Group 7 Project Report

December 3, 2021

Sydney, Brandon, Maria, Daniel, Eric
December 3, 2021 Test text for the report that we need to write within the new few days.

Contents

1 Part 1: Comparison of Sorting Algorithms **4**
1.1 1. Theoretical question 4

2 Data generation and experimental setup **8**
2.1 Machine Use and time Mechanism 8
2.2 Experiment execution and times report 8
2.3 Selected inputs 8
2.4 Using different input for third list 8
2.5 Execution Time Gather 9

3 Which of the five sorts seems to perform the best? **18**
3.1 The sort that perform the best 21

4 To what extent do the best, average and worst-case analyses of each sort agree with the experimental results? **21**
4.1 Insertion Sort: 21
4.2 Selection Sort: 24
4.3 Bubble Sort: 25
4.4 Merge Sort: 27

5 Is The Number of Comparisons A Good Predictor of the Execution Time? 29
5.1 Insertion Sort Correlation 29
5.2 Selection Sort Correlation 30
5.3 Bubble Sort Correlation 31
5.4 Merge Sort Correlation 32
5.5 Heap Sort Correlation 33
5.6 Quick Sort Correlation 34
5.7 Conclusion 34

6 Part 2: Brute/Inefficient method 35
6.1 PseudoCode 35
6.2 Running time analysis 35
6.3 Report of brute/inefficient method 36

7 Part 2: Efficient method 36
7.1 PseudoCode 36
7.2 Running time analysis 37
7.3 Report of Efficient method 37

List of Figures

1 Part 1: Comparison of Sorting Algorithms

The efficient method of finding all the pairs/elements equal to the sum.

1.1 1. Theoretical question

Insertion Sort:

Best Case: O(n) - Sorted Array
Average Case: O(n^2) - Random Order Array
Worst Case: O(n^2) - Reversed Array

Regarding Insertion Sort, the best case is achieved when the input array is in already sorted order. This is because the algorithm must do only one comparison for each element, so it must do n comparisons which results in a time complexity of O(n). The average case occurs when the elements in the array are in any random order. They are neither reversed nor in sorted order. This results in a time complexity of O(n^2). The worst case occurs when the input array is in reverse order. This is because each element in the array must be compared with all of the other elements. Like the average case, this also results in a time complexity of O(n^2).

Selection Sort:

Best Case: O(n^2) - Sorted Array
Average Case: O(n^2) - Random Order Array
Worst Case: O(n^2) - Reversed Array

Regarding Selection Sort, the best case, worst case, and average case all result in the same time complexity, which is O(n^2) . This is due to the fact that the algorithm will do (n-1) comparisons depending on the size of n. For example if n = 8, the algorithm will do (n-1) + (n-2) + (n-3) + (n-4) + (n-5) + ... + 1. The algorithm also uses double nested for loops, which would explain why Selection Sort has a time complexity of O(n^2) in all 3 cases.

Bubble Sort:

Best Case: O(n) - Sorted Array
Average Case: O(n^2) - Random Order Array
Worst Case: O(n^2) - Reversed Array

Regarding Bubble Sort, The best case occurs when the array is already sorted because during the first iteration of the algorithm the swap will not occur because the left element will not be greater than the right. This means that the time complexity will be O(n) because the algorithm will traverse through each element one time. Unless the array is already in sorted order the best case, worst case, and average case will all be O(n^2). This is because the algorithm will traverse through each element and compare it with all other elements and also swap them if left ¿ right.

Merge Sort:

Best Case: O(n log n) - Sorted Array
Average Case: O(n log n) - Random Order Array
Worst Case: O(n log n) - reversed Array

Regarding Merge Sort, it is a divide and conquer algorithm, and a re-

currence needs to be resolved to estimate its time consumption. In the case of Merge Sort, an important feature is that its efficiency regardless of the case (best, worst, or average) the Merge Sort will always be O(nlogn). This is because splitting the problem always generates two subproblems half the size of the original problem (2T(n/2)). The algorithm bases the ordering on successive merge runs, a routine that joins two ordered parts of an array into an ordered one. Also, it is important to remember that the merge algorithm is (O(n1 + n2) = O(n).

Quick sort:

Best Case: O(nlogn)
Occurs when the chosen pivot splits the vector into two size subvectors equal to n/2.
T(n) = 2T(n/2) + n – 1
T(1) = T(0) = 1
Solving (divide and conquer):
T(n) = O(n log n).

