2017

An Evaluation of Sorting Algorithms

TCSS 558 – MASTER’S SEMINAR

JESSE BANNON

Description of Sorting Algorithms

Insertion Sort

Insertion sort constructs the sorted array by moving single elements right-to-left, comparing and moving itself left until it has satisfied its ordering condition. For example, suppose our list is [0, 2, 3, 1, 4]. [1] moves itself to the left until it's condition is satisfied:
[0, 2, 3, 1, 4]: (1 < 3) -> [0, 2, 1, 3, 4]: (1 < 2) -> [0, 1, 2, 3, 4]: (1 > 0)
The algorithm has a O(n^2) runtime complexity, and is computed in-place, which results in a O(n) space complexity.

Selection Sort

Selection sort constructs the sorted array one element at a time starting on one side and progressing to the other side. For example, when sorting in ascending order from left-to-right, we start at index zero. We search every element in the list to find the min element in [0, n), and swap that element to reside in index zero. We repeat this process for [1, n), [2, n), …, [n-1, n) to sort the list. The algorithm has a O(n^2) runtime complexity, and is computed in-place, which results in a O(n) space complexity.

Bubble Sort

Bubble sort naively iterates the entire list and swaps elements a single space per pass if the ordering condition does not satisfy a neighboring element until no swaps are performed in an iteration. The worst case is if the entire list is sorted except the last element should be the first element. To move the last element a single space, the algorithm must iterate the entire list. The algorithm has a O(n^2) runtime complexity, and is computed in-place, which results in a O(n) space complexity.

Merge Sort

Merge sort is a deterministic algorithm where it partitions the data in a binary tree-like structure. A top-down merge sort recursively partitions the data in two until there are only one or two elements in each partition, then constructs a sorted partition by iterating the two sorted sub-partitions. This algorithm has a O(nlog(n)) runtime complexity, but requires an additional list of size n to construct sorted partitions, which results in a O(2n) = O(n) space complexity.

Quick Sort

Quick sort is similar to merge sort in the sense that it recursively partitions the data in two. Instead of splitting equally in two, quick sort chooses a pivot element, and places elements less than the pivot element on the left and elements greater than the pivot element on the right. Once you have two partitions that are < pivot or > pivot, you recursively choose a pivot element for those new partitions and repeat the process until the list is sorted. On average, quicksort has a nlog(n) runtime complexity, but its worst case makes it O(n^2). Quick sort is computed in-place, which results in a O(n) space complexity.

Description of Data Sets

Shuffled (1000) – elements [1,1000] shuffled
Ascending (800) – elements [1,800] in ascending (sorted) order
Descending (800) – elements [1,800] in descending order
N-8 Equal Elements Shuffled (800) – 9 unique elements in the data, where 8 of them have a count of 1
Article Share Count – Number of shares among 1000 articles published by Mashable (https://archive.ics.uci.edu/ml/datasets/Online+News+Popularity)
Breast Cancer Treatment Age – Age of patients who received breast cancer treatment (https://archive.ics.uci.edu/ml/datasets/Mammographic+Mass)

Performance of Sorting Algorithms

Based on the performance metrics gathered below, there are several implications to consider when choosing a sorting algorithm. Insertion sort, selection sort, and bubble sort all fall into the class of naïve O(n^2) sorting algorithms. Despite their high average complexity, they are still useful in certain situations. Because they operate in place and do not require recursion, they are especially useful for sorting small lists for several reasons: 1) They do not need to allocate additional address spaces/stacks because they do not recurse. 2) If they are contiguous in memory, fetching it is fast. 3) They can potentially be fast if the list is nearly sorted.

In the data where there are n-8 equal elements shuffled, insertion sort performed the best because it only had to consider those 8 elements and iteratively move them until their ordering condition was satisfied.

Both bubble sort and insertion sort performed well on the ascending (already ordered) data, because they iteratively check every element to see if any need to be swapped. If the list is sorted, the algorithms only have to perform n comparisons.

Selection sort is the worst of the three naive algorithms, because no matter what ordering of the data is, it must perform O(n^2) comparisons before it completes. Hence why its performance line is relatively in the same position regardless of the data.

Merge sort is reliable in the sense that its runtime and space is deterministic for every run. Its low runtime complexity compared to other sorting algorithms makes it excel in performance no matter what data it sorts on. In nearly every case, merge sort had the best or was very close to the best algorithm in terms of runtime. Its only drawback is its memory complexity and recursive nature when allocating new address spaces/stacks.

Quick sort can easily vary in performance depending on the data and pivot policy. In my implementation, I chose the highest element in the partition as the pivot, which resulted in it being the slowest algorithm for the ascending data since every element landed in the left partition. In the uniform random scenarios, which is most often the case for real world data, it can excel in runtime without the burden of merge sort's space complexity.

Issues with Metrics

For my measure of sortedness per algorithm, I used number of swaps or sets (merge sort). This metric however did not provide any useful information, because in most cases, performance is affected based on number of comparisons – selection sort can have 0 swaps but have O(n^2) comparisons. It has no correlation with runtime or memory used, so I placed those graphs in a scatterplot since no line can be drawn to represent the metrics.

Memory usage was constant per algorithm based on the input data. I am not sure why this is something worth comparing/plotting for each data set when it could easily be represented better in a table.

The only useful performance metrics are Runtime against Data Size.

Acknowledgements

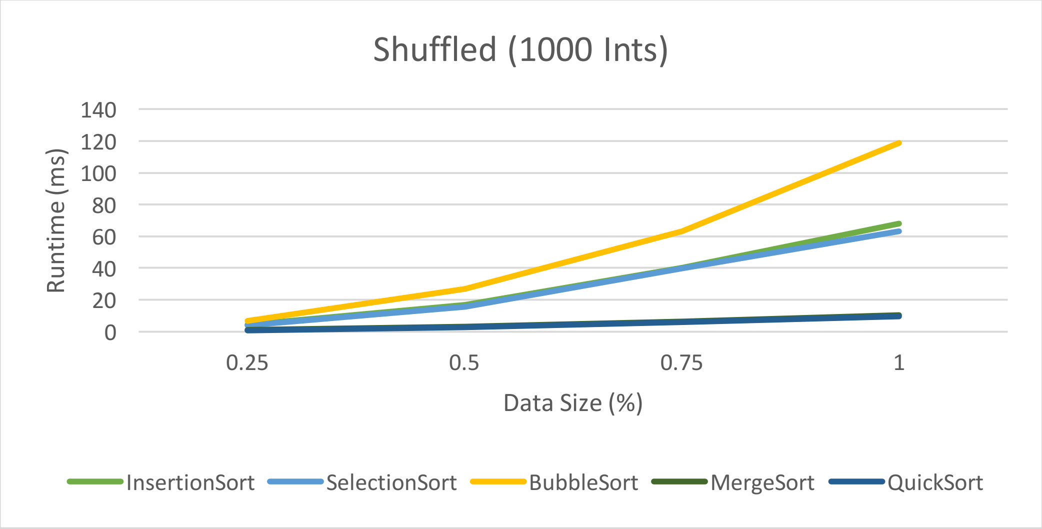
This code has been written in C++ using Timo Bingmann's "Sound-of-Sorting" (http://panthema.net/2013/sound-of-sorting/, https://github.com/bingmann/sound-of-sorting)

