Prof. Santos

IT 2660 - Fall 2020

Robert Pratt

Assignment 8

Merge Sort vs Quick Sort

The Merge Sort and Quick Sort algorithms are both similar in many respects, with their differences stemming from their opposing methods of performing their sort. Both algorithms will, in their own way, recursively divide the data set into smaller and smaller units until all have been placed into the desired order. In a Merge Sort, this is done by dividing the list down until each list has no more than two nodes, requiring a new list (read array) be created to store each of the values/key fields as the lists are divided. Then these lists are merged back into one as the algorithm sorts each node into its proper location, potentially resorting the list with each merger as it works back out of its recursive function. For the Quick Sort, instead of dividing the list down into smaller and smaller units, the nodes are sorted within the list instead of requiring memory to copy all the nodes into external lists.

Because it doesn’t copy actually divide the list into smaller ones, it selects a node in the middle and “pivots” the nodes directly around it into their relative positions, making all values to the nodes left smaller than it and those on the right larger than it. The rest of the Quick Sort’s algorithm is very similar to the merge part of the Merge Sort’s; it continues to move recursively through the left and right sides of the list to place them into their correct positions. Because it does not require the data to be copied to external lists, there is very little extra memory needed for the Quick Sort

For the Merge Sort, the best case would be were the entirety of one sublist is less than that of the other; the worst case, however, would be when which node is highest alternates between the nodes of each list. In the best case scenario, this algorithm would only need *n*/2 passes to sort all of the nodes. The worst case, however, would require *n*-1 passes to sort all of the nodes, with one final partial pass to place the *n*th node into the last location. For the Quick Sort, the best case scenario would be for the pivot values/nodes to always be in the middle of the partitions, and the worst scenario would be for the pivot values/nodes to always be to the far side of the data list. In the best case, the Quick Sort would need only *log*2(*n*-1) passes, but the worst case would require *n* passes to ensure the nodes are sorted correctly.

The Merge Sort algorithm does well in all situations, except for where data overhead must be kept to a minimum. This algorithm would would also potentially not perform as well with small data sets as the Quick Sort given that the Merge Sort must divide the lists into smaller lists, copying them into temporary lists, and then reassemble the lists as the nodes are sorted. The Quick Sort performs well on most data sets, especially smaller ones, but will potentially not perform as well on larger data sets when compared to the Merge Sort.

The Bubble Sort algorithm, or an O*n*2 algorithm, could be useful when a mostly-sorted data set needs to be quickly checked, sequentially, with very little adjustments and overhead needed. I have seen suggestions of quick checks like this being used in game develop where a Bubble Sort-esque algorithm to verify there that the sprites have not shifted to incorrect positions.