Average Case: O(n^2)
It is the average of all possible cases of pivot placement after the partition.
T(n) = O (n log (n)).

Worst Case: O(n^2)
Occurs when the pivot divides the vector into two subvectors, one with zero.
element and another with n – 1 elements.
Complexity:
T(n) = T(n - 1) + n - 1
T(0) = 1

Solving:

T(n) = O(n^2). Regarding Quick Sort, it is an efficient divide and conquer sorting algorithm. The algorithm bases the ordering on successive partitioning runs, which means it gets a pivot and positions it in the array in such a way that elements less than or equal to the pivot are on its left and the largest are on its right. Despite being of the same complexity class as Merge Sort and Heap Sort, Quick Sort is the fastest of them, as its constants are smaller. However, it is important to note beforehand that, in its worst case, Quick Sort is O(n^2), while Merge Sort and Heap Sort guarantee nlogn for all cases. A good thing about quick sort is that there are simple strategies with which we can minimize the chances of the worst-case happening. Also, Quick Sort is not stable, which means that it does not guarantee the order of the repeated elements, but does not need to use the position as a tie-breaker, it can be a second defined key or a third, and so on.

Heap sort:

Best Case: O(n log n) - Sorted Array
Average Case: O(n log n) - Random Order Array
Worst Case: O(n log n) - reversed Array

Regarding Heap sort, it uses a data structure called binary heap to order elements as you insert them into the structure. Thus, at the end of insertions, elements can be successively removed from the heap root, in the desired order. Binary heaps are representations of complete binary trees (all levels are filled completely but not the last level, that can or cannot be filled completely). Heaps are complete binary trees, and it is possible to store them in arrays. Exist two types of binary trees, being min-heap that all the child nodes are ¿ than their parent, and the root node holds a small value. Max heap on the other hand has all the parents ¿ than the child and makes the root the biggest node. The big-O for the average performance of a heap sort algorithm is O(nlog n) and the space complexity is O(n).

Counting Sort:

Best Case: O(n+k)
Average Case: O(n+k)

Worst Case: O(n+k)

Counting Sort is a non-comparison-based sorting algorithm, but it is efficient in execution time even though it needs more memory than the algorithms that use comparison methods (Selection, Insertion, Bubble Sort etc..). The idea of the Counting sort is that an integer value can be mapped to the index of the same value in an auxiliary array. This strategy prevents us, at first, from ordering a sequence with negative numbers, since the smallest index in an array is 0.

Radix Sort:

Best Case: O(d*(n+k))
Average Case:O(d*(n+k))
Worst Case: O(d*(n+k))

Radix sort is another non-comparison based algorithm, utilizing counting sort. This means that it does not directly compare the elements to figure out their relation to one another. Instead, it finds the length of the largest element in the array, and sets it equal to 'd'. It does this because it goes from the outside in, of all the elements. For example, if all elements were in the hundreds, d would be equal to 3. Radix sort starts from the left, also known as the ones, and then to the tens and so on. It does this using a for loop until it reaches 'd'. During every iteration it calls counting sort to do the sorting. This is why the time complexity is O(d*(n+k)).

2 Data generation and experimental setup

2.1 Machine Use and time Mechanism

Dell laptop specs: Intel ® Core (™) i5 - 6300U CPU @ 2.40GHz 2.50 Ghz 16.0 GB Installed Ram. For the time mechanism we are using the ctime library to determine the execution time.

2.2 Experiment execution and times report

We repeated the experiment five times for each sorting algorithms. The execution times being reported will be below in 8 different tables for each algorithm.

2.3 Selected inputs

We select these inputs to see the real difference of time it takes to execute. All functions have the first list being 100 where the most of the algorithm will give zero and then the second list would be much bigger with 10,000 to see some difference to occur for it execute. The last input is the third list being the largest that we could run without overflowing so that we could see the actual difference of time execution.

2.4 Using different input for third list

The only same inputs that are used for all sorting algorithms are the first two lists of the inputs which are the 100 and 10,000. The third list for most of the functions has a different size. The reason for different input sizes is so we don't overflow the functions as each one has a different limit before overflows.

2.5 Execution Time Gather

The time that was gather is place in data tables that are shown below with the name of the algorithms in the top left corners for each table.

