Program 4 – Sort Algorithms
DSA / S24
Group 1: Logan Phillips, Owen Hartson, Cole Vaughn, Chris Schneider

General Summary:

This program highlights the relative speed and limitations of several popular sorting algorithms. The takeaway from this assignment shows that for large vectors of randomized data, several sorts are so effective (std sort, merge sort) that it is more difficult to find their limitations than to use or implement them. In contrast, some sorts (bubble, quick sort) reveal that they are too slow for large data sets to be used efficiently. Though most of the algorithms tested reached speed bottlenecks faster than space, bucket sort proved to be very fast but very spatially expensive.

In conclusion, this program offers several benchmarks of performance to guide programmers to select a sorting algorithm that will be suitable for the size and shape of their data. However, the fastest and easiest sort to use is the sort that the standard library provides us, revealing the humbling scale of the shoulders of giants we sit on.

Boundary Testing Overview:

- ***At what size vector does the run time break 1 minute?***

Sort Type	Approximate Size *	Margin of Error **	Seconds
Std sort	760,000,000	10,000,000	32 ***
Bubble sort	231,000	1,000	60
Merge sort	515,000,000	1,000,000	69
Insertion sort	870,000	1,000	62
Selection sort	493,000	1,000	62
Quick sort	166,000	1,000	64
Bucket sort	470,000,000	10,000,000	16 ***

** Note that the size in practice varies based on the random vector. For example, Quick Sort broke a minute between 166,000 – 190,000 items.*

*** This is the increment size when looping around the estimated one minute mark. An exact size would take ~1 minute * i increments = way too many minutes when the estimate is in thousands, let alone millions. For this reason, these sizes are single passes as close to 60 seconds as possible and not means.*

**** My machine ran out of memory before I could run this sort on a larger size to get closer to 1 minute.*

- ***Can we create a specific vector that can dramatically alter the run time for some of the sorting algorithms?***
    - Yes, we can create a specific vector that dramatically alters the run time.
        - If our vector is already sorted or nearly sorted, sorts such as Insertion sort will be significantly faster than other sorts.
        - If our vector is reverse sorted, sorts such as Merge or Heap sort will be significantly faster than other sorts.
        - If our vector is randomized, sorts like Heap or Quick sort will be faster than other sorts.

- ***Can a larger number of duplicates alter the run time?***
    - Yes, a larger number of duplicates can alter the run time.
        - If our vector is all duplicates, it is already sorted and the logic of the previous question applies.
        - If our vector contains many duplicate values, sorts such as Quick or Merge sort will be faster than other sorts.

- ***Why is this information useful?***
    - Knowing the general "shape" of our input vector, such as the number of duplicates or the extent that it is already sorted, affects which sorting algorithm will be the most effective, as described above and visualized here.
    - This is why there is no "best" sorting algorithm and why the best option is dependent on the input data.