Author: Thomas D. Robertson II

Assignment: Empirical Algorithmic Analysis

Due Date: 10/27/2023

Description of project

In this assignment I will be comparing the algorithmic efficiency of nine different sorting methods. The efficiency of the algorithms will be tested against three different arrays, A randomly generated array, an increasing array, and a decreasing array. The size of the array will be tested on a 10^k magnitude scale, where K is input by the user. Five magnitude levels will be tested, which is up to 10^5, which will result in 100,000 numbers being sorted for each array. Each sorting method is ran a total of five times for each array, at each magnitude level. The results will be timed and the average of the five runs will be taken and used for analysis. This will help account for any system environment variance. The highest number generated in each run is the magnitude of k, so for a run of 10^5, the highest number will be 100,000 at 6 digits.

Expected Results

The nine sorting methods used are as follows: Shell Sort, Insertion Sort, Selection Sort, Bubble Sort, Quick Sort, Merge Sort, Heap Sort, Radix Sort, and Bucket Sort. My implementation of quick sort and merge sort are non-recursive because of runtime size limitations in the python interpreter environment. See the below table for expected runtime efficiencies of each sorting method based on known algorithmic complexity of the methods.

Sorting Method	Average Case	Best Case (Already Sorted)	Worst Case (Reverse Sorted)
Shell Sort	$n^{1.5}$	n log n	$n^{1.5}$
Insertion Sort	n^2	n	n^2
Selection Sort	n^2	n^2	n^2
Bubble Sort	n^2	n	n^2
Quick Sort	n log n	n log n	n^2
Merge Sort	n log n	n log n	n log n
Heap Sort	n log n	n log n	n log n
Radix Sort*	nk	nk	nk
Bucket Sort	$n + n^2/b + b$ **	n + b ***	n^2 ****

*For Radix Sort, n is the number of integers, and k is the number of digits in the maximum integer.
** Where b is the number of buckets, and assumes input is uniformly distributed across the buckets (becomes O(n) when b = n).
*** When each element is placed in its own bucket.
**** When each element is placed into a single bucket.

When looking at sorting method efficiencies, there are a couple of things to keep in mind. For sorting methods with quadratic efficiency (such as n^2), as the input size of n increase, the number of actions required to compare, and sort increase proportionally to the square of n. In other words, if the input of the list is size 10^5, or 100,000 elements, the sort would require an order of $(10^5)^2 = 10^{10}$, or 10,000,000,000 actions. So quadratic sorting methods are not suitable for large input sizes.

The required actions of Linear Complexity sorting methods, O(n), grow at a rate directly proportional to input size. So an input size of 10^5, or 100,000 elements, would require 100,000 actions to sort. Linearithmic Complexity, O(n log n) grows faster than linear methods, but much slower than quadratic methods. For example, an input size of 100,000 elements would require approximately 100,000 * log(100,000) or 500,000 actions to sort as log(100,000) = 5.

For Radix Sort, O(n*k) where n is the number of elements to be sorted, and k is the number of digits in the maximum number, the expected time complexity at the highest magnitude will be 100,000 * 6 or 600,000 actions for a sort on 10^5 values, which maintains a linear growth rate. For Bucket Sort, my implementation creates a bucket for each element in the array, so the average case efficiency for bucket sort would be 100,000 + ($100{,}000^2$/100,000) + 100,000 or 300,000 actions for 100,000 elements for the average scenario. If the randomly generated list has a lot of repeating numbers, multiple values will be assigned to the same buckets which will decrease performance.

As a general statement, using the knowledge that linear sorting methods are faster than linearthmic, and linearthmic sorting methods are faster than quadratics. I would expect the Fastest sorting methods to be as follows when sorting 10^5 elements:

Average Cases (randomly generated):
Bucket Sort(variable) > Quick Sort = Merge Sort = Heap Sort > Radix Sort > Shell Sort > Insertion Sort = Selection Sort = Bubble Sort

Best Cases (already sorted)
Insertion Sort = Bubble Sort > Bucket Sort > Shell Sort = Quick Sort = Merge Sort = Heap Sort > Radix Sort > Selection Sort

Worst Case (sorted in reverse)
Merge Sort = Heap Sort > Radix Sort > Shell Sort > Bucket Sort = Insertion Sort = Selection Sort = Bubble Sort = Quick Sort

Radix Sort would surpass the efficiency of linearthmic sorting methods at smaller values of k. And there will certainly be variance in the results that will not completely match the expected outcomes. After all, I can run my sorting program many times and get differing results for each section purely from changing environmental stress on my operating system at any given moment. So, approximations of expected results are what we are primarily looking at for this analysis.