I wrote all of the sorting algorithms and scripts myself in separate files. I only use his class SortArray, which is essentially a wrapper around std::vector (equivalent to ArrayList in Java), and modified it to read CSVs and count the number of swaps and sets. The reason for using this code is for the next assignment when we have to visualize our code - I can simply use the same function headers and get the visuals found in the very famous video "15 Sorting Algorithms in 6 Minutes" (https://www.youtube.com/watch?v=kPRA0W1kECg).

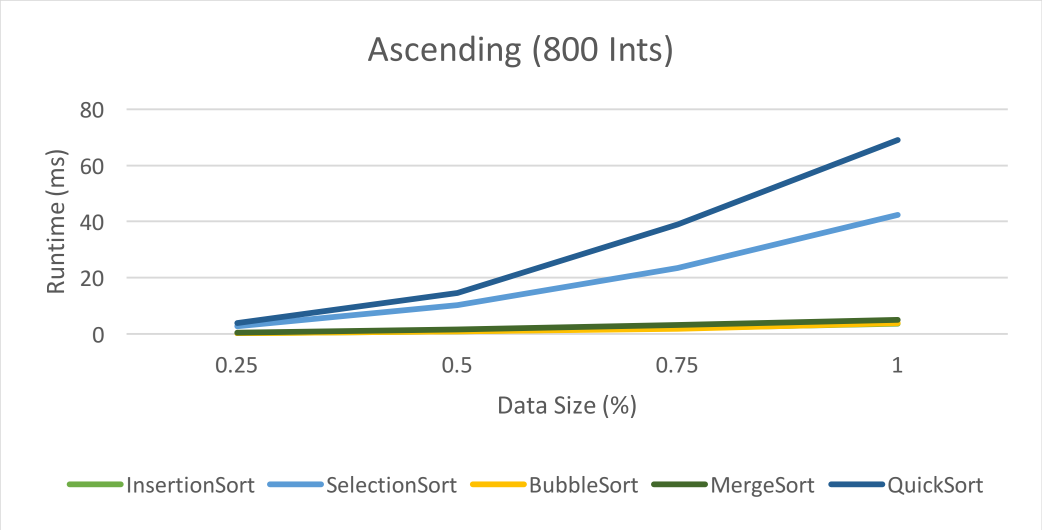
If this is not allowed, I will have no problem implementing my own visuals. As for this assignment, I see SortArray in the same vein as importing a Java ArrayList, and do not consider it to affect the core implementation of my sorting algorithms. Again, if this is an issue, please let me know and I will gladly fix the issue if permitted.

Metrics

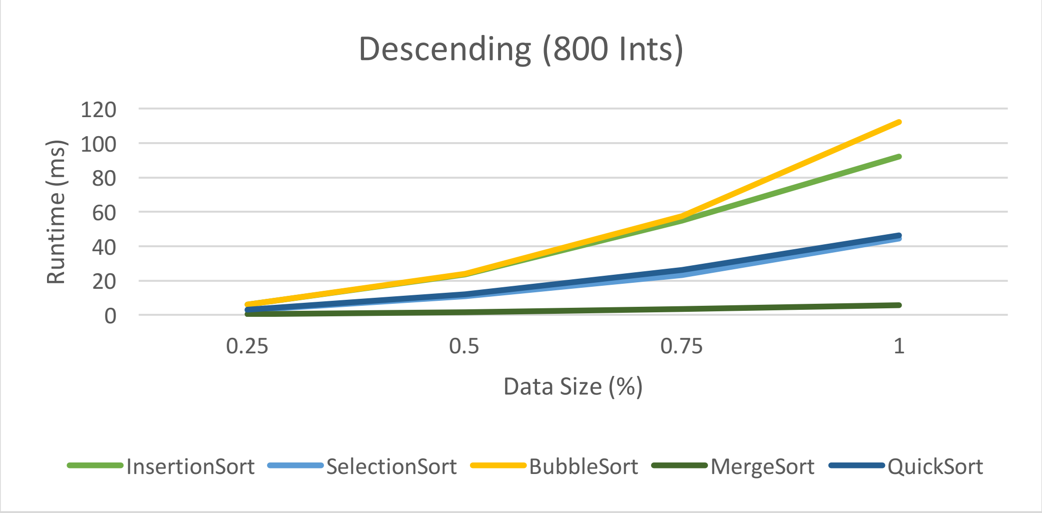
Runtime against Data Size



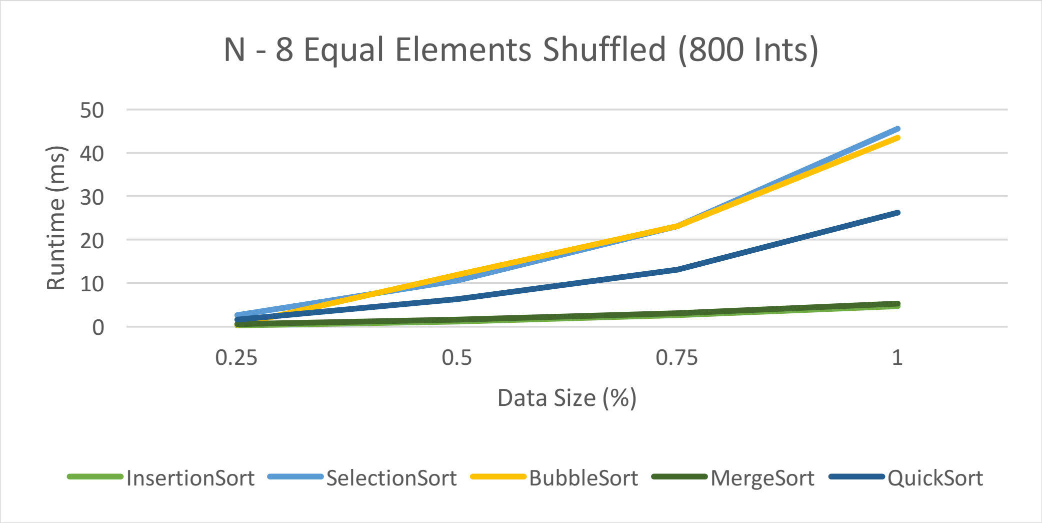
Shuffled (1000 Ints)

