**Comb Sort Analysis**

The sorting algorithm our group decided to use that has not been mentioned in class is the comb sort. The approach to sorting a dataset of numbers with comb sort is very similar to the concept of shell sort where a gap is utilized to traverse through the array of numbers. However, the difference with comb sort is it only compares two values at a time rather than multiple values. In addition, the comb sort does not completely sort the array with every pass like shell sort and only does so when the gap size is small enough. The gap size is determined by dividing the number of elements in the array by 1.3 (vs. shell sort’s 2.2) and continuously doing so until the gap size is 1 where a standard insertion sort is performed. Therefore, the big O notation is more than O(n) since it must iterate the entire array at least once, but less that O(n2 ) since a gap size is utilized to eliminate excessive swaps, which leads to a big O notation of O(nlogn).

\* = Not checked during iteration

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Original (n = 6) | 3 | 6 | 1 | 2 | 7 | 4 |
| Gap = 6/1.3 = 4 | 3 | 4 | \*1 | \*2 | 7 | 6 |
| Gap = 4/1.3 = 3 | 2 | 4 | 1 | 3 | 7 | 6 |
| Gap = 3/1.3 = 2 | 1 | 3 | 2 | 4 | 7 | 6 |
| Gap = 2/1.3 = 1 (insertion sort) | 1 | 2 | 3 | 4 | 6 | 7 |

Unfortunately, parallelizing comb sort isn’t ideal since the algorithm requires the previous iteration to continue sorting effectively. Therefore, the only way to parallelize comb sort is the divide the size of the dataset by the number of processors and utilize comb sort on the subset in each processor, then utilizing merge sort to combine the sorted subsets into the sorted array.