Description of Sorting Methods

1. Shell Sort – Shell Sort is an in-place comparison sort that starts by sorting pairs of elements far apart from each other, then progressively reducing the gap between elements to be compared. In my implementation, the initial gap size is set to half of the array's length, and it is reduced by half in each iteration until it becomes 0. During each iteration, a modified insertion sort is performed where elements are moved by the current gap size.

2. Insertion Sort builds the final sorted array (or list) one item at a time. It is much less efficient on large arrays but performs well for small arrays. In my implementation, it iteratively takes the next element from the unsorted part of the array and places it into its correct position in the sorted part of the array by shifting elements.

3. Selection Sort is a comparison-based sorting algorithm that divides the input into a sorted and an unsorted region, and repeatedly selects the smallest (or largest) element from the unsorted region and moves it to the sorted region. My implementation reflects this approach, iterating through the array to find the minimum element and then swapping it with the first unsorted element.

4. Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. My implementation includes an optimization that breaks out of the loop if no swaps are performed in a pass, indicating that the array is already sorted and increasing efficiency in the best-case scenario.

5. Heap Sort is a comparison-based sorting technique based on a Binary Heap data structure. It is
   like selection sort where we first find the maximum element and place the maximum element at the end. In my implementation, I repeatedly build a max heap and extract the maximum element from the heap, placing it at the end of the array. Uses the “heap” helper function.

6. Quick Sort is a divide-and-conquer algorithm that selects a pivot element from the array and partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. My implementation does this in a non-recursive manner using a stack to keep track of the sub-arrays that need to be sorted. A non-recursive method was required due to size limitations of the python interpreter.

7. Merge Sort is an efficient and stable, comparison-based, divide and conquer sorting algorithm. My implementation is an non-recursive iterative version, where I repeatedly merge sub-arrays of increasing size until the entire array is sorted. I use a helper function

"Merge" to merge two sorted sub-arrays. A non-recursive method was required due to size limitations of the python interpreter.

8. Radix Sort is a non-comparative integer sorting algorithm that sorts numbers by processing individual digits. My implementation sorts the array of integers digit by digit starting from the least significant digit to the most significant digit. It uses "counting sort" as a subroutine to sort digits.

9. Bucket Sort works by distributing the elements of an array into several buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sort algorithm. In my implementation, I divide the range of inputs into equal-sized sub-ranges (or buckets), and then sort the elements in each bucket using the built-in sort. The buckets are then concatenated to get the sorted array.

