CPSC 319 ASSIGNMENT 1

John Ezekiel Juliano <john.juliano>

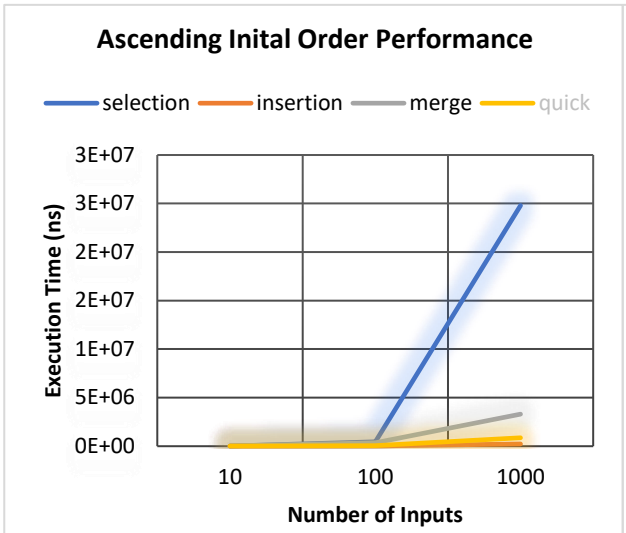
T05 <Rashid Mamunur>

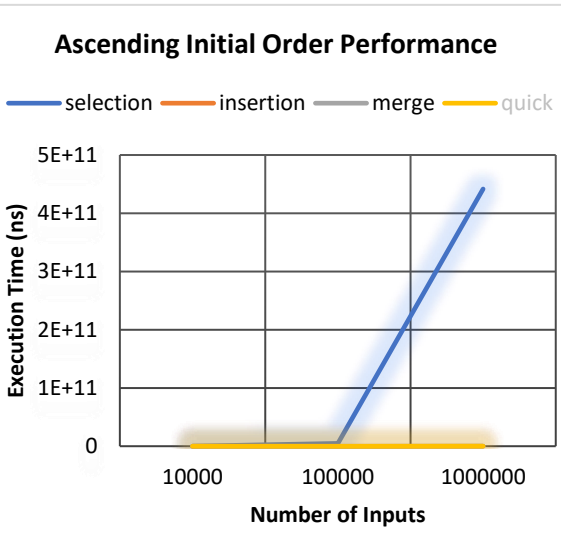
I. Experimental Methods

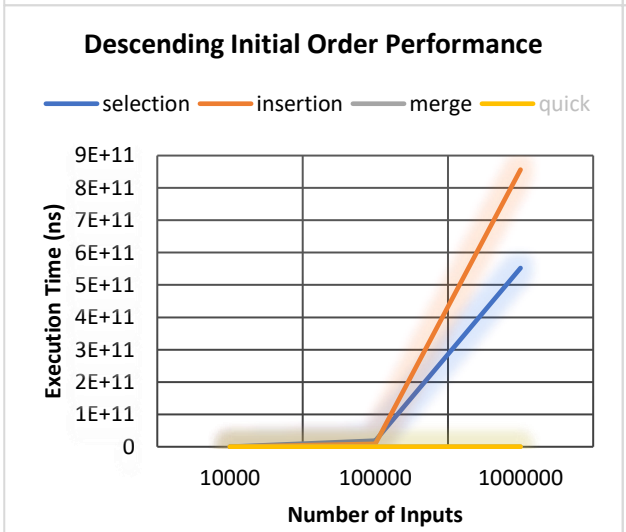
There are various ways to sort an array or list with an arbitrary number of inputs. Some are faster and more efficient than others. There are also ones that need extra memory and ones that incorporates recursive method calls. The objective of this experiment is to compare the performance of four different sorting algorithms: selection, insertion, merge, and quick sort. Three different cases will be examined for each sorting algorithm an initial random, ascending, and descending array will be used to determine the average, best and worst cases (respectively) for each algorithm. A varying number of inputs will also be used to show the performance as inputs grow.