Insertion Sort:	Sort (sec)	HalfSort (sec)	ReverseSort (sec)
100 1st exe	0	0	0
10000 1st exe	0	0.095	0.472
70000 1st exe	0.001	1.476	13.878
\			
100 2nd exe	0	0	0
10000 2nd exe	0	0.042	0.496
70000 2nd exe	0	1.392	12.215
\			
100 3rd exe	0	0	0
10000 3rd exe	0	0.051	0.529
70000 3rd exe	0.001	1.32	13.674
\			
100 4th exe	0	0	0
10000 4th exe	0	0.041	0.476
70000 4th exe	0.001	1.563	13.363
\			
100 5th exe	0	0	0
10000 5th exe	0	0.071	0.532
70000 5th exe	0	1.448	12.51

Figure 1: Insertion Sort Data Table

Selection Sort	Sort (sec)	HalfSort (sec)	ReverseSort (sec)
100 1st exe	0	0	0
10000 1st exe	0.398	0.284	0.21
70000 1st exe	7.388	7.451	7.018
\			
100 2nd exe	0	0	0
10000 2nd exe	0.231	0.24	0.223
70000 2nd exe	8.103	7.419	7.63
\			
100 3rd exe	0	0	0
10000 3rd exe	0.251	0.219	0.193
70000 3rd exe	7.294	7.946	7.168
\			
100 4th exe	0	0	0
10000 4th exe	0.201	0.238	0.187
70000 4th exe	7.257	7.255	7.821
\			
100 5th exe	0	0	0
10000 5th exe	0.37	0.3	0.271
70000 5th exe	8.255	7.498	7.123

Figure 2: Selection Sort Data Table

Counting Sort:	Sort (sec)	HalfSort (sec)	ReverseSort (sec)
100 1st exe	0	0	0
10000 1st exe	0	0	0
70000 1st exe	0.002	0.001	0.002
\			
100 2nd exe	0	0	0
10000 2nd exe	0	0	0
70000 2nd exe	0.002	0.001	0.002
\			
100 3rd exe	0	0	0
10000 3rd exe	0	0	0
70000 3rd exe	0.003	0.001	0.004
\			
100 4th exe	0	0	0
10000 4th exe	0	0	0
70000 4th exe	0.003	0.002	0.002
\			
100 5th exe	0	0	0
10000 5th exe	0	0	0
70000 5th exe	0.001	0.004	0.002

Figure 3: Counting Sort Data Table

Radix Sort:	Sort (sec)	HalfSort (sec)	ReverseSort (sec)
100 1st exe	0.001	0	0
10000 1st exe	0.001	0.002	0.003
45000 1st exe	0.008	0.009	0.01
\			
100 2nd exe	0.001	0	0
10000 2nd exe	0.002	0.003	0.004
45000 2nd exe	0.01	0.009	0.011
\			
100 3rd exe	0	0	0.001
10000 3rd exe	0.002	0.001	0.002
45000 3rd exe	0.011	0.014	0.013
\			
100 4th exe	0	0	0
10000 4th exe	0.002	0.002	0.002
45000 4th exe	0.012	0.014	0.012
\			
100 5th exe	0	0	0
10000 5th exe	0.001	0.002	0.002
45000 5th exe	0.007	0.01	0.006

Figure 4: Radix Sort Data Table

Merge Sort:	Sort (sec)	HalfSort (sec)	ReverseSort (sec)
100 1st exe	0.001	0.001	0
10000 1st exe	0.007	0.005	0.007
60000 1st exe	0.046	0.04	0.035
\			
100 2nd exe	0.001	0	0
10000 2nd exe	0.004	0.005	0.006
60000 2nd exe	0.055	0.052	0.043
\			
100 3rd exe	0.001	0	0
10000 3rd exe	0.004	0.005	0.003
60000 3rd exe	0.027	0.026	0.025
\			
100 4th exe	0	0	0
10000 4th exe	0.005	0.004	0.004
60000 4th exe	0.026	0.031	0.026
\			
100 5th exe	0	0	0
10000 5th exe	0.006	0.01	0.004
60000 5th exe	0.051	0.038	0.038

