# **Objectives**

The goal of this assignment was to implement the selection sort algorithm and two variations of the quick sort algorithm to test their performance utilizing a high-level programming language.

# **Program Design**

In order to successfully implement the required sorting algorithms, as well as test and compare their performance, several functions were implemented in addition to the actual sorting algorithms themselves. Several of these functions were modified from the last assignment to improve the testing process. These are:

1. main() – This driver function calls readFile() on each different text file.
2. readFile() – This function contains the code to read the text files, implement the sorting algorithms, calculate their execution time, and print the results.
3. selectionSort() – Function containing the selection sort algorithm.
4. printArray() – This function adds each element of the sorted array to a string and prints it for testing purposes.
5. quickSort() – Function containing the quick sort algorithm.
6. partition() – Called in quickSort(), this function this function is responsible for selecting a pivot element and partitioning the input array into two subarrays then returns the new pivot.
7. medianOfThreeQuickSort() – Function containing the median of three quick sort algorithm.
8. medianOfThree() – This function uses a series of if statements to determine the median of the lowest, middle, and highest index in the array.
9. partitionMedian() – This function is responsible for partitioning the array at the pivot value determined by medianOfThree() into two subarrays.

**main()**

This driver function contains a function call of the readFile() function on each text file containing a different unsorted array. There are also empty lines printed in between each call for the sake of easier reading during testing. This was done differently than the last assignment because during testing I found it tedious to have to change the file parameter for each file, so this is much easier and digestible as results.

**readFile()**

The code contained in this function is responsible for reading the file, adding each element in the represented array from the text file to an actual array, calling each sorting algorithm on the resulting array, then calculating the execution time for each and printing the results. The text file is read using the Scanner class. The trim function is then called on the resulting string to remove the space at the beginning. The string is then split at each space using a regex and added to an array of strings. This array is then iterated through using a for loop and each element is converted to an integer using parseInt(). Within this same for loop each integer element is then added to an array and a copy of this array, so that it can be reset after each sorting algorithm is called on it. Each sorting algorithm is then called on the array. But before the function call, the system time is recorded using system.nanoTime(). The time is then once again recorded after. The difference between the two is calculated to give the execution time represented in nanoseconds. This is then printed in addition to a float representation of the time converted to milliseconds and seconds. The file name and the array size are also printed for understandable results. The printArray() function is also called when not commented out to verify the sorting algorithm sorted the array as expected.

**printArray()**

This is the final helper function not directly involved in the sorting algorithms. It simply iterates through the array using a for loop after it has been sorted and adds each element to a string. This string is then printed. This function is used only for testing purposes and is commented out in the final submission of this assignment.

**selectionSort()**

The selection sort algorithm first finds the minimum element in the unsorted portion of the array by comparing each element to the current minimum. The program accomplishes this using a nested for loop, in which the outer for loop initializes the minimum as the current index i. The inner for loop then iterates through the array comparing each element in the array after i to i, assigning the new minimum is an element is less than i. Once a new minimum is found, it is swapped with the first element in the unsorted portion, effectively moving it to the correct position in the unsorted portion of the array. This is accomplished programmatically by assigning i to a temp value, then assigning the new min to i, and then assigning the temp value of i to the min. This is repeated for each element in the array until each element has been moved to the correct position, sorting the array.

**quicksort()**

The quickSort() function calls partition() to find a pivot index and then recursively calls itself on the resulting left and right subarrays which are defined by the pivot – 1 and pivot + 1 respectively. This continues as long the lowest index is less than the highest index, essentially meaning the function will recursively call itself until the subarrays have been reduced to a size of one.

**partition()**

This function is responsible for selecting a pivot element and partitioning the array into two subarrays: one containing elements less than or equal to the pivot, and the other containing elements greater than the pivot. After the partition, the function returns the new pivot index. This is accomplished programmatically by iterating through the array and comparing the value of the current index to that of the highest index in the array using an if statement to check if this is true. If it is, then the value of the current index is swapped with that of the pivot. After the for loop is executed, the pivot + 1 is swapped with the highest index in the array to ensure it is also in the correct position. This pivot value is then returned as i + 1.