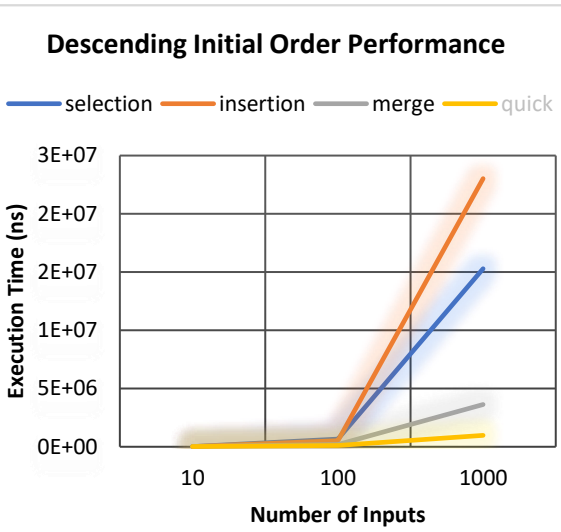
II. Data

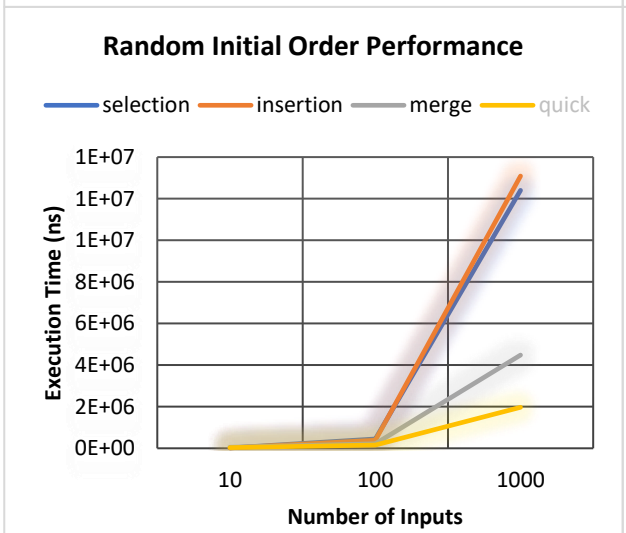
Time (ns)	Selection Sort			Time (ns)	Insertion Sort		
	Random	Ascending	Descending		Random	Ascending	Descending
10	1.97E+04	1.92E+04	1.92E+04	10	1.71E+04	1.28E+04	1.80E+04
100	4.43E+05	4.39E+05	6.64E+05	100	3.88E+05	3.46E+04	5.23E+05
1000	1.24E+07	2.48E+07	1.53E+07	1000	1.31E+07	2.31E+05	2.30E+07
10000	1.85E+08	6.54E+07	1.94E+08	10000	6.40E+07	3.05E+06	1.25E+08
100000	1.73E+10	4.64E+09	1.88E+10	100000	4.03E+09	1.95E+07	9.34E+09
1000000	1.87E+12	4.42E+11	5.52E+11	1000000	4.46E+11	2.97E+07	8.56E+11
Time (ns)	Merge Sort			Time (ns)	Quick Sort		
	Random	Ascending	Descending		Random	Ascending	Descending
10	5.05E+04	2.91E+04	3.08E+04	10	3.29E+04	2.05E+04	2.14E+04
100	2.21E+05	3.62E+05	1.94E+05	100	1.63E+05	7.31E+04	1.17E+05
1000	4.48E+06	3.30E+06	3.63E+06	1000	1.97E+06	8.70E+05	9.78E+05
10000	1.55E+07	1.17E+07	1.05E+07	10000	7.06E+06	6.21E+06	6.84E+06
100000	6.81E+07	7.24E+07	1.20E+08	100000	5.85E+07	6.75E+07	7.07E+07
1000000	3.46E+08	1.20E+08	2.18E+08	1000000	2.36E+08	1.08E+08	2.40E+08