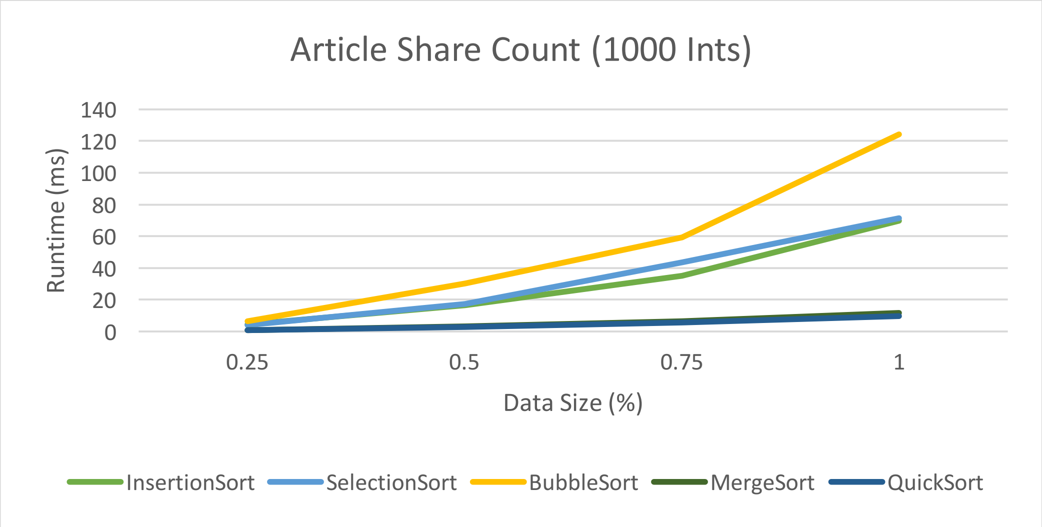


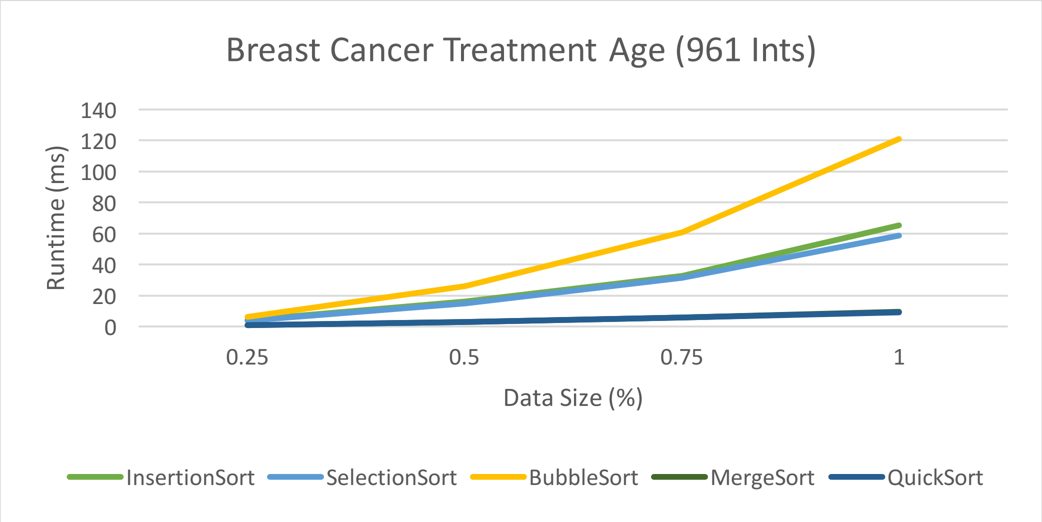
Ascending (800 Ints)



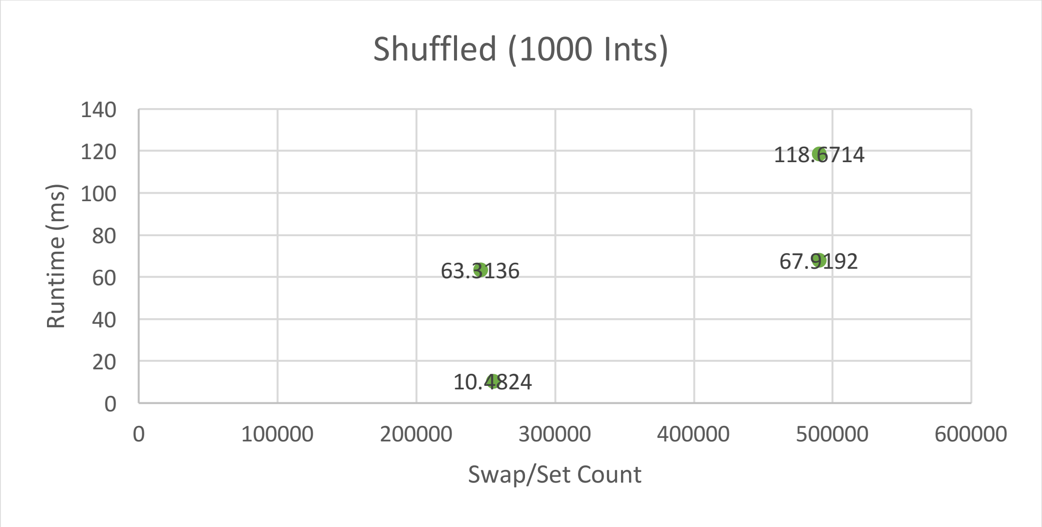
Descending (800 Ints)