**medianOfThreeQuickSort()**

The median of three quick sort algorithm functions essentially the same as the quick sort algorithm apart from several key differences. The pivot value is not determined by partition(), instead it is the median of the lowest, middle, and highest index in the array found in the medianOfThree() function. The array is then partitioned using partitionMedian() which functions similarly to partition(), partitioning the array into two subarrays.

**medianOfThree()**

This function utilizes a series of if statements to determine pivot value at which partition median splits the array into the two subarrays. The function first calculates the median index by adding the lowest index to the difference of the highest and lowest index divided by two. The function then uses a series of if statements to determine which of the three indices has the median value and returns that index as the pivot value. The if statement logic first determines if the lowest index’s value is higher than the highest index, causing the middle index to be the median if it is lower than the highest index, the highest to be the median if the lowest index value is greater than the middle, and the lowest to be the median if neither. If the lowest isn’t greater than the highest, then the function returns the lowest as the pivot. The lowest is also returned as the pivot if the middle value is greater than the highest, and if none else the middle index is returned as the pivot.

**partitionMedian()**

The partitionMedian() function partitions the array into two subarrays: one containing elements less than or equal to the pivot, and the other containing elements greater than the pivot, then returns the new pivot value. First the function swaps the temp value for the pivot and then swaps the pivot index for the highest index in the array. The function then iterates through the array from the lowest to highest index in the array and checks with an if statement if the value of the current index is less than or equal to the pivot. If this is true then the function replaces the current index value with temporary pivot value. After the for loop has completed, the value of the highest index is then swapped with the value of the pivot, essentially placing the pivot in the correct position in the array. The function then returns this pivot value.

A white background with black and white clouds

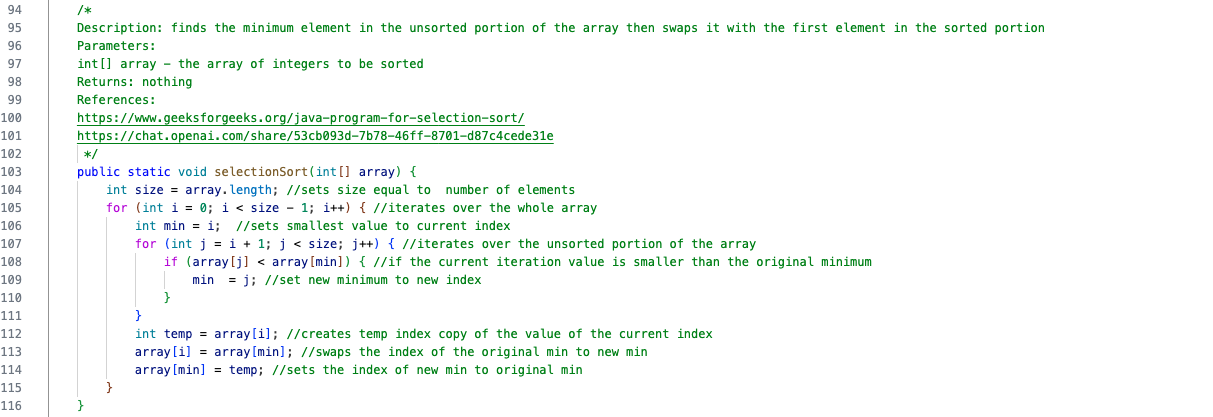
Description automatically generated

main()