Figure 5: Merge Sort Data Table

Bubble Sort:	Sort (sec)	HalfSort (sec)	ReverseSort (sec)
100 1st exe	0	0	0
10000 1st exe	0.267	0.306	0.594
70000 1st exe	7.819	9.973	24.697
\			
100 2nd exe	0	0	0
10000 2nd exe	0.182	0.362	0.57
70000 2nd exe	7.694	10.104	25.279
\			
100 3rd exe	0	0	0
10000 3rd exe	0.218	0.389	0.577
70000 3rd exe	7.762	11.16	26.353
\			
100 4th exe	0	0	0
10000 4th exe	0.233	0.351	0.617
70000 4th exe	7.56	10.086	26.258
\			
100 5th exe	0	0	0.001
10000 5th exe	0.533	0.379	0.58
70000 5th exe	7.239	10.262	25.674

Figure 6: Bubble Sort Data Table

Heap Sort:	Sort (sec)	HalfSort (sec)	ReverseSort (sec)
10000 1st exe	0.006	0.005	0.006
100000 1st exe	0.067	0.08	0.076
1000000 1st exe	0.747	0.708	0.721
\			
10000 2nd exe	0.005	0.005	0.005
100000 2nd exe	0.06	0.058	0.057
1000000 2nd exe	0.683	0.69	0.783
\			
10000 3rd exe	0.007	0.008	0.004
100000 3rd exe	0.062	0.063	0.074
1000000 3rd exe	0.668	0.709	0.691
\			
10000 4th exe	0.005	0.005	0.005
100000 4th exe	0.061	0.059	0.06
1000000 4th exe	0.737	0.699	0.778
\			
10000 5th exe	0.006	0.005	0.007
100000 5th exe	0.07	0.067	0.066
1000000 5th exe	0.721	0.672	0.697

Figure 7: Heap Sort Data Table

Quick Sort:	Sort (sec)	HalfSort (sec)	ReverseSort (sec)
100 1st exe	0	0	0
10000 1st exe	0.186	0.001	0.28
25000 1st exe	0.991	0.012	1.683
\			
100 2nd exe	0	0	0
10000 2nd exe	0.23	0.003	0.341
25000 2nd exe	0.95	0.011	1.725
\			
100 3rd exe	0	0	0
100003rd exe	0.176	0.002	0.422
25000 3rd exe	0.982	0.011	1.698
\			
100 4th exe	0	0	0
10000 4th exe	0.261	0.005	0.371
25000 4th exe	0.973	0.013	1.652
\			
100 5th exe	0	0	0
10000 5th exe		0.003	0.33
25000 5th exe	0.0947	0.01	1.663