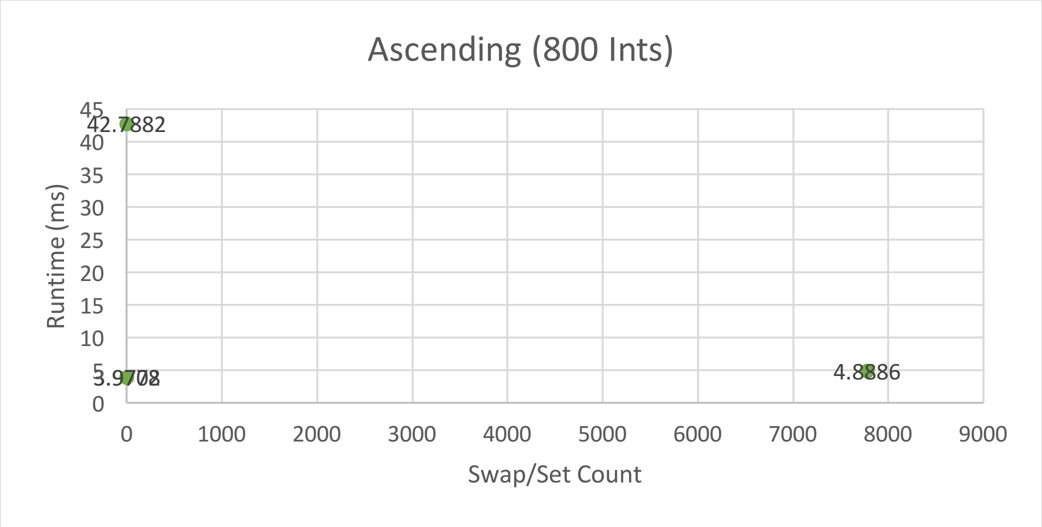

N - 8 Equal Elements Shuffled (800 Ints)
Runtime (ms)
50
40
30
20
10
0
0.25
0.5
0.75
1
Data Size (%)
InsertionSort
SelectionSort
BubbleSort
MergeSort
QuickSort

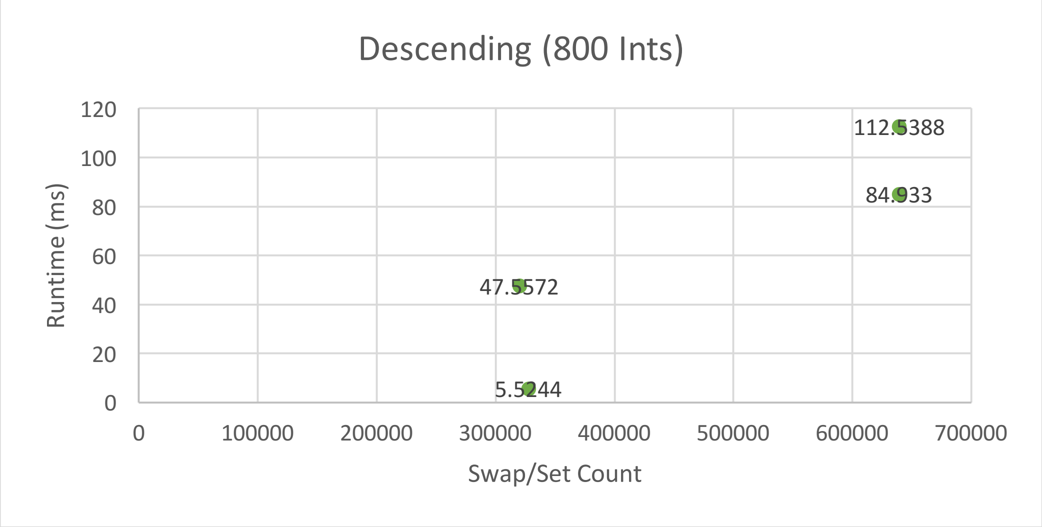

Article Share Count (1000 Ints)
Runtime (ms)
140
120
100
80
60
40
20
0
0.25
0.5
0.75
1
Data Size (%)
InsertionSort
SelectionSort
BubbleSort
MergeSort
QuickSort

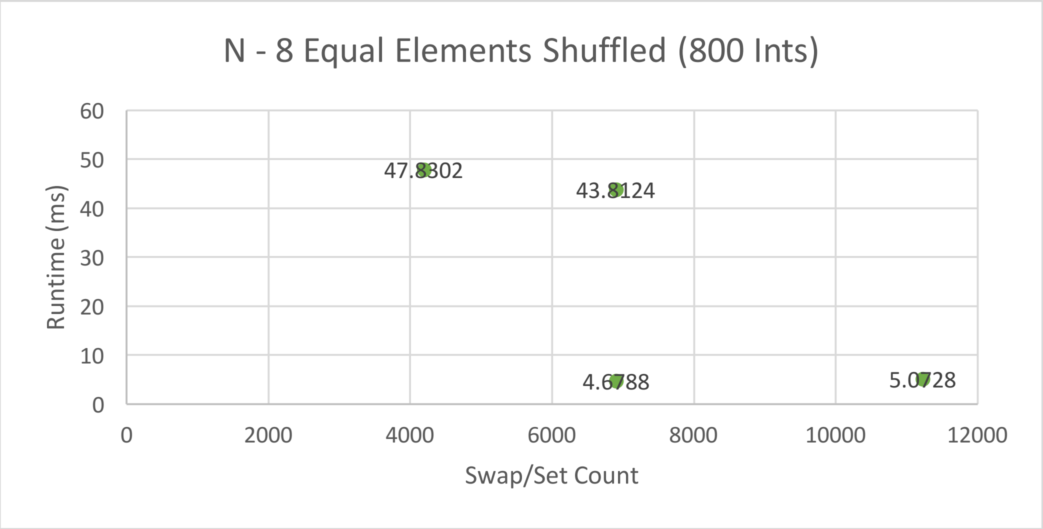

Breast Cancer Treatment Age (961 Ints)
Runtime (ms)
140
120
100
80
60
40
20
0
0.25
0.5
0.75
1
Data Size (%)
InsertionSort
SelectionSort
BubbleSort
MergeSort
QuickSort