Required Helper Functions

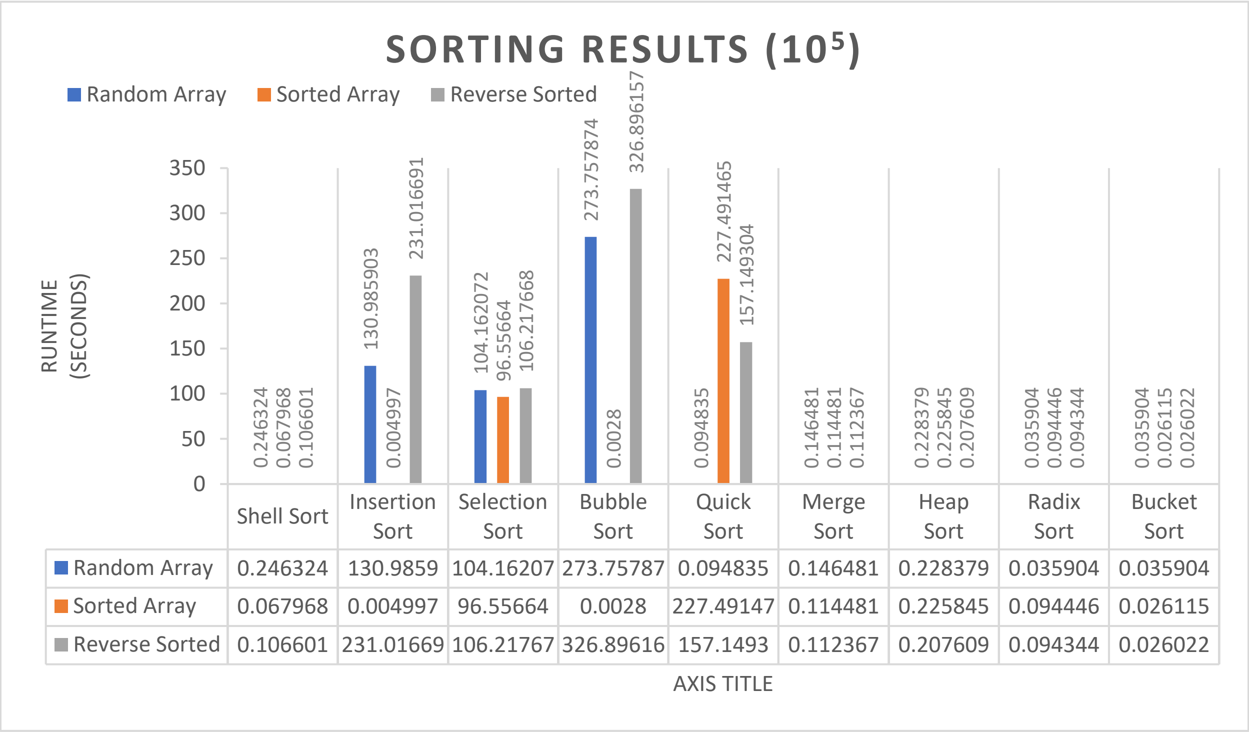
1. heapify: This is a helper function for the heap sort to maintain the heap property.
2. partition: This is a helper function for the quicksort algorithm, used to partition the array.
3. merge: This is a helper function for the iterative merge sort, used to merge two sorted sub-arrays.
4. counting sort: This is a subroutine used by radix sort to sort digits based on their significance.

Empirical Results

The results of the highest order magnitude runs are displayed in the Chart below which uses the table results provided near the end of the document for every run. From the chart we can see that my actual results almost completely match the expected results for the algorithmic efficiency of each sorting method. The only sorting methods that meaningfully increase in time are the quadratic methods, which is to be expected given the high growth rate of actions needed to sort the given inputs.

At the current growth rate, trying to sort the arrays at a magnitude of 10^6 would take over a day. Anything higher is simply infeasible. Conversely, the linear and linearthmic sorting methods perform exponentially well in comparison to the quadratic functions. Those methods could easily handle larger sizes of k values, while the quadratic functions would be crippled in attempting to sort any large dataset. This wastes far to much time and hogs to many resources in a system to be viable.

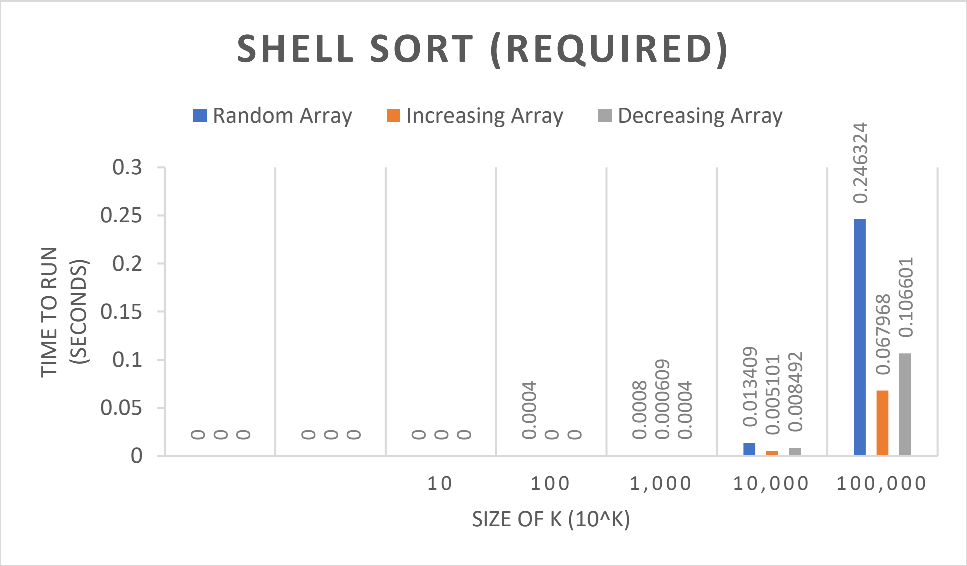
The results match expectations that large data sets should be sorted using sorting methods that are more efficiently designed for problems of higher magnitude. Using quadratic sorting methods outside of extremely small input sizes is inefficient, to infeasible. The proper sorting algorithm for an application should be chosen to match the needs and expectations of a project. This assignment shows the importance of researching and implementing effective solutions to a problem.



