

<Explanation of Sorting Algorithms>

Bubble Sort: In bubble sort, the array of numbers is divided into an unsorted section and a sorted section. The unsorted section is on the left-hand side (front of the array) and initially consists of the entire array, while the sorted section is on the right-hand side (end of the array) and is initially empty. We iterate through the integer array starting from the leftmost element (index 0), comparing two adjacent elements at a time. If the element on the left is greater than the element on the right, the two elements are swapped. After each iteration, the index for the last element of the unsorted array is reduced by 1 because the newly sorted element should be excluded from the comparisons in the future iterations. Finally, to increase the efficiency of bubble sort, I declared a Boolean variable that records whether any swapping has occurred during each iteration through the integer array. If an iteration did not have any swapping event, then the array is sorted.

Insertion Sort: Unlike bubble sort, insertion sort places the sorted section on the left-hand side and the unsorted section on the right-hand side. In each iteration, the leftmost element in the unsorted section is added to the sorted section in the right spot. Iteration begins with the second element (index 1) because the first element is considered to be already sorted, and the element to be inserted (declared as `insertionItem`) is compared with the left adjacent elements in the sorted section one by one. If `insertionItem` is less than the left adjacent element and we have not reached the front of the array, the left element is shifted to the right. If `insertionItem` is greater than the left adjacent element or it reaches the front the array, it is inserted into that current position. After each iteration, the number of sorted elements increases by 1, so the index for the last element of the sorted array and the maximum number of comparisons increments after each iteration.

Heap Sort: We first converts the original array into a heap by calling `BuildHeap()`, in which we iterate through each internal node and percolate down so that each parent node consists only of child nodes that are smaller. After the array is heapified, we iterate through the array starting from the last element and call `deleteMax()`, which swaps the values of the current node with the root node because it the root node is guaranteed to carry the maximum value. The element swapped into the root node is then percolated down to its appropriate position. Note that there are two percolate down functions in my code (`percolateDownBuildHeap()` and `percolateDownHeapSort()`) because when percolating down during sorting, we need to adjust the index of the last element to exclude the already sorted elements. The algorithm logic of the two functions is the same, however.

Merge Sort: I created the more efficient version of Merge Sort, which uses a clone of the original array (line 192~195) which we will alternate back and forth during the sorting process. In `switchingMergeSort()`, we calculate the middle index of the array and recursively call `switchingMergeSort()` on the two halves of the unsorted array. This recursion continues until each subarray contains a single element. Then, the 2 subarrays are merged into a sorted array by calling `switchingMerge()`. In `switchingMerge()`, the array passed in as C is used just for reference, and the array passed in as D the final array that will carry the sorted array. We declare two integers i and j that each point to the first element of each subarray within the assistant array C. These two elements are compared, and the smaller one is inserted into array D. If either subarray is exhausted, then no more comparisons are needed, and the remaining elements in the other subarray are inserted one by one to the array D. Note that the "subarrays" mentioned here are not two separate arrays, instead the integers i and j both point to elements in array C that would correspond to two different subarrays in a normal Merge Sort. This version of Merge Sort is more efficient because it alternates back and forth between the original array A and cloned array B, eliminating the need to copy the sorted array from a temporary array to the original array during each merge step.

Quick Sort: In `DoQuickSort()`, a "pivot element" in the integer array is first selected, which is used as a standard for comparison. On the initial call of `DoQuickSort()`, the pivot element is usually set as the last element of the original array. The array is then partitioned based on this pivot element by calling `Partition()`, where each of the remaining elements in the array are compared to the pivot element and placed to the left if its value is smaller, and to the right if its value is greater. After partitioning is done, the function returns the index of the pivot element, and `DoQuickSort()` is called recursively on the two partitioned arrays on the left and right of the pivot element.

Radix Sort: In Radix Sort, elements are sorted by comparing the place values of the elements starting from the least significant digit to the most significant digit. For this, we first need to find the element with the greatest value because we need to know how many significant places to iterate through.

The accessory sorting mechanism used for sorting in each significant place is Counting Sort, a stable sorting algorithm that works by counting the number of times that each unique element appears in the array. In this counting array, each unique element of the array to be sorted is used as the index value, under which each count is stored. In this case, the unique elements are values between 0 (inclusive) and 10 (exclusive) because we are comparing elements based on place values. We then iterate through each element of the counting array, reassigning each element as the cumulative sum of all values to the left. Finally, the index of each element in the original array is found in the counting array, through which we can derive the position of the element in the sorted array.

Since we must consider both negative and positive values, I separate the negative elements and positive elements are separated into 2 different arrays, apply Radix Sort on each array individually, then merge them at the end.

<Time Complexity Comparison & Analysis>

Comparison Between Various Sorting Algorithms

Figure 1. Comparison of Average Time Taken (ms) for Various Sorting Algorithms of 20 Trials. The range of elements given were 200000000 ~ 2000000000 to include as much of the integer data type range as possible.