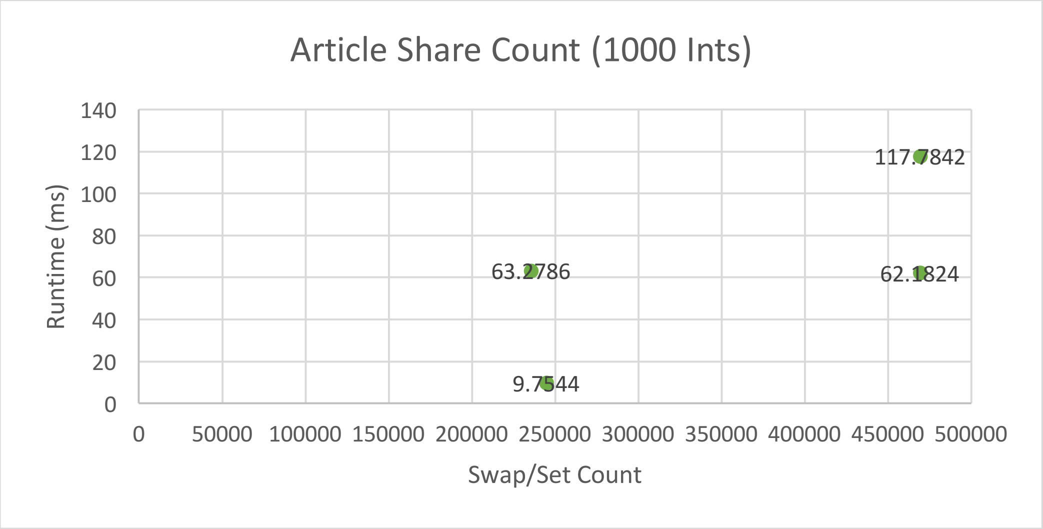
Runtime vs Swap/Set Count

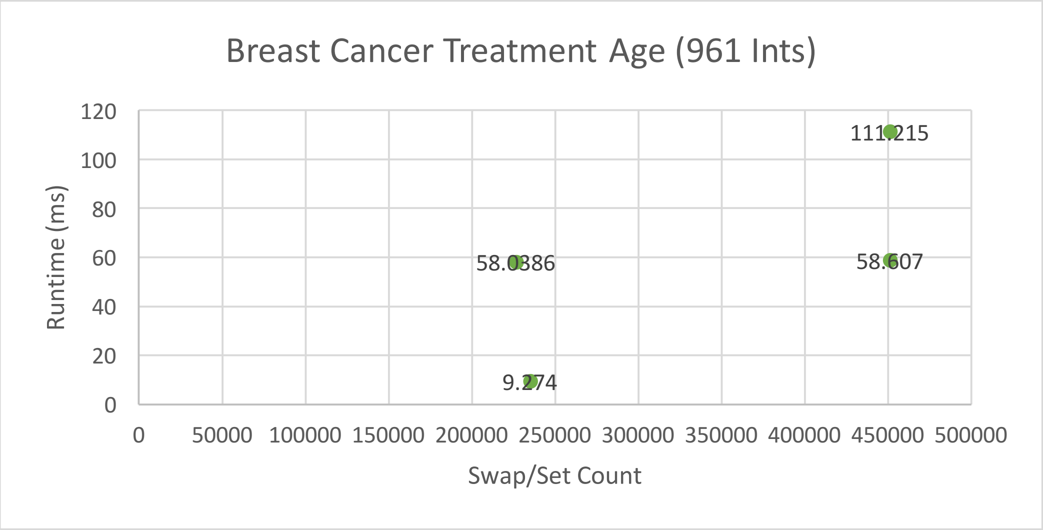

Shuffled (1000 Ints)


Ascending (800 Ints)

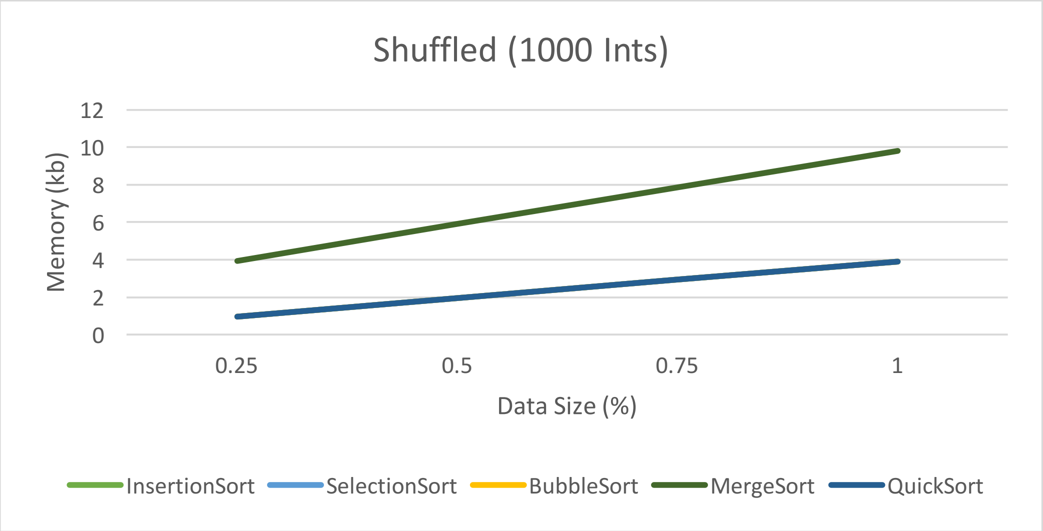

Descending (800 Ints)

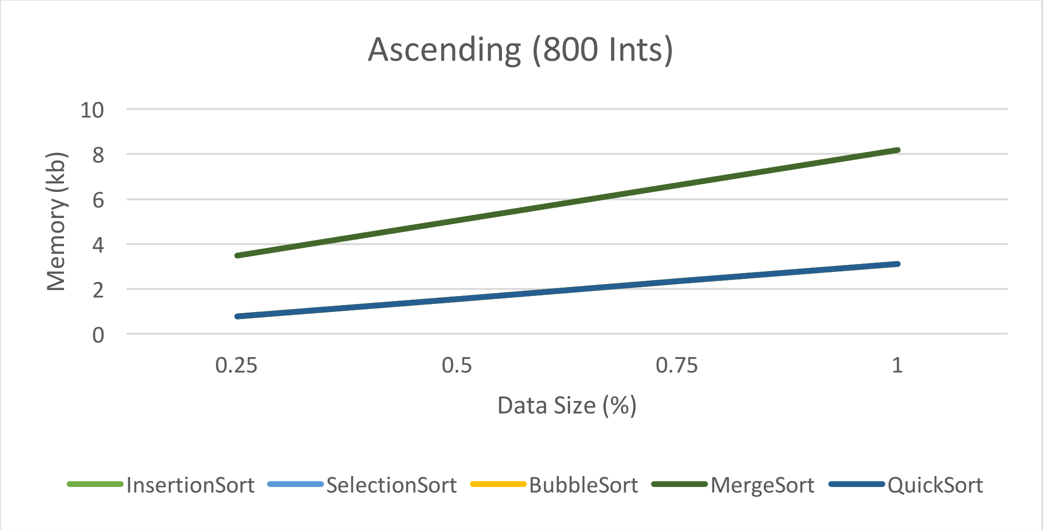

N - 8 Equal Elements Shuffled (800 Ints)

