To run this tool, execute the following command: python3 sortTest.py 1000 500 2000 4

Usage
This tool tests the average running time of various sorting algorithms in Python. By default, it tests the algorithms QuickSort, MergeSort, and SelectionSort, in that order. Each algorithm is contained in its own file, called “mergeSort.py,” “quickSort.py,” and “selectionSort.py,” respectively.

The program also makes use of “makeArray.py,” which generates Python lists of various sizes n for testing purposes. Each list contains n elements, each of which is an integer. Three methods of generating these lists are implemented. “Random” selects each list item as a random integer between 0 and n-1, with each random choice made independently of the others. “Ordered” generates an ordered list of length n which has the property that Array[i] = i. “Evil” generates a list of length n in reverse order, which means that Array[i] = [n-i-1].

The QuickSort algorithm is implemented to partition the array around the last element. It first sorts the array into elements smaller than the last element, and elements larger than the last element. Then, it recurses on each side until the subsets it’s sorting are a single element. When an array is generated in evil mode, it is the worst case scenario for QuickSort. The array is in reverse order, and QuickSort partitions around the last element, which means that every time QuickSort chooses an element to partition around, it has to move every element in the section of the array that it is looking at in front of the partition element. This also means that when it goes to recurse on a subarray of size n, it is sorted into an empty set of smaller elements, the partition element, and a set of larger elements of size n-1. This results in a running time of O(n^2).

The Merge Sort algorithm pursues a “divide and conquer” strategy. A top-level array is split in half, and each half is recursively sorted by Merge_Sort(). The sorted halves are then merged back together by the Merge() procedure, which preserves the ordering of the overall array through careful comparison of the elements in the two halves. See in-code documentation for details.

Selection Sort pursues a more linear strategy. Beginning with consideration of the last element in the array, it searches all elements to the left of that element for their maximum, and compares it with the rightmost element. If the found prefix maximum is higher than the rightmost element, it swaps the two elements. It then recurses on a slice of the array excluding the rightward elements that have already been sorted until all elements have been sorted. In this way the elements with higher values “float” to the top until the array has been fully sorted.

Arguments:
- Minimum input size (1000 in the example), the size of the array to start
- Increment size (500 in the example), the amount the array’s size increases by each loop
- Maximum input size (2000 in the example), the size of the array the program will stop at, inclusive

- Number of trials (4 in the example), the number of times the program will create and sort an array of a certain size to get an average time from

Optional Arguments:

- -w/–whichsorts + QuickSort, MergeSort and/or SelectionSort, specifies which sorts to use. If not included, it will use all three
- -g/–generator + random, ordered, and/or evil, specifies how the arrays should be made. If not included, it will use random
- -h/–help, prints help documentation