Figure 8: Quick Sort Data Table

3 Which of the five sorts seems to perform the best?

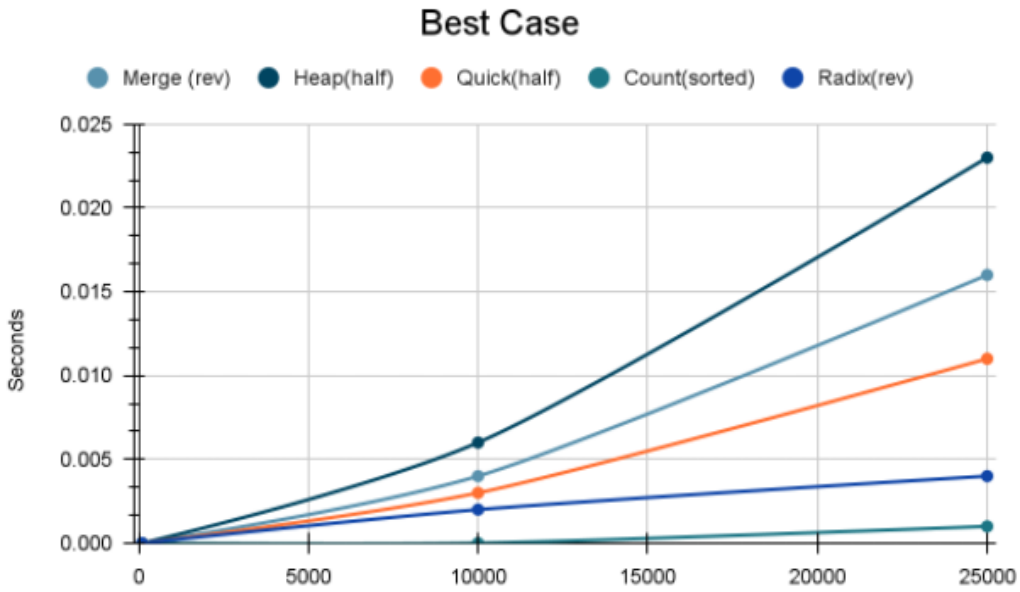

Figure 9: Best Case Graph

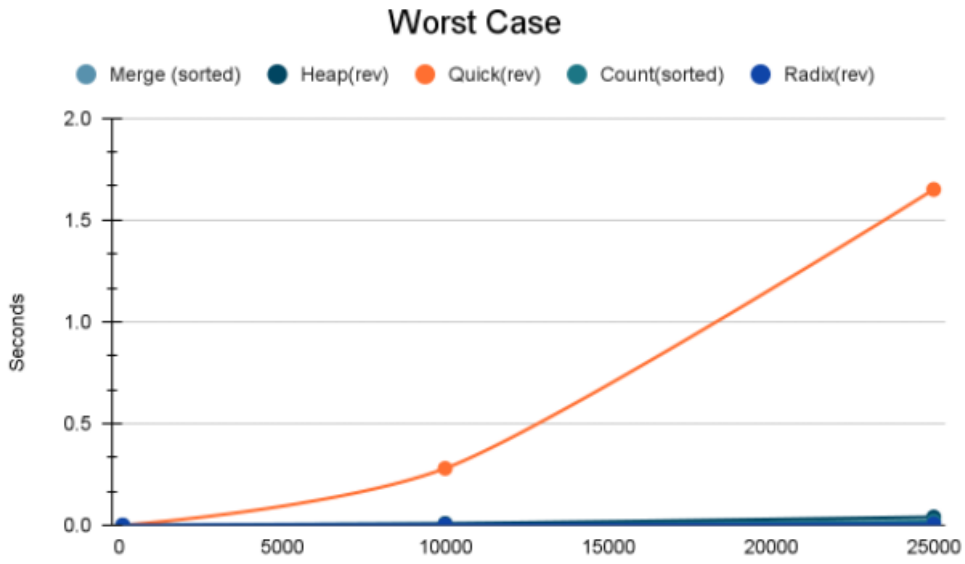

Figure 10: Worst Case Graph

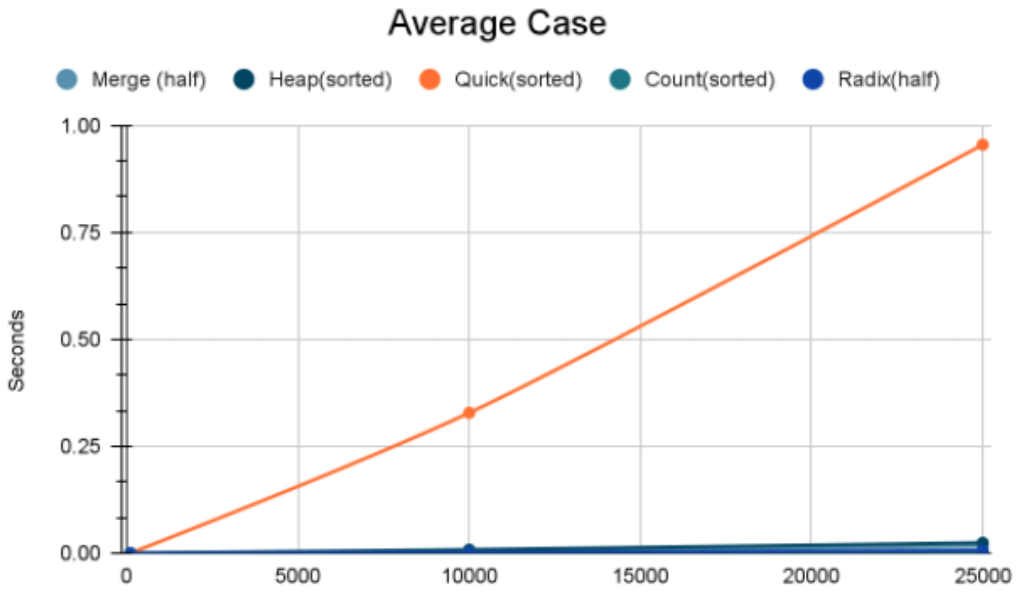

Figure 11: Average Case Graph

3.1 The sort that perform the best

Counting Sort performs the best in comparison to the other sorting algorithms which are the mergesort, heapsort, quicksort, and radix sort. Even when Quicksort at it's best case performs O(nlogn) in addition with Merge and Heap. Counting Sort runs in linear time, resulting in it to be the quickest sorting algorithm. Taking in the input of a sorted array ranging from 100 to 25000 elements, Counting Sort outputs a running time no greater than 0.001 seconds. While Quicksort took in a half-sorted array to execute in it's best case, it still ran at a time complexity of O(nlogn). Furthermore, Quicksort performance as seen from the graph