	Shell Sort	Insertion Sort	Selection Sort	Bubble Sort	Quick Sort	Merge Sort	Heap Sort	Radix Sort	Bucket Sort
■ Random Array	0.246324	130.9859	104.16207	273.75787	0.094835	0.146481	0.228379	0.035904	0.035904
■ Sorted Array	0.067968	0.004997	96.55664	0.0028	227.49147	0.114481	0.225845	0.094446	0.026115
■ Reverse Sorted	0.106601	231.01669	106.21767	326.89616	157.1493	0.112367	0.207609	0.094344	0.026022

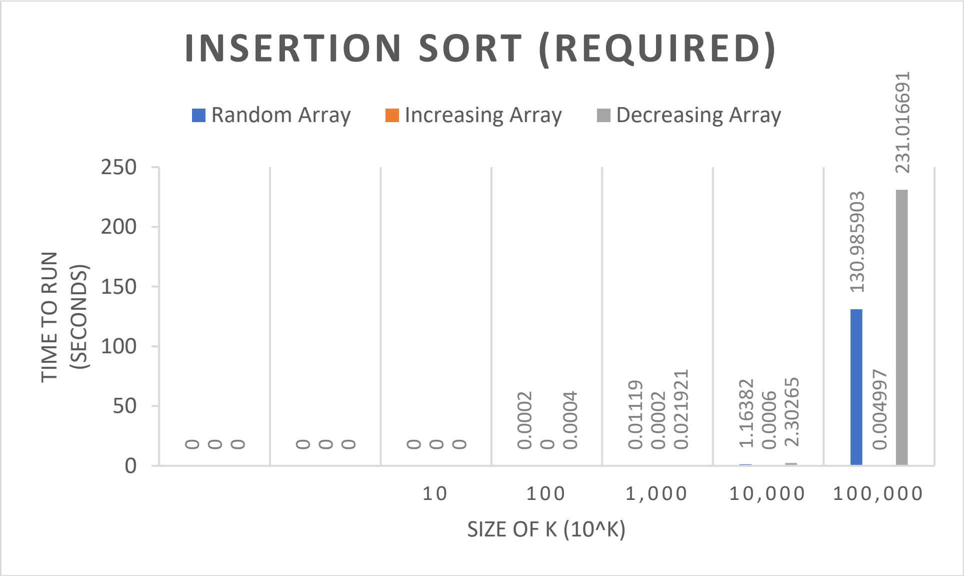
See the below Graphs and data tables for runtime values of each sorting method averaged across five runs.

***NOTE Factor increase in time from 0.00000 seconds to first trackable change is not being considered as it cannot be accurately assessed at the listed precision levels, in addition to variance caused by system resource usage.



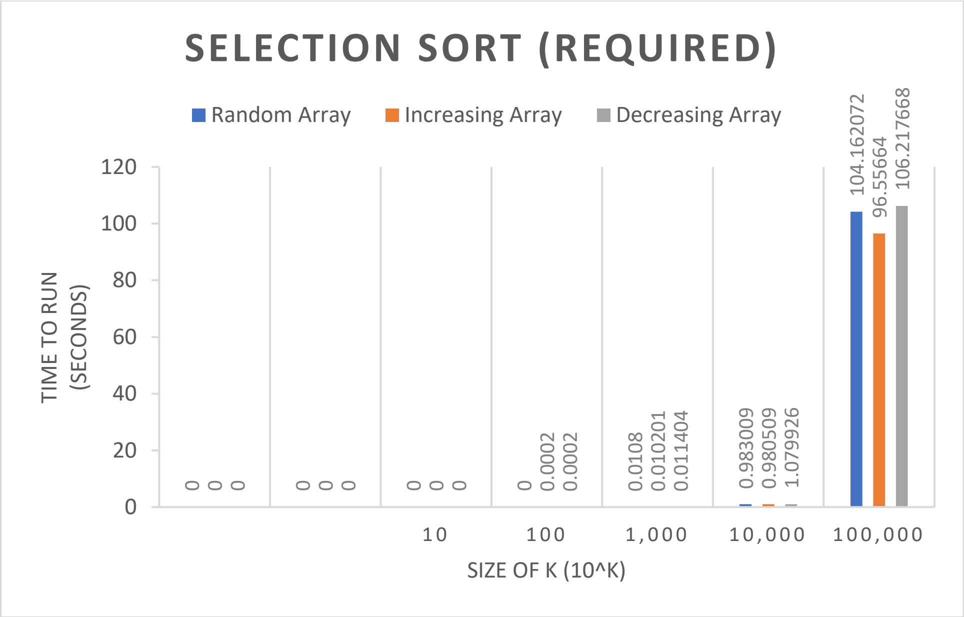
Shell Sort (Required)