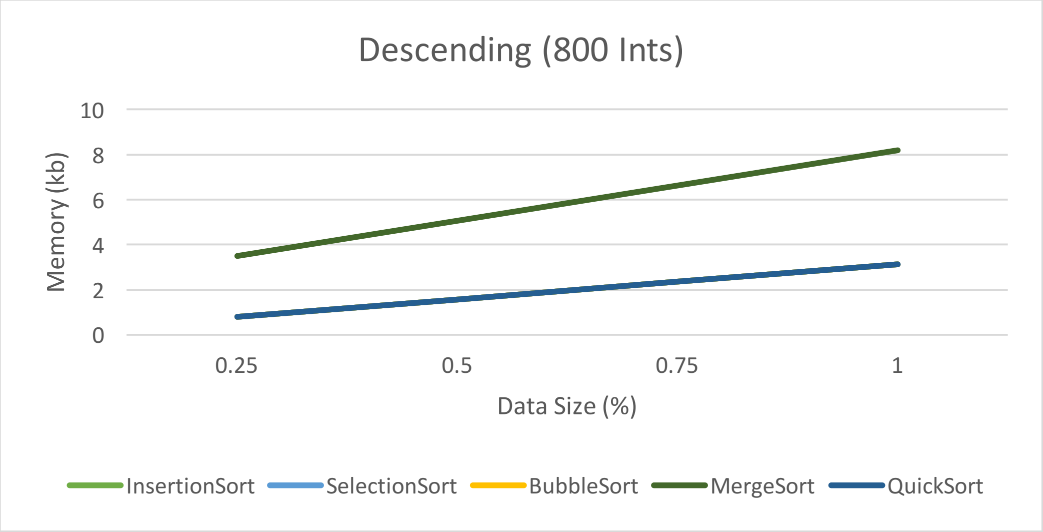

Article Share Count (1000 Ints)

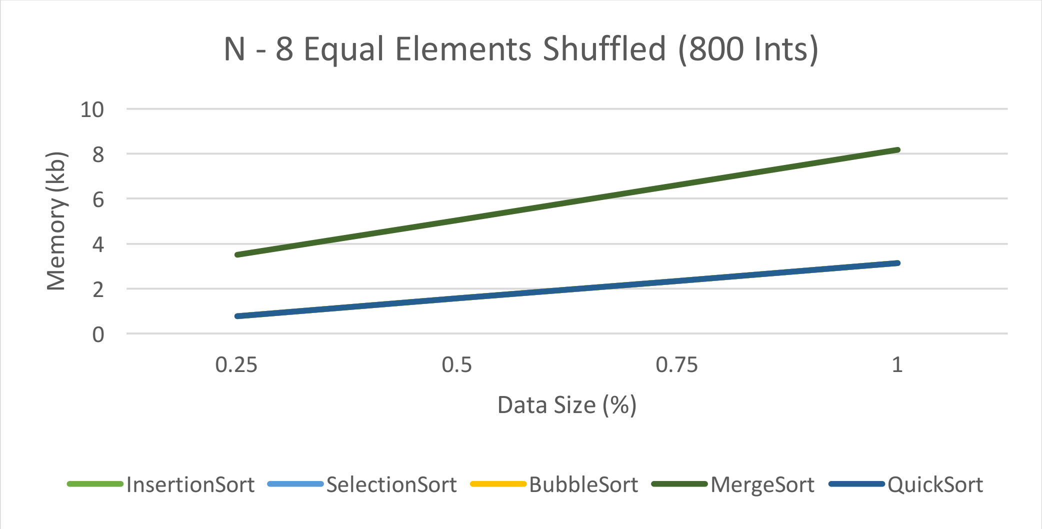

Breast Cancer Treatment Age (961 Ints)