and execution time performed the worst in comparison to the other sorts, O(n^2) to be exact. Additionally, Counting Sort constantly remains the best in terms of running time in all three cases. In addition to that, Radix Sort as seen from the graph follows slightly behind Counting Sort. The time-complexity of Radix Sort is O(d*(n+k)) for all three cases.

4 To what extent do the best, average and worst-case analyses of each sort agree with the experimental results?

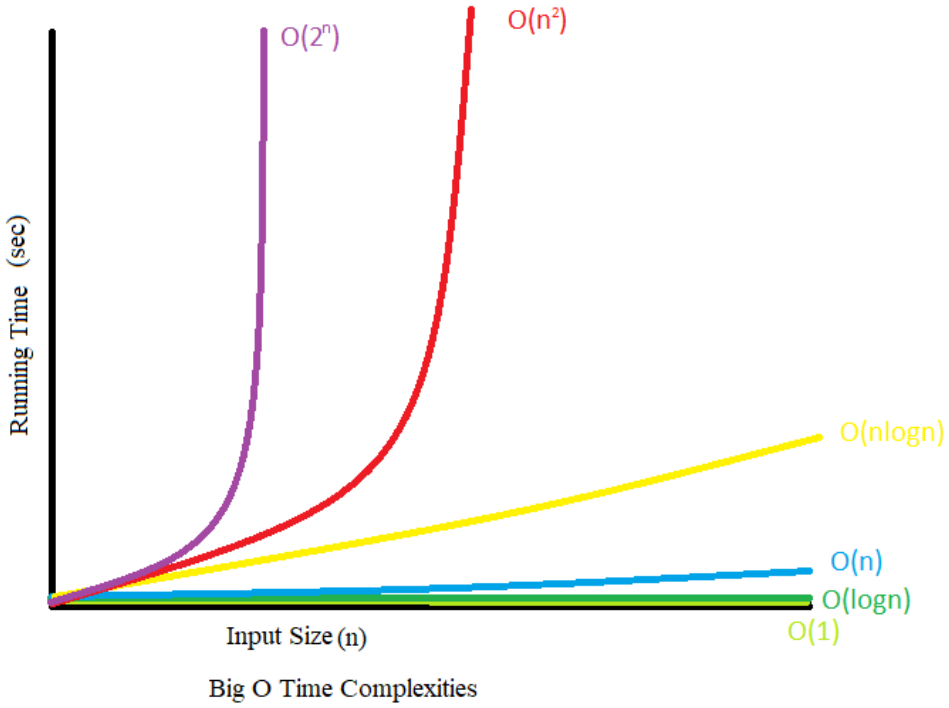

Figure 12: Asymptotic Notations

The graph above, Big O Time Complexities, will be used to compare how the graphed experimental results relate to the actual expected worst cast, best case, and average case of each sorting algorithm. For each sorting algorithm, the reversed input array line will correspond to the worst case, the half-sorted array line will correspond to the average case, and the already sorted array line will correspond to the best case.