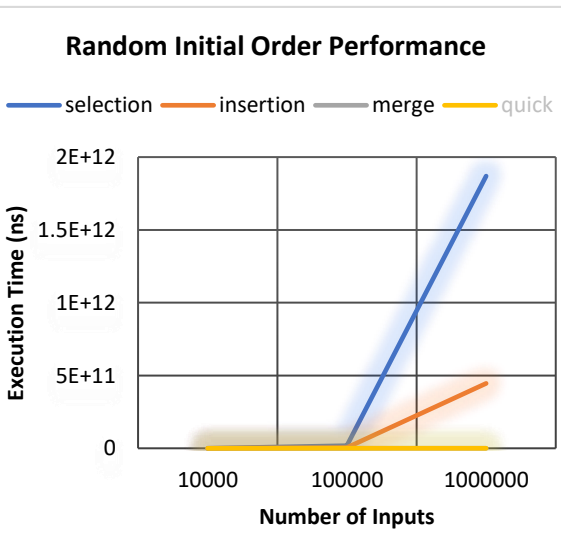

Ascending Inital Order Performance
selection
insertion
merge
quick
3E+07
3E+07
2E+07
2E+07
1E+07
5E+06
0E+00
Execution Time (ns)
10
100
1000
Number of Inputs


Ascending Initial Order Performance
selection
insertion
merge
quick
5E+11
4E+11
3E+11
2E+11
1E+11
0
Execution Time (ns)
10000
100000
1000000
Number of Inputs


Descending Initial Order Performance
selection
insertion
merge
quick
9E+11
8E+11
7E+11
6E+11
5E+11
4E+11
3E+11
2E+11
1E+11
0
Execution Time (ns)
10000
100000
1000000
Number of Inputs


Descending Initial Order Performance
selection
insertion
merge
quick
3E+07
2E+07
2E+07
1E+07
5E+06
0E+00
Execution Time (ns)
10
100
1000
Number of Inputs


Random Initial Order Performance
selection
insertion
merge
quick
1E+07
1E+07
1E+07
8E+06
6E+06
4E+06
2E+06
0E+00
Execution Time (ns)
10
100
1000
Number of Inputs


Random Initial Order Performance
selection
insertion
merge
quick
2E+12
1.5E+12
1E+12
5E+11
0
Execution Time (ns)
10000
100000
1000000
Number of Inputs

III. Data Analysis

Examining the graphs of ascending initial order, it can be seen that selection sort does poorly in both small and big number of inputs compared to the other algorithms. It is not as visible in the graph but from the table, insertion sort does the best under all situations. For descending initial order, it is insertion sort that suffers as the number of inputs increase while merge and quick sort are relatively the quickest ones. Lastly, for a random initial order, for a small number of input insertion sort is the slowest but for larger ones – selection. Quick sort is the one that performs well in a randomly arranged array.

IV. Complexity Analysis

The complexity analysis of each sorting algorithm is conducted assuming the worst-case scenarios. This could mean that if-statements are always true and other conditional statements are always satisfied.

```
converted format of selectionSort
NOTE: Accessing an array element is 1 op.
NOTE: 'n' is data.length
    TASK                          NUMBER OF OPERATIONS
int i = 0;                            1   op
while(i<data.length-1){               n   ops
    int j = i+1;                      n-1 ops
    int min = i;                      n-1 ops
    while(j<data.length){                       n+1 ops
        if(data[j] < data[min]){                3n  ops // 2 access + compare
            min = j;                            1n  ops
        }
        j++;                                    n   ops
    }
    int temp = data[min];             2(n-1)  ops         // access + assign
    data[min] = data[i];              3(n-1)  ops         // 2 access + assign
    data[i] = temp;                   2(n-1)  ops         // access + assign
    i++;                              n-1 ops
}                                    ------------------------------
                                      11n-9   *   6n+1 = 66n^2 - 43n - 9 >> O(n^2)
```

```
converted format of insertionSort
NOTE: Accessing an array element is 1 op.
NOTE: 'n' is data.length
    TASK                          NUMBER OF OPERATIONS
int i = 1;                            1   op
while(i<data.length){                 n   ops
    int temp = data[i];               n-1 ops
    int j = i;                        n-1 ops
    while(j>0 &&                                (1,2...(n-1))   ops
        temp < data[j-1]){                      2(1,2...(n-1))  ops
        data[j] = data[j-1];                    3(1,2...(n-2))  ops
        j--;                                    (1,2...(n-2))   ops
    }
    data[j] = temp;                   2(n-1)  ops // access + assign
    i++;                              n-1 ops
}                                    ------------------------------
NOTE: Assuming that (1,2...(n-1)) can be simplified to n-1 and (1,2...(n-2)) to n-2...
                                      6n-4    *   7n-7 = 42n^2 -70n + 28 >> O(n^2)
```