Size of k	Random Array Avg time to complete (seconds)	Increasing Array Avg time to complete (seconds)	Decreasing Array Avg time to complete (seconds)	Factor Increase in time 1ST, 2ND, 3RD
10	0.000000	0.000000	0.000000	N/A
100	0.000400	0.000000	0.000000	N/A
1,000	0.000800	0.000609	0.000400	2, N/A, N/A
10,000	0.013409	0.005101	0.008492	16.8, 8.4, 21.2
100,000	0.246324	0.067968	0.106601	18.3, 13.3, 12.6



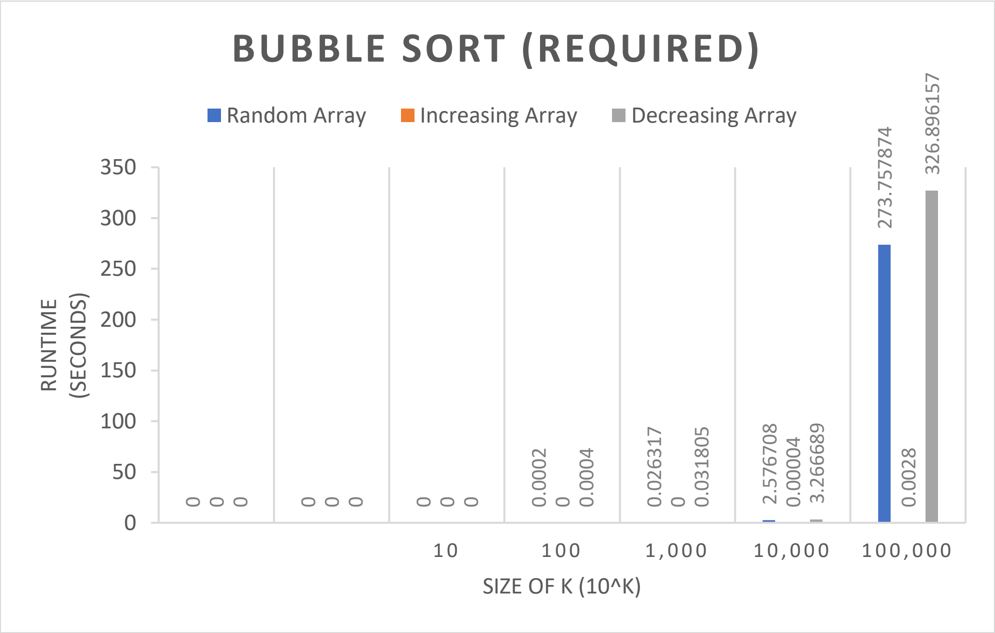
Insertion Sort (Required)

Size of k	Random Array Avg time to complete (seconds)	Increasing Array Avg time to complete (seconds)	Decreasing Array Avg time to complete (seconds)	Factor Increase in time
10	0.000000	0.000000	0.000000	N/A
100	0.000200	0.000000	0.000400	N/A
1,000	0.011190	0.000200	0.021921	56.0, N/A, 54.8
10,000	1.163820	0.000600	2.302650	103.9, 3, 105.1
100,000	130.985903	0.004997	231.016691	112.6, 8.3, 100.3