4.1 Insertion Sort:

As previously mentioned, the worst case, best case, and average case analysis for Insertion Sort are as follows:

Best Case: O(n) - Sorted Array
Average Case: O(n^2) - Random Order Array
Worst Case: O(n^2) - Reversed Array

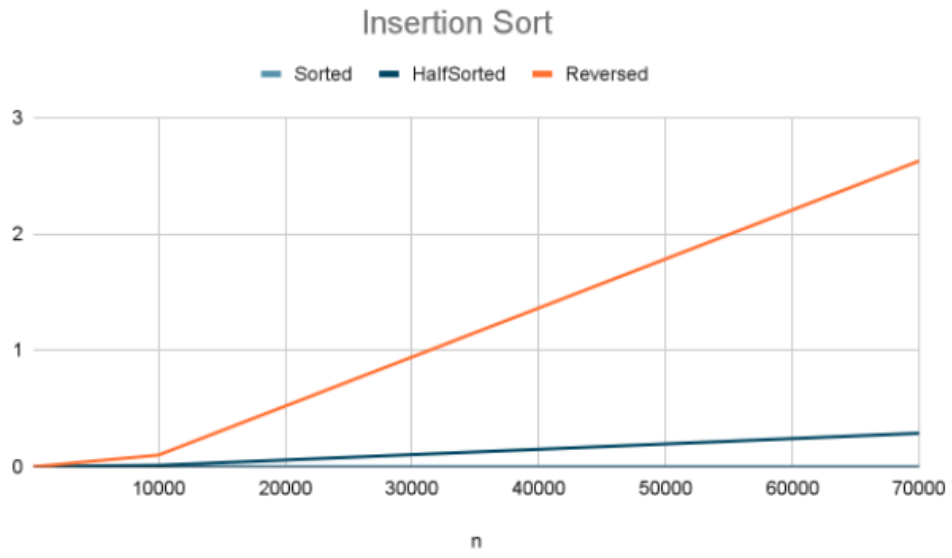

Figure 13: Insertion Sort Graph

The reversed array input results in the worst case for the Insertion Sort algorithm which is O(n^2). The orange line, which is from the reversed input array, corresponds to the worst case. When comparing it to O(n^2) in the Big O Time complexities graph it appears to agree with the worst case analysis of Insertion Sort which as is known, is O(n^2). As the input size increases the running time increases quadratically.

The dark blue line from the half-sorted array is also expected to be O(n^2) because it corresponds to the average case which as is known is also O(n^2).

However, when comparing the Half Sorted line to the O(n^2) line in the Big O Time complexities graph they do not appear to be the same. In fact, the Half Sorted line appears to relate more closely to the O(nlogn) line. When it comes to the average case, the experimental results do not appear to agree with the expected time analysis of O(n^2). The reason for this is most likely due to the fact that the array is already half sorted. As is known the time of insertion sort for an already sorted array is O(n) which is a bit faster than O(n^2). Knowing this, it is clear that the algorithm ran quicker through the first half of the array than it did in the reversed array. If the array were truly random as expected for the average case, then time complexity would most likely correspond to the expected O(n^2). Of course, there is an argument to be made that a half sorted array is within the realm of possibilities when it comes to randomly ordered arrays. So, when it comes to whether or not the average case of Insertion Sort agrees with the experimental results that were derived, the answer is that it depends on how truly random the array is.

Moving on to the best case of Insertion Sort, the already sorted array line is a flat line all the way across the board. This appears to be completely in line with O(n) on the Big O Time complexities graph. As previously mentioned the expected best case of Insertion Sort is O(n) or linear time, which occurs when the input is in an already sorted order. This means that the algorithm will only do n comparisons. So when it is given a sorted array of size 70000, it will do 70000 comparisons, whereas in the worst case when given the same 70000 sized array but in reverse order, it will do 4,900,000,000 comparisons. As such, the running time for an already sorted array should have a much lower increase as the input size grows than the worst case. This is clearly reflected in the experimental graph. When it comes to the best case of insertion sort, the experimental data appears to completely agree with the asymptotic analysis for the best case of Insertion Sort which is O(n).

4.2 Selection Sort:

As previously mentioned, the worst case, best case, and average case analysis for Selection Sort are as follows:

Best Case: O(n^2) - Sorted Array
Average Case: O(n^2) - Random Order Array
Worst Case: O(n^2) - Reversed Array