converted format of mergeSort and merge
NOTE: Accessing an array element is 1 op.
NOTE: 'n' is last which is also data.length
NOTE: first = 0

TASK	NUMBER OF OPERATIONS
`if(first<last){`	1 op
`int mid = (first+last)/2;`	1 op
`mergeSort(first, mid);`	1 op
`mergeSort(mid+1, last);`	1 op
`merge(first, mid, last);`	1 op
`*** merge ***`	
`int i = first;`	1 op
`while(i<=last){`	n+1 ops
`    tempArr[i] = data[i];`	3(n+1) ops
`    i++;`	n+1 ops
`}`	
`int leftSubIndex = first;`	1 op
`int rightSubIndex = mid+1;`	1 op
`int dataIndex = first;`	1 op
`while(leftSubIndex<=mid && rightSubIndex<=last){`	3[(n+1)/2] ops*
`    if(tempArr[leftSubIndex] <= tempArr[rightSubIndex]){`	3([(n+1)/2]-1) ops
`        data[dataIndex] = tempArr[leftSubIndex];`	3([(n+1)/2]-1) ops
`        leftSubIndex++;`	[(n+1)/2]-1 ops
`    }else{`	
`        data[dataIndex] = tempArr[rightSubIndex];`	3([(n+1)/2]-1) ops**
`        rightSubIndex++;`	[(n+1)/2]-1 ops**
`    }`	
`    dataIndex++;`	[(n+1)/2]-1 ops
`}`	
`while(leftSubIndex<=mid){`	C ops***
`    data[dataIndex] = tempArr[leftSubIndex];`	C-1 ops***
`    dataIndex++;`	C-1 ops***
`    leftSubIndex++;`	C-1 ops***
`}`	
	(19/2)(n+1) + Constant

Even though the complexity is just O(n), we need to know that the number of different partitions is proportional to the log(n). This is due to the recursive method call. Using the product rule this will give us O(n log n).

* assuming that even sub-arrays are always created then it should only take half the total time

** will not be considered since both cases A) when 'if' is true and B) when 'else' is true produce the same number of operations

*** these instructions will always be less than 'n' ops and will just be a constant 'C' because that is just to take care of the leftovers

converted format of quickSort
NOTE: Accessing an array element is 1 op.
NOTE: 'n' is last which is also data.length

TASK	NUMBER OF OPERATIONS
int first = lo;	1 op
int last = hi;	1 op
int temp;	
int pivot = data[(lo+hi)/2];	2 op
while(first<=last){	n+1 ops
while(data[first]<pivot){	(n+1)/2 ops*
first++;	[(n+1)/2]-1 ops*
}	
while(data[last]>pivot){	(n+1)/2 ops*
last--;	[(n+1)/2]-1 ops*
}	
if(first<=last){	1 op
temp = data[first];	2 ops
data[first] = data[last];	3 ops
data[last] = temp;	2 ops
first++;	1 op
last--;	1 op
}	
}	
if(lo < last)	1 op
quickSort(lo, last);	1 op
if(first < hi)	1 op
quickSort(first, hi);	1 op