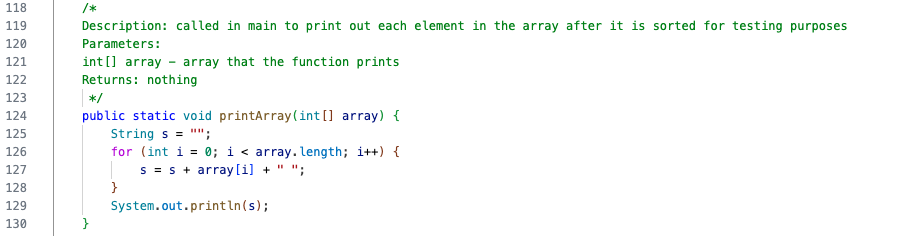
A screenshot of a computer program

Description automatically generated

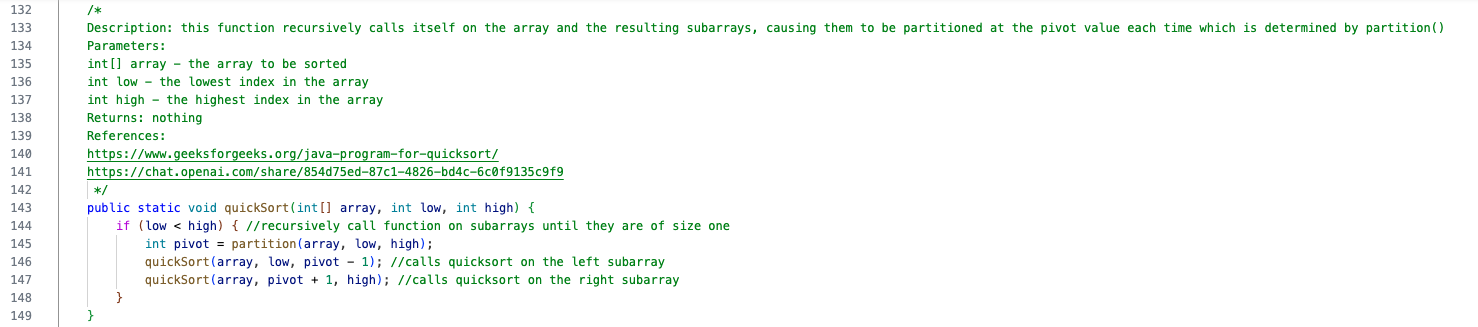
readFile()



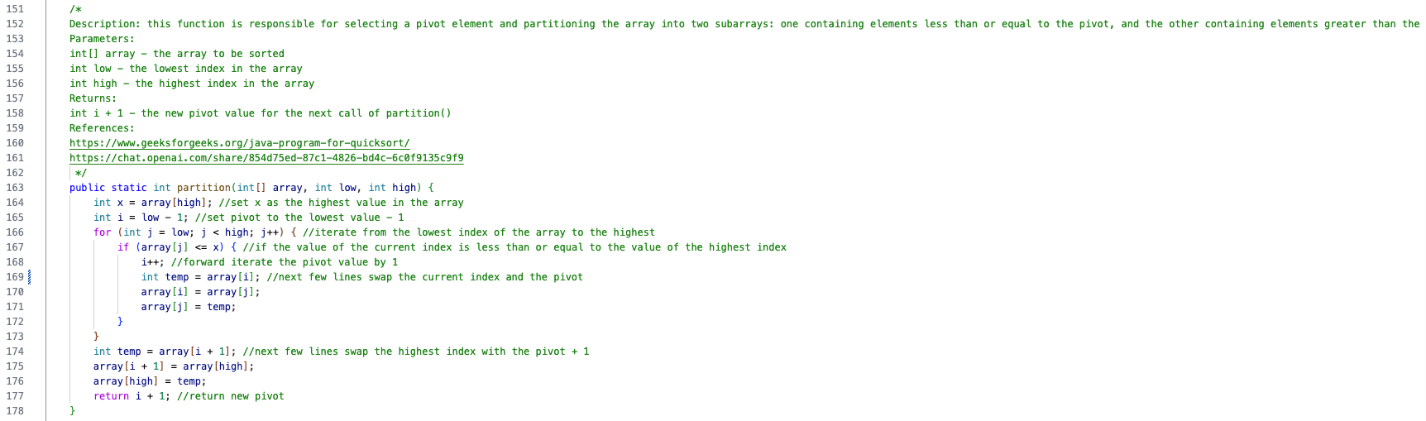
selectionSort()