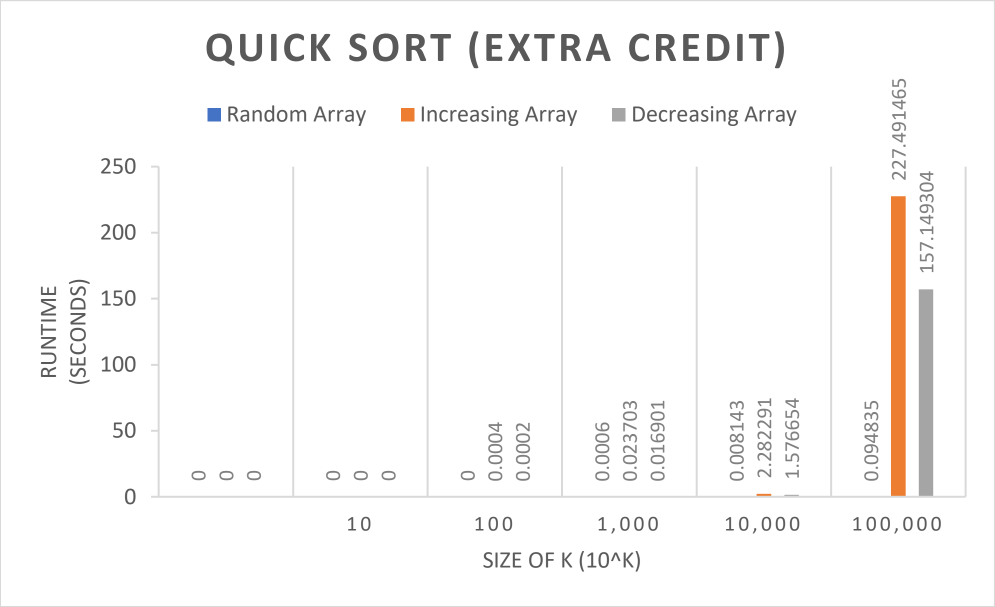
Selection Sort (Required)

Size of k	Random Array Avg time to complete (seconds)	Increasing Array Avg time to complete (seconds)	Decreasing Array Avg time to complete (seconds)	Factor Increase in time (approximation)
10	0.000000	0.000000	0.000000	N/A
100	0.000000	0.000200	0.000200	N/A
1,000	0.010800	0.010201	0.011404	N/A, 51.0, 57.0
10,000	0.983009	0.980509	1.079926	91.0, 96.1, 94.7
100,000	104.162072	96.556640	106.217668	106.0, 98.5, 98.3



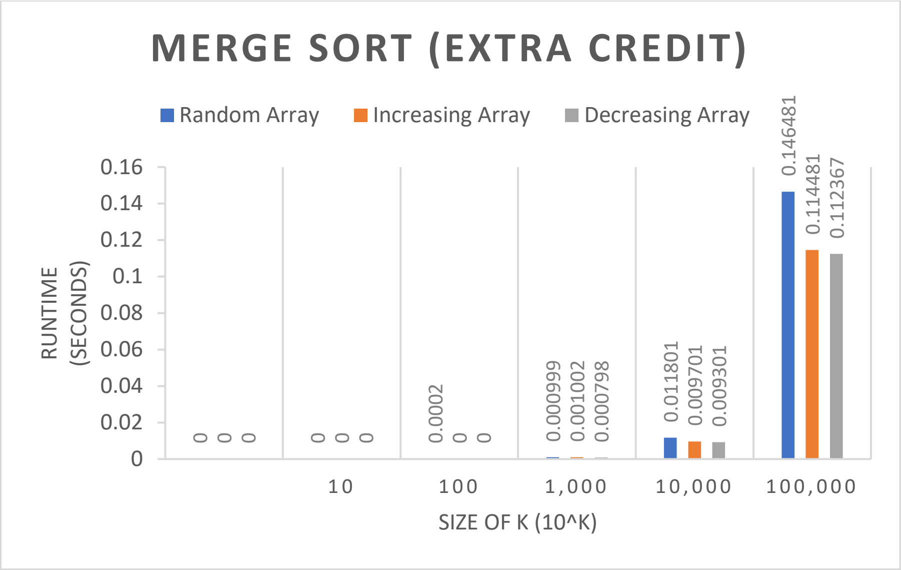
Bubble Sort (Required)