	Array Size (number of elements)					Array Size (number of elements)					Array Size (number of elements)			
Bubble Sort	10 ³	10 ⁴	10 ⁵	10 ⁶	Insertion Sort	10 ³	10 ⁴	10 ⁵	10 ⁶	Heap Sort	10 ³	10 ⁴	10 ⁵	10 ⁶
Minimum	0	157	14354	N/A	Minimum	0	0	997	N/A	Minimum	0	0	24	231
Maximum	16	235	15872	N/A	Maximum	16	79	1159	N/A	Maximum	1	16	78	359
Average	6.8	190.15	14998.2	N/A	Average	5.3	27.85	1067.3	N/A	Average	0.05	5.6	60	286.6
Standard Deviation	7.367068	23.78473	416.659	N/A	Standard Deviation	7.13848	15.95809	37.92111	N/A	Standard Deviation	0.223607	7.264912	10.49311	30.35301

	Array Size (number of elements)					Array Size (number of elements)					Array Size (number of elements)			
Merge Sort	10 ³	10 ⁴	10 ⁵	10 ⁶	Quick Sort	10 ³	10 ⁴	10 ⁵	10 ⁶	Radix Sort	10 ³	10 ⁴	10 ⁵	10 ⁶
Minimum	0	0	47	110	Minimum	0	0	31	100	Minimum	16	15	78	180
Maximum	16	16	71	236	Maximum	1	16	68	176	Maximum	41	54	171	361
Average	1.25	4.1	54.8	169.4	Average	0.1	2.15	48.5	139.5	Average	29.5	35.9	114.9	271
Standard Deviation	3.905125	6.734827	8.211673	25.9684	Standard Deviation	0.307794	5.294237	7.990125	21.52722	Standard Deviation	5.89826	12.80995	23.77726	52.25746

Figure 2. Known Time Complexity Data for the Sorting Algorithms in Figure 1

	Time Complexity		
	Worst Case	Average Case	Best Case
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Heap Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Quick Sort	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Radix Sort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

From Figure 1, it can be seen that the time taken for Bubble Sort and Insertion Sort increases much more rapidly with increasing array size compared to the other sorting algorithms. This reflects their average time complexity of $\Theta(n^2)$ because both Bubble Sort and Insertion have a nested looped inside another for loop. Of these two sorting algorithms, however, we can see from my results that the average time taken for Insertion Sort increases at a slower rate than Bubble Sort. This is likely because of the difference in time complexity of the best case between the two sorting algorithms. The best-case time complexity for Bubble Sort is $\Theta(n^2)$ which is the same as the worst and average case, but the best-case time complexity for Insertion Sort is $\Theta(n)$ which occurs when an already-sorted array is given. Hence, if some of the test arrays were more sorted, then insertion sort would have been faster than bubble sort, giving a lower average time taken.

The more advanced sorting algorithms Heap Sort, Merge Sort, and Quick Sort increases at a significant slower rate with increasing array size compared to Bubble Sort and Insertion Sort. This reflects their average time complexity of $\Theta(n \log n)$, which has a much steeper slope than n^2 . If we look at the average running time for an array with size 10^6 , Quick Sort is noticeably the fastest, followed by Merge Sort and Heap Sort. This shows that with increasing array size, Quick Sort becomes much more efficient in the average case.

In my experiment, Radix Sort took longer time than Heap Sort, Merge Sort, and Quick Sort. In theory, Radix Sort should be faster with an average time complexity of $\Theta(n)$, but this was not the case in my experiment, which may be because my code for Radix Sort sorts the negative elements and positive elements separately in two arrays and undergoing a merge step.

Quick Sort – Comparison Between Worst Case and Average Case

It was mentioned above that Quick Sort is very efficient with increasing array size assuming the average case, which has a time complexity of $\Theta(n \log n)$. However, in the worst case, Quick Sort can become as slow as selection sort with a time complexity of $\Theta(n^2)$. This occurs when all the elements in the array are identical or the array is already sorted, because in these cases, we get two highly unevenly partitioned subarrays. The following table shows the comparison of time taken (ms) by Quick Sort to sort arrays either with all identical elements or random elements. In both conditions, I inputted an array of size 10^5 and ran 20 trials for each.

	All Identical Elements	Random Elements
Minimum	11	0
Maximum	43	16
Average	29.5	4.15
Standard Deviation	6.881133628	6.702797923

Heap Sort – Comparison Between Best Case and Average Case

It was also shown above that Heap Sort, on the other hand, is the slowest out of the three sorting algorithms with an average time complexity of $\Theta(n \log n)$. However, Heap Sort can become very efficient in the best case, giving a time complexity $\Theta(n)$. This occurs when the array carries all identical elements because in this case, percolateDown only takes 1 step, hence it is proportional to n . The table below shows a comparison in time taken by Heap Sort in the two different conditions. In both conditions, I inputted an array of size 10^4 and ran 20 trials for each.

	All Identical Elements	Random Elements
Minimum	0	15
Maximum	16	34
Average	9.4	29
Standard Deviation	7.123201527	5.08920426