ALI SALEHI ASGN5 WRITEUP.pdf

BUBBLE SORT

Bubble sort performs by comparing two adjacent elements in an array, and swapping them if they are in the wrong order. It is an effective sort, and one of the easiest to understand; however it is one of the slowest because of the amount of moves it makes. Bubble sort has to move a single element multiple times, slowly progressing and moving the greatest elements to the end of the array, and thus has a time complexity of O(n^2). It is tied for the slowest with binary insertion sort.

BINARY INSERTION SORT

Insertion sort operates by iterating through the array from left to right, comparing the current element to all the elements to the left of it, and placing it at the correct location. This is expensive because it needs to iterate through the entire array, and then iterate again through all the elements to the left, progressively getting larger. Like bubble sort this sorting algorithm is simple yet slow, and also has a time complexity of O(n^2) because of the number of iterations it must do.

SHELL SORT

Shell sort is an improvement to insertion sort that decreases the number of insertions necessary by comparing elements at a varying length determined by an array of gaps. This allows elements that are far out of order to become closer to their final destination, helping insertion sorting because insertion performs much faster on an array that is almost sorted. The gap array in shell sort is crucial to the time complexity, and for this lab I chose to decrement the gap size by (size * 5 / 2) like the example provided, though the gap size can also decrement by a factor of 2, 1.5, or any other value. Finding an efficient gap size is important because two few gaps forces the insertion sort to do too much work, whereas two many gaps is redundant and uses too much time. An optimized shell sort can perform with a time complexity of O(n^5/3).

QUICK SORT

Lastly, quick sort is often the most used sorting algorithm because of its consistent speed and ability to avoid the worst case time complexity. Quick sort utilizes a divide and conquer approach that divides the original array into smaller sub arrays, and sorts these through the same process. Because of the repetitive nature of divide and conquer, the quick sort algorithm is recursive, and calls a partition function that uses a pivot to determine whether elements are less than or greater than the pivot, and then sorts them accordingly. Because the algorithm gradually divides the array into smaller pieces, it has a time complexity of O(NlogN). Its speed can be seen through our lab

experiments, as quick sort consistently has the least amount of moves.

AUXILARY FUNCTIONS

For this lab I chose to implement a few functions that would help my testing process and streamline working with dynamic arrays. I included an init() function that dynamically creates the array using calloc() and initializes it with random values. I also included a print() function to print the array that was helpful for testing and to limit repetitive print statements. Finally, I included a swap() function because we swap elements in all of the sorting algorithms, and this also helped me keep track of how many moves each sorting algorithm took.