Size of k	Random Array Avg time to complete (seconds)	Increasing Array Avg time to complete (seconds)	Decreasing Array Avg time to complete (seconds)	Factor Increase in time (approximation)
10	0.000000	0.000000	0.000000	N/A
100	0.000200	0.000000	0.000400	N/A
1,000	0.026317	0.000000	0.031805	131.6, N/A, 79.5
10,000	2.576708	0.000040	3.266689	97.9, N/A, 102.7
100,000	273.757874	0.002800	326.896157	106.23, 70, 100.0



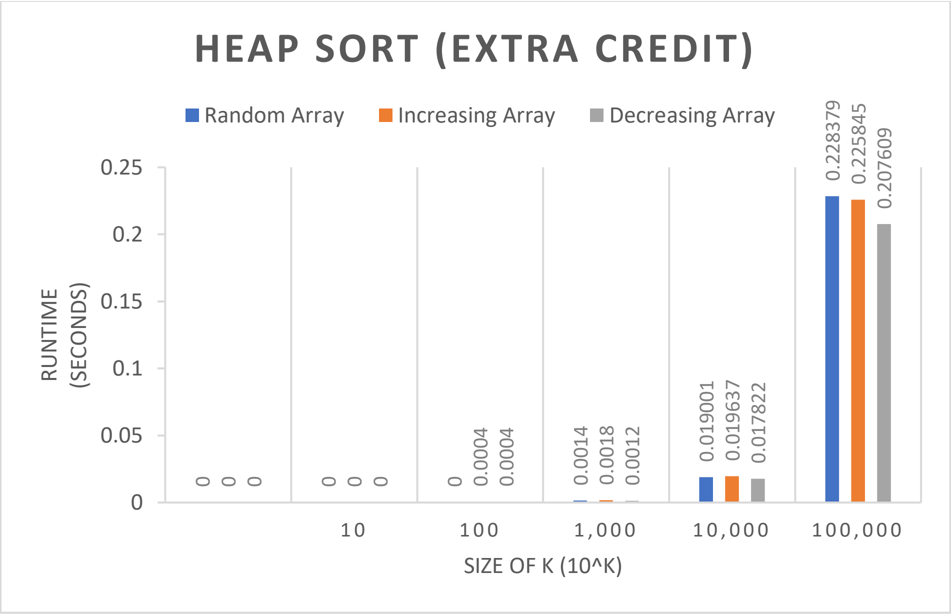
Quick Sort Non-Recursive (Extra Credit)

Size of k	Random Array Avg time to complete	Increasing Array Avg time to complete	Decreasing Array Avg time to complete	Factor Increase in time
10	0.000000	0.000000	0.000000	N/A
100	0.000000	0.000400	0.000200	N/A
1,000	0.000600	0.023703	0.016901	N/A, 59.3, 84.5
10,000	0.008143	2.282291	1.576654	13.6, 96.3, 93.3
100,000	0.094835	227.491465	157.149304	11.7, 99.7, 99.7



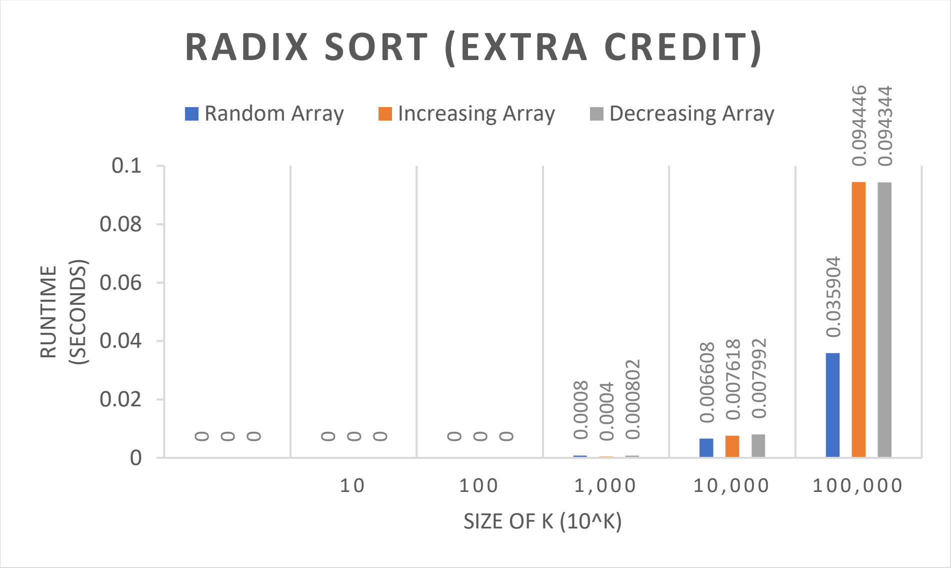
Merge Sort Non-Recursive (Extra Credit)