	3n + 19 ops

Similar to merge sort, quick sort gain log n proportionality due to its recursive method calls.
Using the product rule this will result into a O(n log n)
* assuming that even sub-arrays are always created then it should only take half the total time

V. Interpretation

Based on the empirical data collected, for a fixed number of inputs all but one algorithm – insertion sort – performs at a relatively constant speed. This means that the sorting speed of selection, merge and quick sort are not dependent on the initial order of the array. In terms of which algorithm is best for each input size, we can infer to the data table and see that selection and insertion are the most fit for this task. Due to the simplicity of these algorithms, smaller sample sizes are very easily handled compared to the more complex ones such as merge and quick sort. Comparing the experimental with the theoretical performance of each algorithm based on the complexity analysis conducted, we get the following table:

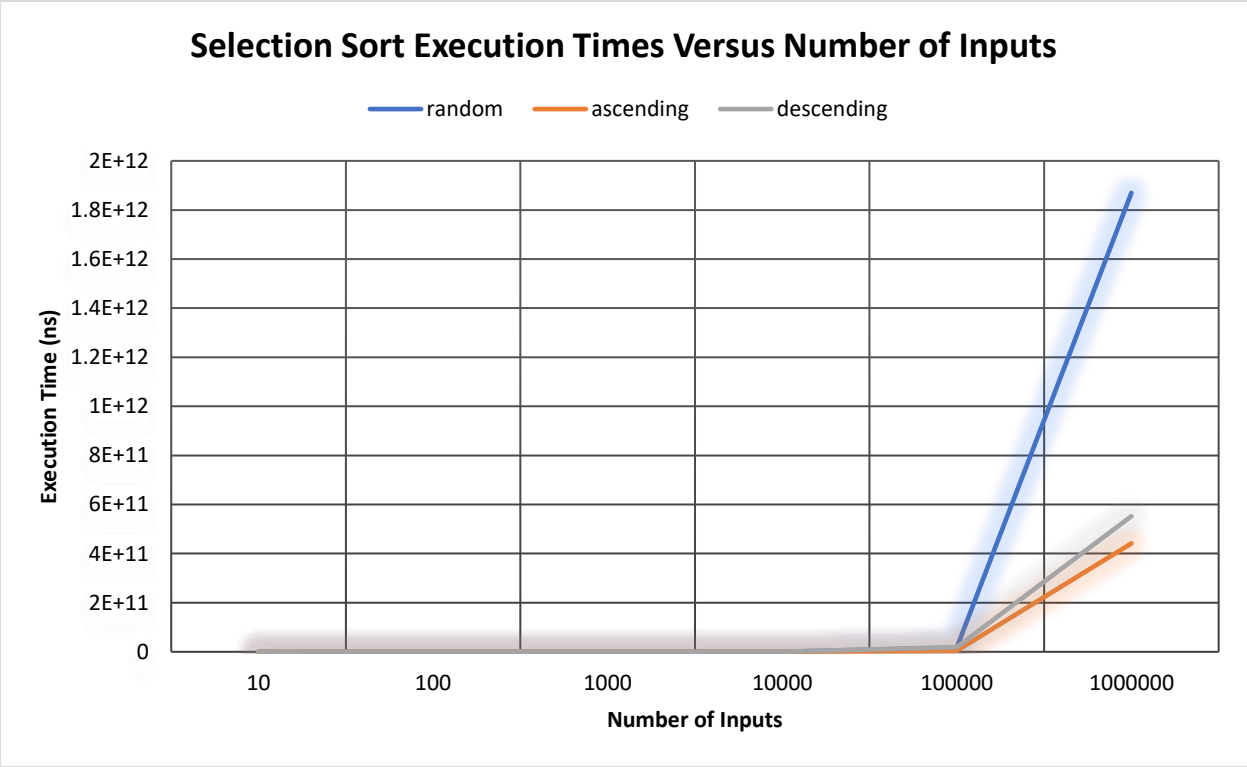
Big O	Worst-Case Complexity Analysis	
	Experimental	Theoretical
Selection	$O(n^2)$	$O(n^2)$
Insertion	$O(n^2)$	$O(n^2)$
Merge	O(n log n)	O(n log n)
Quick	O(n log n)	O(n log n)