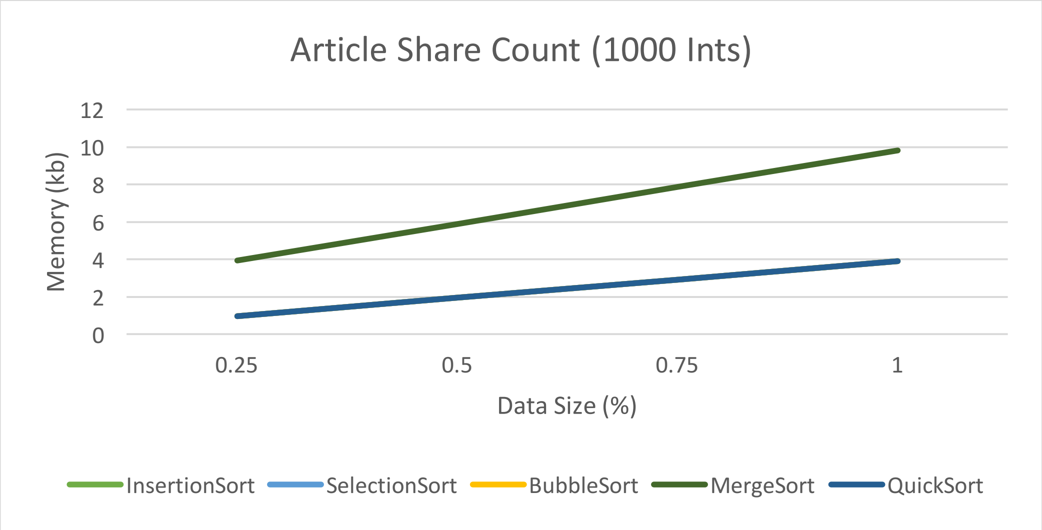
Memory Usage vs Data Size

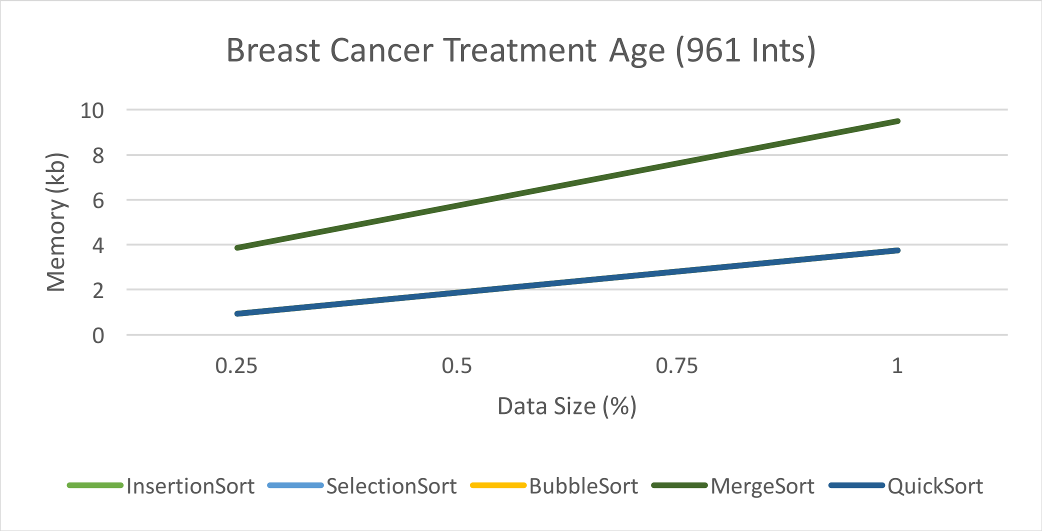




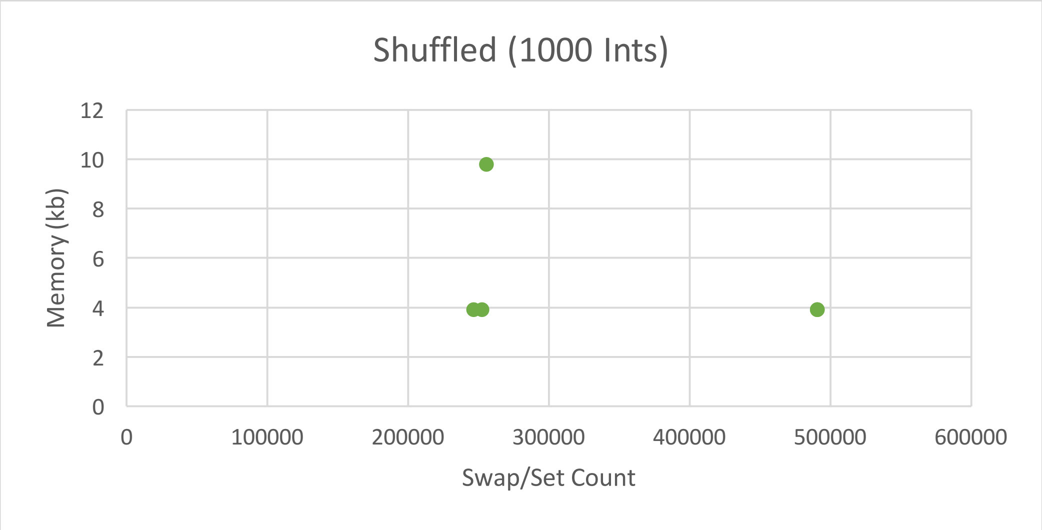


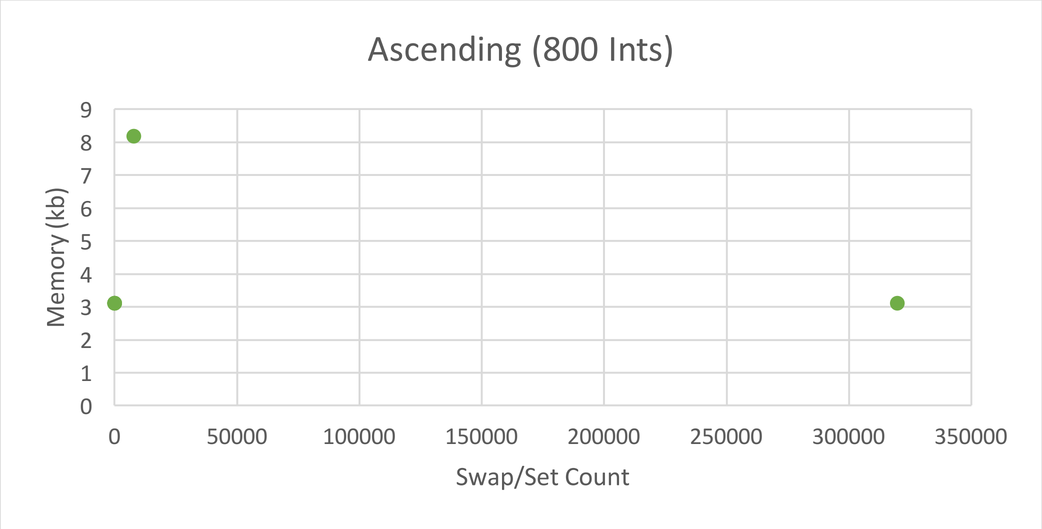

N - 8 Equal Elements Shuffled (800 Ints)
Memory (kb)
10
8
6
4
2
0
0.25
0.5
0.75
1
Data Size (%)
InsertionSort
SelectionSort
BubbleSort
MergeSort
QuickSort

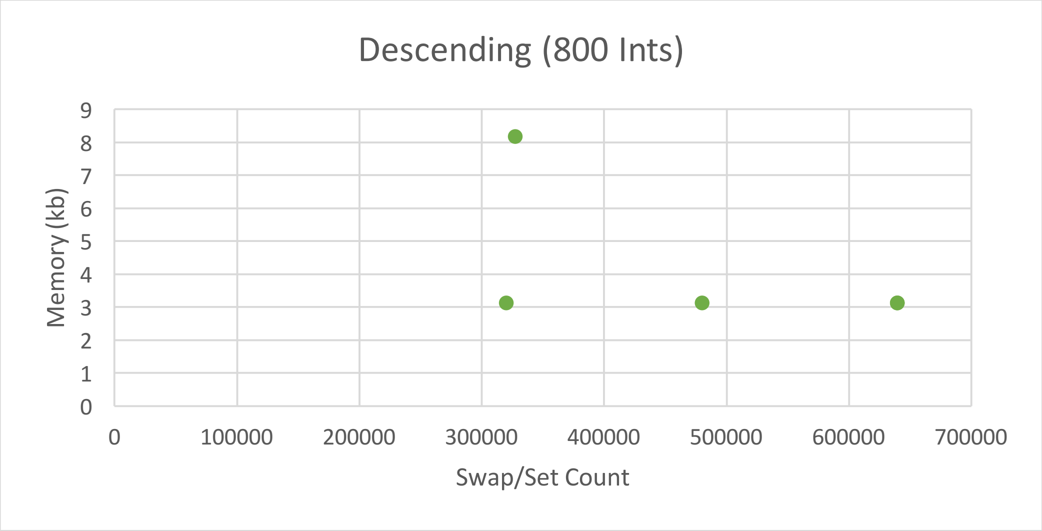

Article Share Count (1000 Ints)
Memory (kb)
12
10
8
6
4
2
0
0.25
0.5
0.75
1
Data Size (%)
InsertionSort
SelectionSort
BubbleSort
MergeSort
QuickSort