Size of k	Random Array Avg time to complete	Increasing Array Avg time to complete	Decreasing Array Avg time to complete	Factor Increase in time
10	0.000000	0.000000	0.000000	N/A
100	0.000200	0.000000	0.000000	N/A
1,000	0.000999	0.001002	0.000798	5.0, N/A, N/A
10,000	0.011801	0.009701	0.009301	11.8, 9.7, 11.7
100,000	0.146481	0.114481	0.112367	12.4, 11.8, 12.1



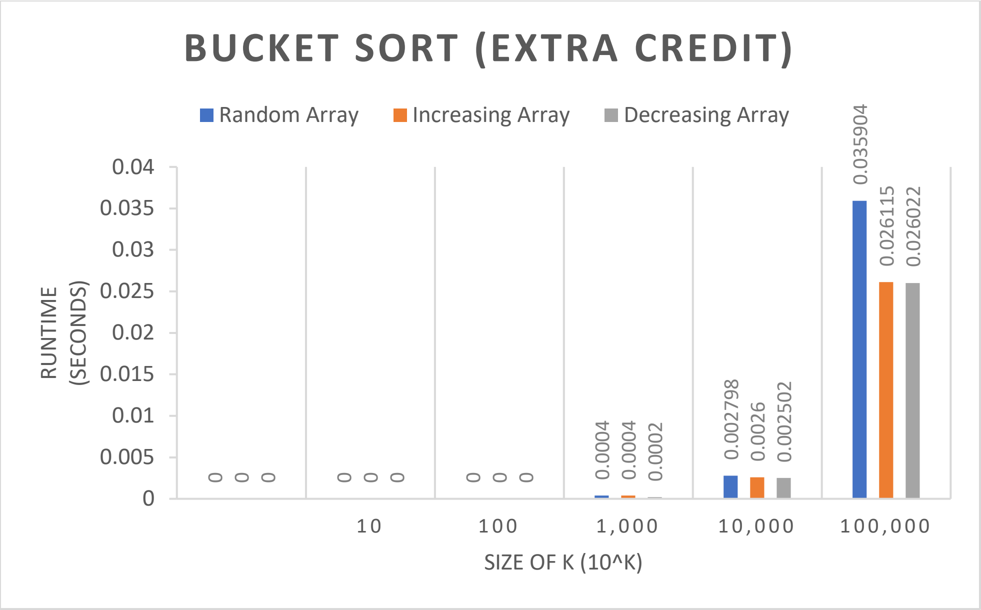
Heap Sort (Extra Credit)

Size of k	Random Array Avg time to complete	Increasing Array Avg time to complete	Decreasing Array Avg time to complete	Factor Increase in time
10	0.000000	0.000000	0.000000	N/A
100	0.000000	0.000400	0.000400	N/A
1,000	0.001400	0.001800	0.001200	N/A, 4.5, 3.0
10,000	0.019001	0.019637	0.017822	13.6, 10.9, 14.9
100,000	0.228379	0.225845	0.207609	12.0, 11.5, 11.6



Radix Sort (Extra Credit)

Size of k	Random Array Avg time to complete	Increasing Array Avg time to complete	Decreasing Array Avg time to complete	Factor Increase in time
10	0.000000	0.000000	0.000000	N/A
100	0.000000	0.000000	0.000000	N/A
1,000	0.000800	0.000400	0.000802	N/A
10,000	0.006608	0.007618	0.007992	8.3, 19.0, 10.0
100,000	0.035904	0.094446	0.094344	5.4, 12.4, 11.8



Bucket Sort (Extra Credit)

Size of k	Random Array Avg time to complete	Increasing Array Avg time to complete	Decreasing Array Avg time to complete	Factor Increase in time
10	0.000000	0.000000	0.000000	N/A
100	0.000000	0.000000	0.000000	N/A
1,000	0.000400	0.000400	0.000200	N/A
10,000	0.002798	0.002600	0.002502	7.0, 6.5, 12.5
100,000	0.035904	0.026115	0.026022	12.8, 10.0, 10.4