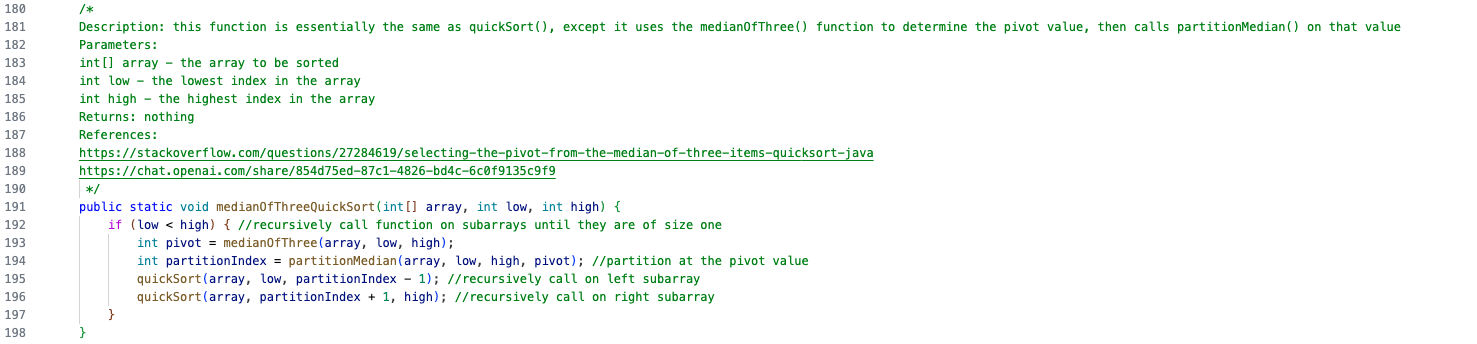
printArray()



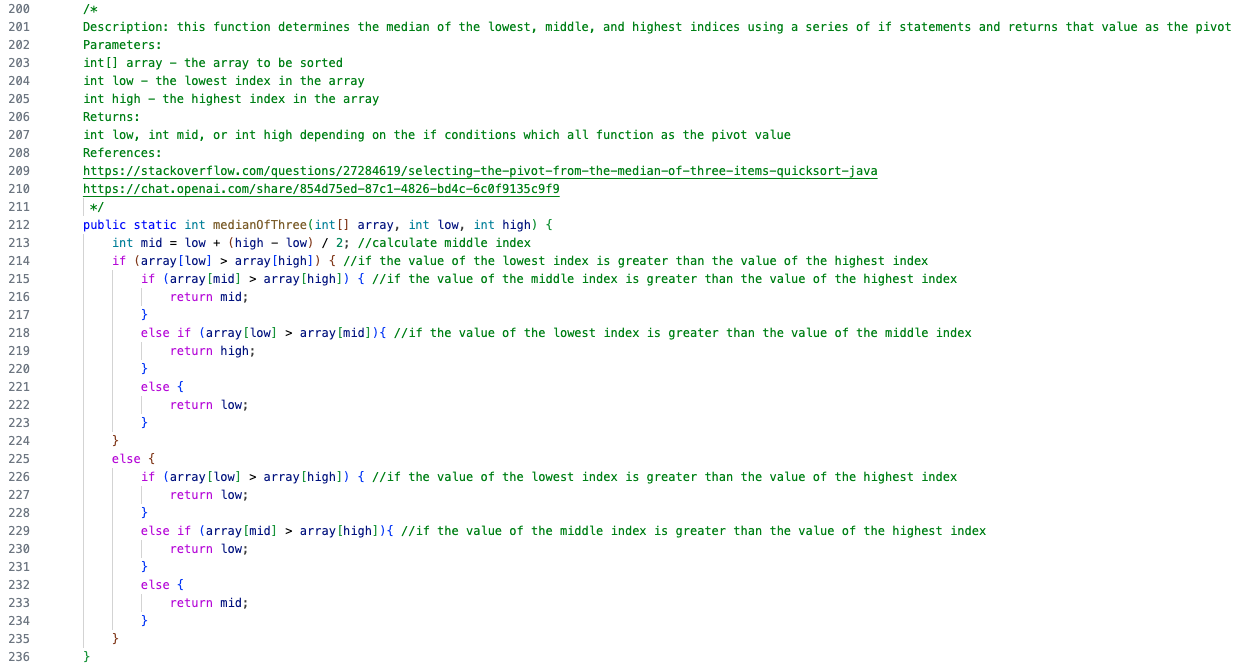
quickSort()



