2. O(nlog(n)).
3. Gamma’s growth rate agrees with Heap Sort’s growth rate, in both the average and the worst case, since it is always O(nlog(n)). Another clue to notice in Figure 1 that Gamma makes slightly less comparisons and moves in the reverse case than in the in order case. This makes sense, since the Heap Sort uses a maxheap, and the reverse order array will have the max value in the first index. Thus, every initial insertion in the heap should not result in many comparisons. Finally, out of all the O(nlog(n)) algorithms given (Gamma, Delta, Epsilon, Zeta), Gamma has the most movements in Figure 1. This makes sense, since copying values to create and sort the heap will cause a lot of movements.

**Delta = Quick Sort (Simple)**

1. See Figure 1.

|  |  |
| --- | --- |
| **Size (Random)** | **Runtime** |
| 16384 | 2 |
| 32768 | 4 |
| 65536 | 7 |
| 131072 | 15 |
| 262144 | 31 |
| 524288 | 67 |
| 1048576 | 147 |
| 1048576 InOrder | Timed Out |

2. O(nlog(n)).
3. Delta has the same growth rate as Quick Sort (Simple). More importantly, Delta has the same worst case, as it takes O(n2) when the input is either correctly sorted or reverse sorted. This agrees with Quick Sort (Simple), since it always picks the first value as the pivot (thus the smallest or largest value in the set). You can also see that it has higher comparison counts in Figure 1 for both in and reverse order. Finally, it has very similar runtimes to Epsilon, which I believe is Quick Sort (Optimized), which makes sense since the two should be similar in average cases.

**Epsilon = Quick Sort (Optimized)**

1. See Figure 1.

|  |  |
| --- | --- |
| **Size (Random)** | **Runtime** |
| 16384 | 1 |
| 32768 | 3 |
| 65536 | 5 |
| 131072 | 11 |
| 262144 | 23 |
| 524288 | 50 |
| 1048576 | 129 |
| 1048576 InOrder | 79 |

2. O(nlog(n)).
3. Epsilon has the same growth rate as Quick Sort (Optimized), at O(nlog(n)). Unlike Quick Sort (Simple), the worst case runtime is not O(n2), and is in fact quite fast since the median pivot point it chooses should actually be a median for the data. Another thing to note is that at small sizes this sort behaves like Insertion Sort (this can be seen by comparing with Alpha in Figure 1).

**Zeta = Merge Sort**

1. See Figure 1.

|  |  |
| --- | --- |
| **Size (Random)** | **Runtime** |
| 16384 | 2 |
| 32768 | 4 |
| 65536 | 9 |
| 131072 | 19 |
| 262144 | 43 |
| 524288 | 89 |
| 1048576 | 206 |

2. O(nlog(n)).
3. The growth rates for Zeta and Merge Sort are the same. Since Merge Sort splits the data and then makes comparisons, it should be faster for both InOrder and ReverseOrder; this can be seen for Zeta in Figure 1. Another good piece of evidence is that Zeta has constant movement counts for all input types in Figure 1. This agrees with Merge Sort, since, regardless of the input, all data must be moved between an auxiliary array the same number of times when recursing.