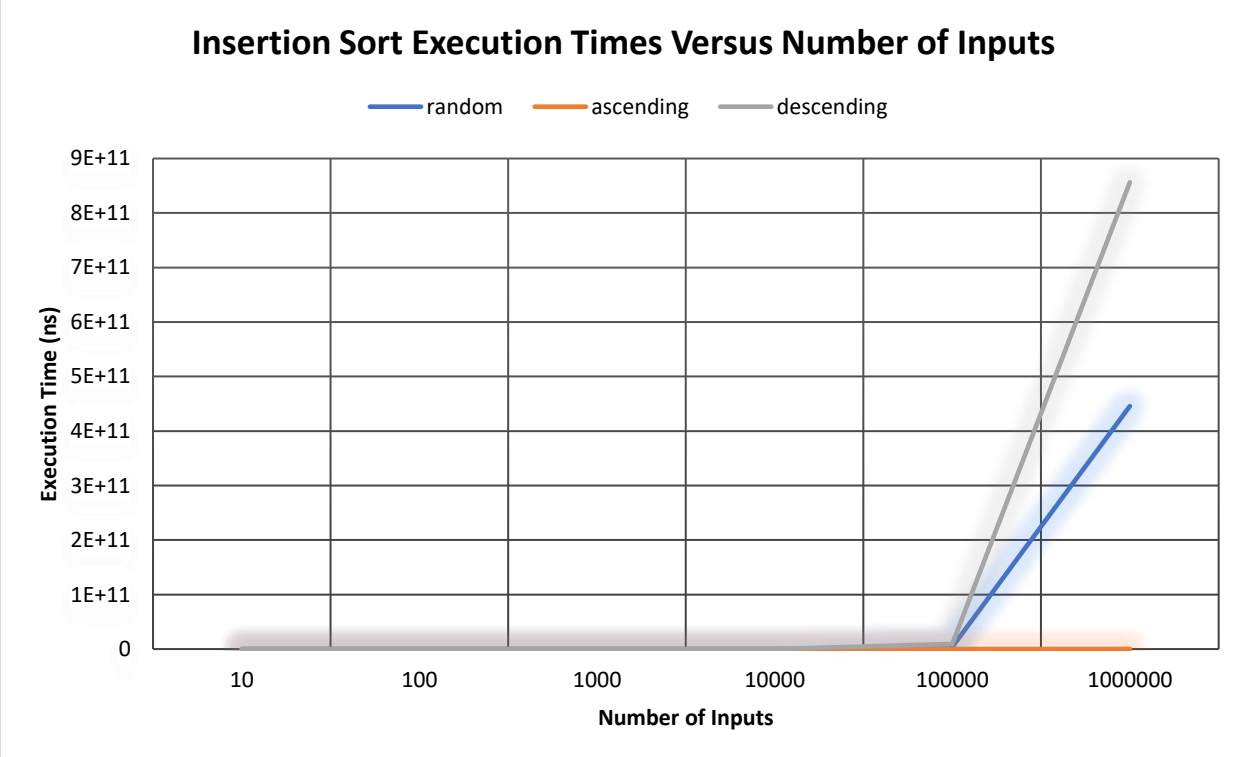
It is evident that the experimental results coincide with the theoretical values of Big-O for each algorithm. We can also see that selection and insertion sort have the same Big-O classification and so does merge and quick sort. Even though these have similar Big-O classifications they may have differences when it comes to their lower bounds since Big-O only specifies the upper bound. We can examine the graphs and table above to see the differences in performance. We can treat the ascending initial order of an array as a condition for lower bound. This clearly illustrates the difference in performance of insertion and selection sort. Insertion sort, even though it is thought to be a simpler algorithm does better than the more complex ones in this situation even more so than selection sort which has a similar Big-O classification.