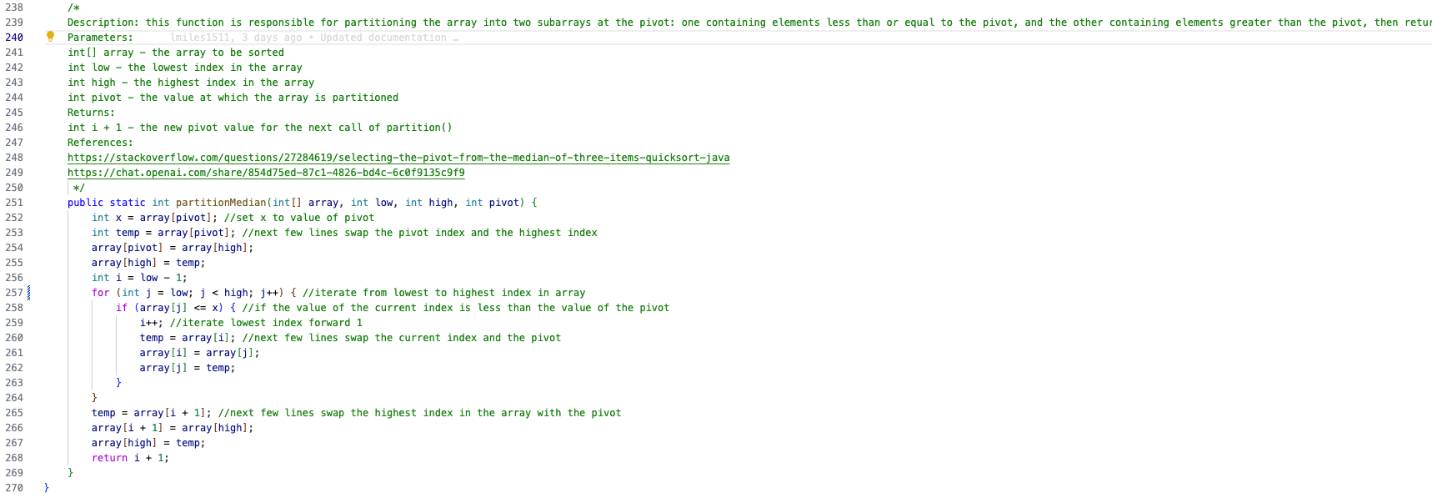
partition()



medianOfThreeQuickSort()



medianOfThree()



