Part 1

The Big-O efficiency of this loop construct is 4nlog(n)^2. The outer loop has x growing exponentially which makes it head towards the loop condition in logarithmic growth (logn). The inner loop is decrementing y by 1 each iteration (n) and taking 4 milliseconds. Also, n^2 in the outer loop condition turns logn into logn^2. Therefore, if we input n = 1000, we get 4(1000)log(1000)^2 = 80,000.

Total milliseconds: 80,000

Part 2

Prototype Set (1-10)

Ordered Dataset Reverse Dataset Random Order

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Comparisons | Exchanges | Comparisons | Exchanges | Comparisons | Exchanges |
| Bubble Sort | 45 | 0 | 45 | 45 | 45 | 22 |
| Selection Sort | 45 | 9 | 45 | 9 | 45 | 9 |
| Insertion Sort | 0 | 0 | 45 | 45 | 22 | 22 |
| Shell Sort  1023/2 | 0 | 0 | 11 | 11 | 6 | 6 |
| Shell Sort  1093/3 | 0 | 0 | 13 | 13 | 13 | 13 |
| Shell Sort -  Ciura’s Seq. | 0 | 0 | 13 | 13 | 13 | 13 |
| Quick Sort4/3 | 45 | 0 | 25 | 5 | 13 | 8 |

Large Set (1-2000)

Ordered Dataset Reverse Dataset Random Order

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Comparisons | Exchanges | Comparisons | Exchanges | Comparisons | Exchanges |
| Bubble Sort | 1999000 | 0 | 1999000 | 1999000 | 1999000 | 100559 |
| Selection Sort | 1999000 | 1999 | 1999000 | 1999 | 1999000 | 1999 |
| Insertion Sort | 0 | 0 | 1999000 | 1999000 | 100559 | 1000559 |
| Shell Sort  1023/2 | 0 | 0 | 6764 | 6764 | 16973 | 16973 |
| Shell Sort  1093/3 | 0 | 0 | 7284 | 7284 | 20871 | 20871 |
| Shell Sort -  Ciura’s Seq. | 0 | 0 | 8412 | 8412 | 15308 | 15308 |
| Quick Sort | 1999000 | 0 | 1000000 | 1000 | 13935 | 8750 |

We can see that shell sort with a gap sequence of 2\*H+1 starting at 1023 produced better comparisons/exchanges than 3\*H+1 starting at 1093. The higher the starting number for the sequence in both 2\*H+1 and 3\*H+1, the better the results. For example, starting at a 1093 gap in the 2000 data set is superior to starting at 121 in the same 3\*H+1 sequence. Computer scientist Martin Ciura’s shell sort sequence (1,4,10,23,57,132,301,701) performs better than both 2\*H+1 and 3\*H+1 sequence, as seen by the results in the table above.

Shell sort is an augmented insertion sort. Instead of comparing elements with a gap of 1 only, shell starts with a larger gap number and gets smaller following a sequence until it reaches a gap of 1 (insertion sort). Both insertion and shell are in-place sorting algorithms, but shell is not stable. Shell works faster because it moves the lows and highs early on, so that as the gaps decrease it makes the dataset more ordered each time, speeding up the sorting process, which is why insertion sort works faster the more sorted a dataset is.

Quicksort works better than any of the sorts tested here in the random order category. Quicksort's average case Big O is the same as its best case, O(nlogn). The worst case of quicksort is O(n^2) with an already sorted list, because if we pick the first pivot element, and each time there are no values smaller to the left, each time the partition would be split to n-1, n-2, n-3... and so on until it reaches 1. Quicksort is an in-place algorithm that requires no additional space, but it is not a stable sort.