**Stark Industries Sorting Algorithm Experiment**

Stark Industries recently conducted an experiment with 5 different sorting algorithms. The experiment was designed to compare the run times between sorts of different Big O notation. To do this, the members took two O(n^2) sorts, Bubble and Selection, and two O(n log n) sorts, Heap and Merge. One additional sort, shell sort, was also O(n log n) and will be brought up later in the paper.

To start the experiment the group generated one randomly generated unsorted array and one sorted array, both with 6,000 elements respectively. These arrays were loaded into the four sorting algorithms that were located in different projects and timed; the results were recorded on an excel spreadsheet. Each algorithm was tested 10 times, 5 with the unsorted array and 5 with the sorted. An average was taken from the sorted and unsorted versions and was then compared to the other algorithms to test which one was the fastest

For the unsorted array, merge sort was deemed the fastest of them all with an average time of 0.00668776 seconds. Falling second was Heapsort, another fast sorting algorithm that averaged a time of only 0.01269726 seconds. Next came selection sort with 0.06513088 seconds and the slowest, bubble sort, coming in at 0.1560388 seconds.

One of the most surprising discoveries in the experiment were the run times with sorted arrays, more importantly: bubble sort. Bubble sort was extremely fast with an average of 0.00002228 seconds. This was over 0.005 seconds faster than merge sort, which was the fastest when it came to unsorted numbers. Next came Heapsort, which came in slightly slower than last time with 0.0101742 seconds. The slowest of all the algorithms was selection sort, which was 0.06619206 seconds on average.

The findings of this experiment have concluded that the larger that the unsorted array becomes, the more complicated the sorting algorithm must be. Merge Sort, which was one of the most complicated out of the four, processes the 6,000 almost .2 seconds faster than bubble sort. The difference with sorted items though, is that the simpler the algorithm is (in this case bubble sort) the quicker it is able to process the elements.

**Shell Sort**

Shell sort works as a variation of insertion sort, however changing the constant interval variable of one. Shell sort is designed to create ease when performing large shifts in an array, minimizing the elements affected in such sorts. The interval is initialized as one, then found through varying sequences such as its original sequence and Knuth’s, Sedgewick’s, Hibbard’s, Papernov and Stasevich’s, and Pratt’s increments. In each variation, the interval is slowly reduced until the interval is equal to one. For example, the original sequence for shell sort operates on the interval equal to N/2, N/4 …. 1, with N being the number of elements in the array.

In a shell sort operating with the original sequence and containing eight elements, N/2, or 8/2, would produce an interval of 4. The first element, in position 0, would be compared with the fourth element, and the lesser of the two values would be placed in position 0. The process would then continue for the rest of the array, producing a comparison between 1st and 5th positions, 2nd and 6th positions, and 3rd and 7th positions. Since there is no element in position 8, 0 to 7 is 8 elements, and the 4th element has already been compared to that of position 0, the algorithm would continue by reducing the interval from N/2 to N/4, which would be 8/4, producing a new interval of two. The program would repeat the steps above, yet comparing positions 0 and 2, 1 and 3, and so on. Now that the interval is small enough to be applicable both ways, the comparisons are extended to include elements existing before the newly shifted elements. For example, if position 4 and 6 were shifted, the algorithm would then compare positions 4 and 2 to ensure that the values were still in ascending order.

Once the interval of the shell sorting algorithm reaches one, the elements are finally compared to those directly next to them, running one last check to ensure that all elements are sorted properly. This method of reducing the interval slowly helps to prevent large shifts affecting many elements of the array at once. The efficiency of shell sort is determined by the sequence chosen and the number of elements that exist in the array. Each sequence operates in the same way as described above, yet produce different variations in the interval as well as introduce new methods of determining what that interval is.

|  |  |  |  |
| --- | --- | --- | --- |
| Team members | Sorting Algorithm | Approximate time - Unsorted | Approximate time-  Sorted |
| Clayton | Heap Sort | 0.01269726 s | 0.0101742 s |
| Parker | Merge Sort | 0.00668776 s | 0.0046954 s |
| Chase | Bubble Sort | 0.1560388 s | 0.00002228 s |
| Kory | Selection Sort | 0.06513088 s | 0.06619206 s |
|  | Shell Sort | 0.00170266 s | 0.00140298 s |