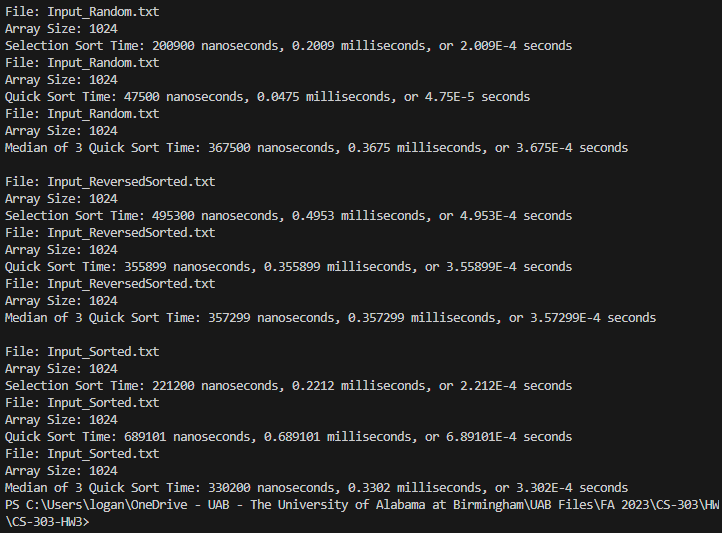
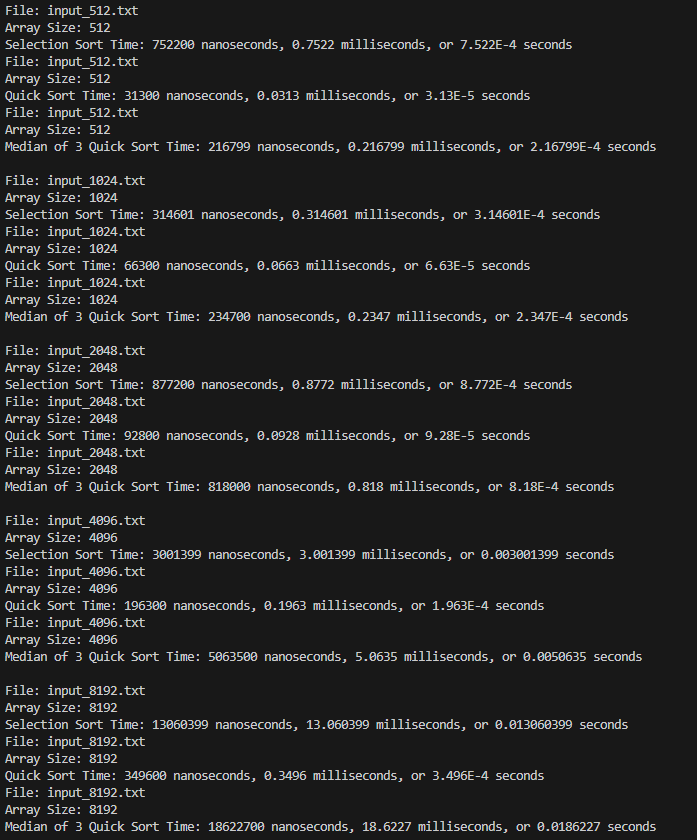
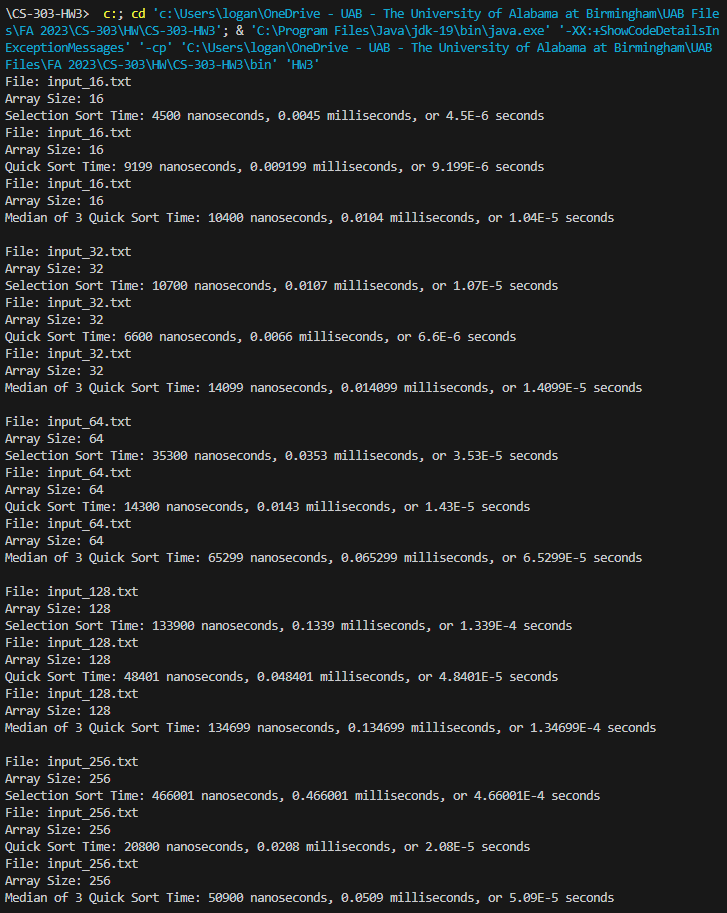
partitionMedian()