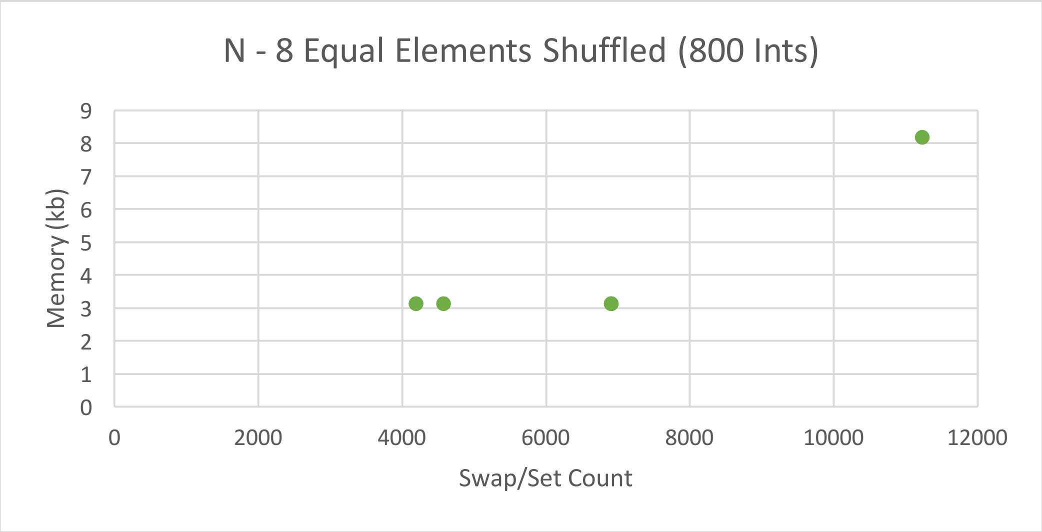

Breast Cancer Treatment Age (961 Ints)
Memory (kb)
10
8
6
4
2
0
0.25
0.5
0.75
1
Data Size (%)
InsertionSort
SelectionSort
BubbleSort
MergeSort
QuickSort

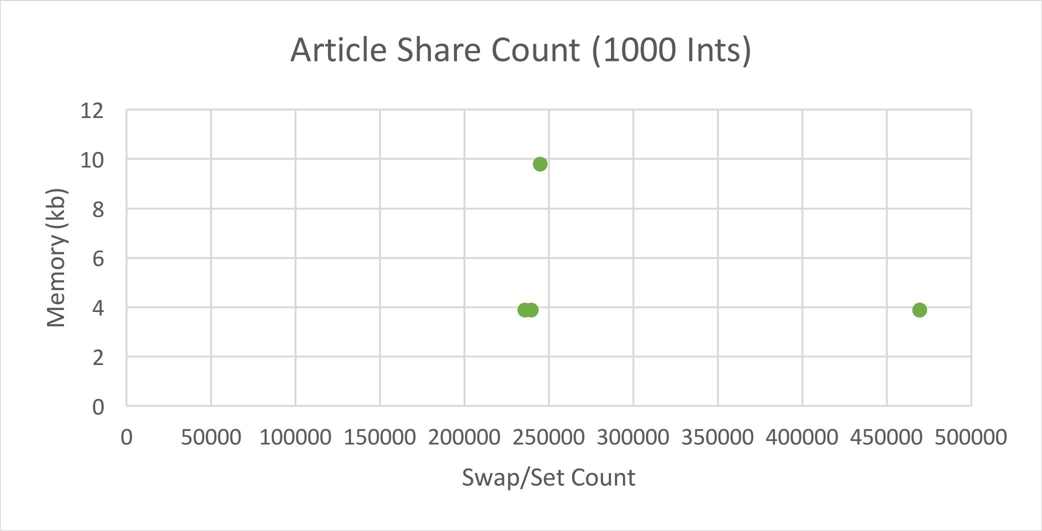
Memory Usage vs Swap/Set Count

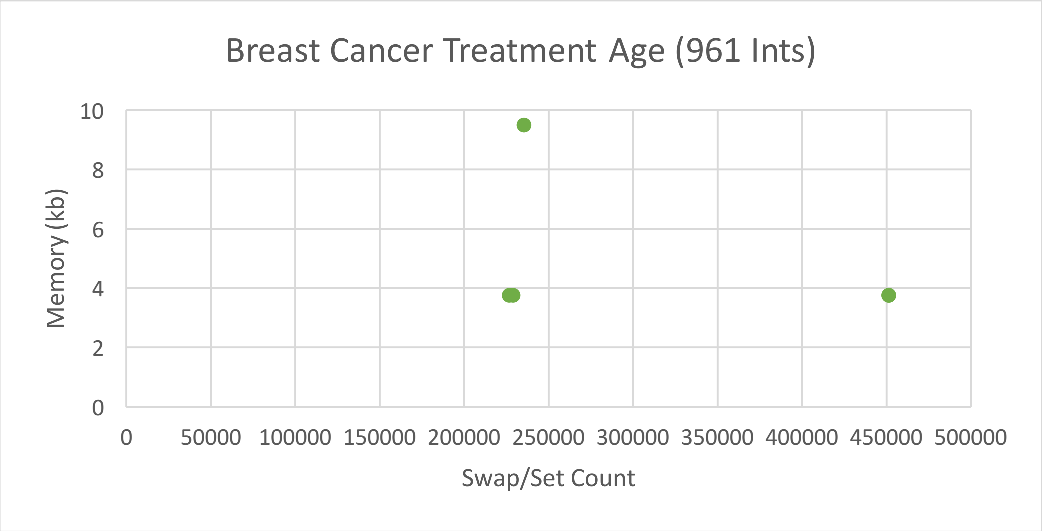







N - 8 Equal Elements Shuffled (800 Ints)


Article Share Count (1000 Ints)


Breast Cancer Treatment Age (961 Ints)