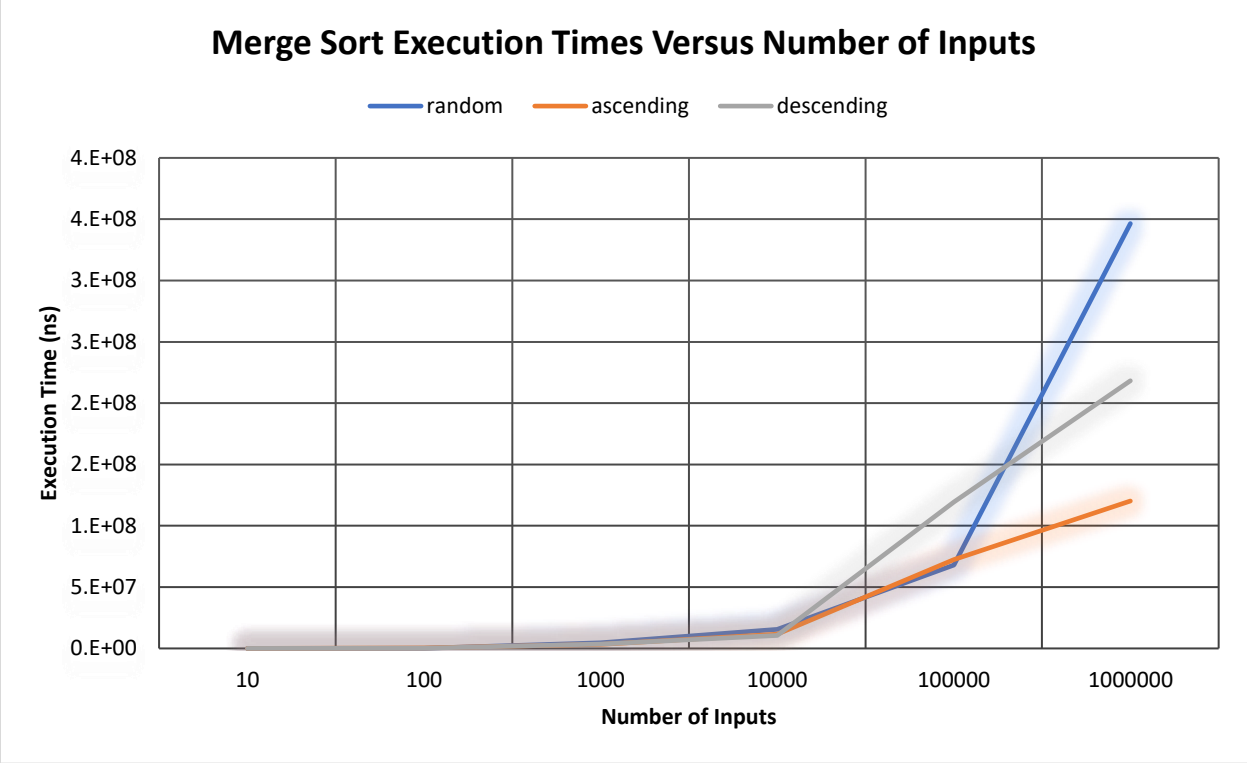
VI. Conclusion

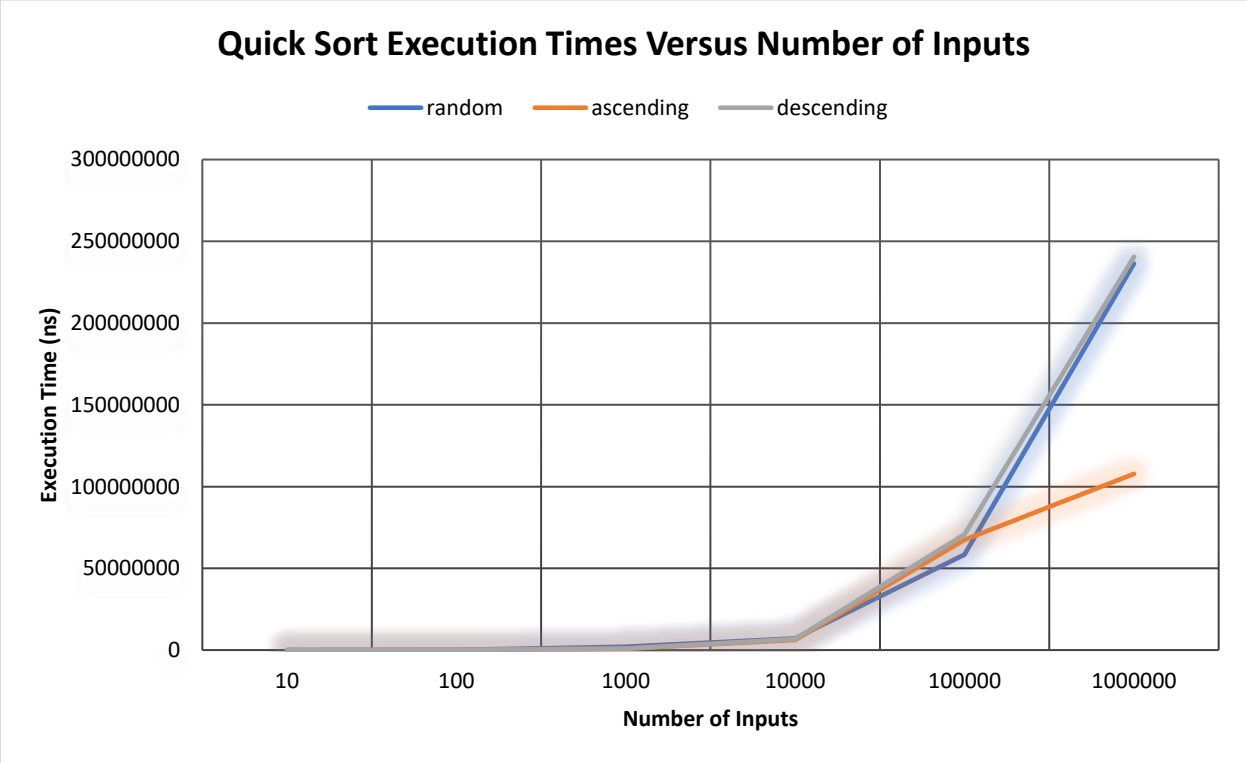
There's plenty of sorting algorithms that can be implemented. Each having their own strong and weak points. From the four algorithms examined in this lab it is ideal to use either selection or insertion sorts for smaller samples. Merge and quick sorts are very useful when dealing with larger amounts of data. There are also specific sorts that might fit a certain task better than others. If it is known that the initial list of data is relatively sorted, regardless of data size, insertion sort is the best algorithm to use (refer to tables and graphs). For most cases (average case), it is ideal to use quick or merge sorts. The main dividing factor is whether using extra memory (merge sort) does not interfere with other tasks at hand. Even though the Big-O classification for some algorithms are the same as others, when further examined for specific cases, differences in performance can be observed. It must also be kept in mind that for some applications a specific algorithm will have limited capabilities such as merge sort in an embedded system. This will be very memory expensive for a system that already lacks the excess memory to perform other tasks even more so a sorting algorithm that demands memory space to create a temporary holder.

VII. Appendix






Merge Sort Execution Times Versus Number of Inputs
random
ascending
descending
4.E+08
4.E+08
3.E+08
3.E+08
2.E+08
2.E+08
1.E+08
5.E+07
0.E+00
Execution Time (ns)
10
100
1000
10000
100000
1000000
Number of Inputs


Quick Sort Execution Times Versus Number of Inputs
random
ascending
descending
300000000
250000000
200000000
150000000
100000000
50000000
0
Execution Time (ns)
10
100
1000
10000
100000
1000000
Number of Inputs