# **Testing**

Testing of the sorting algorithms and their supporting functions was done using several text files containing integers separated by spaces ranging from a size of 16 integers to 8192 integers. There are also three special case text files for testing as well. These are:

1. Input\_Random.txt – Contains random integers.
2. Input\_Random.txt – Contains integers in reversed sorted order.
3. Input\_Sorted.txt – Contains integers sorted in ascending order.

Each different text file is called in readFile(), and the print statements in main() provide the file name, execution time, and array size.



Based on the representation of the execution time versus the number of elements per sorting algorithm shown in the above graph, there is a distinct difference between each of the three algorithms. The execution time for selection sort suffered from a noticeable change in execution time after the number of elements increased from 1024 to 2048 and then increased approximately quadratically with each doubling of the number of elements. This quadratic increase is present throughout each increase in the number of elements; however, it is much more noticeable the greater the number of elements increases. For example, a difference of 245200 nanoseconds to 1257601 nanoseconds is much more significant than an increase from 5799 nanoseconds to 11700 nanoseconds. Overall, the time complexity of selection sort is O(n2). This is because the algorithm utilizes a nested for loop. The median of three quick sort algorithm also suffered from an increase in execution time after an increase in elements. The worst-case time complexity for quick sort is O(n2) and the best-case scenario is O(nlog(n)). This algorithm avoids the worst case time-complexity by choosing the median of the first, middle, and last elements, however this also sacrifices the possibility of achieving the best-case time-complexity. The quick sort algorithm, however, did not have a dramatic increase in execution time with an increase in the number of elements. Unlike the selection sort algorithm and median of three quick sort algorithms it only increased by approximately nlog(2n) with each array size increase. This means that quick sort has a time complexity of O(nlog(n)). Comparing each of the sorting algorithms, there doesn’t seem to be much of a difference between their execution when the number of elements is lower, but as the number of elements increases, this difference becomes more apparent. This is because a quadratic increase in time is much greater than a logarithmic increase in time.

The graph above compares three specific scenarios for each of the sorting algorithms. Each array has 1024 elements and represents one of the following:

1. The array consists of randomized integers ranging from 1 to 1024.
2. The array consists of integers sorted in descending order.
3. The array consists of integers sorted in ascending order.

The difference between the way each of these algorithms handles these scenarios is shown by the difference in execution time taken for each. The selection sort algorithm handled the random and reversed arrays with little difference in execution time, however the sorted array was operated on in approximately half the time. This is because the algorithm does not have to swap any of the elements in the array resulting in a best-case time complexity of O(n). The quick sort algorithm achieved its own best-case time complexity of O(nlog(n)) in the randomly sorted array, but also sunk to its worst-case scenario of O(n2) when the array is already sorted. The best-case time complexity is a result of even array partitioning when choosing the pivot in partition(), while the worst-case time complexity is a result of uneven array partitioning due to the fact that the array was already sorted causing the pivot to be chosen as the first element. The reason that the time complexity of the algorithm did not suffer when the array was reverse sorted is because the first element needed to be in the last position, causing more even partitioning. The median of three quick sort algorithm avoided this worst-case time complexity by choosing the pivot based of the median of the first, middle, and last elements, but at the cost of also not achieving the algorithms best-case time complexity.

# **References**

<https://www.geeksforgeeks.org/java-program-for-selection-sort/>

<https://chat.openai.com/share/53cb093d-7b78-46ff-8701-d87c4cede31e>

<https://www.geeksforgeeks.org/java-program-for-quicksort/>

<https://chat.openai.com/share/854d75ed-87c1-4826-bd4c-6c0f9135c9f9>

<https://stackoverflow.com/questions/27284619/selecting-the-pivot-from-the-median-of-three-items-quicksort-java>