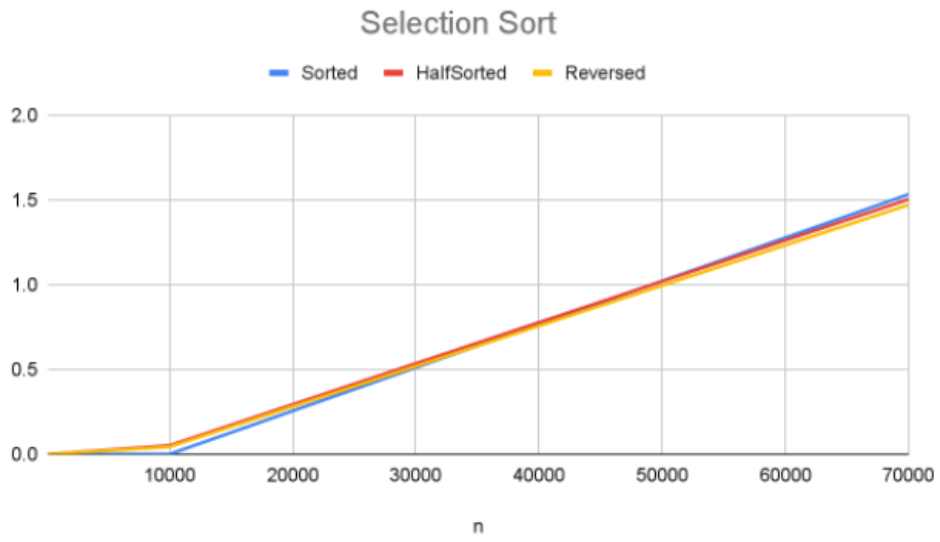

Figure 14: Selection Sort Graph

Regarding Selection Sort, the worst, best, and average case analysis are all O(n^2). Knowing this, one can reasonably expect the experimental results of each array to be very close when graphed and also correspond to the O(n^2) curve shown in the Big O Time Complexities graph. As can be seen from the Selection Sort graph above which shows the experimental results, each line is indeed very close and similar in the way the execution time increases as the input size increases. This clearly indicates that the algorithm had a very similar execution time for the reverse order array, the half-sorted array, and also the sorted array. The execution times for each array also appear to increase

quadratically as the input sizes increase similar to O(n^2) on the Big O Time Complexities graph. One can safely say that in the case of Selection Sort, the experimental results agree with the best, average and worst-case analysis.

4.3 Bubble Sort:

As previously mentioned, the worst case, best case, and average case analysis for Bubble Sort are as follows:

Best Case: O(n) - Sorted Array
Average Case: O(n^2) - Random Order Array
Worst Case: O(n^2) - Reversed Array

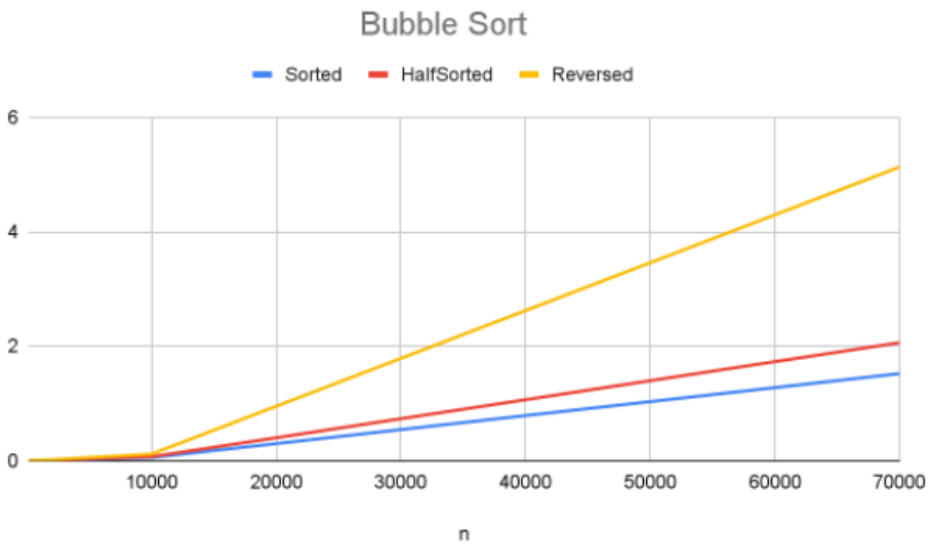

Figure 15: Bubble Sort Graph

To begin, it must be mentioned that the worst case, best case, and average case analysis for Bubble Sort is the same as the worst case, best case, and average case analysis for Insertion Sort. As such, one can expect very similar experimental results to those seen with the Insertion Sort algorithm.

The reversed line in the Bubble Sort graph is very similar to that seen in the Selection Sort graph. Similarly, as the input size grows, the execution times appear to grow quadratically. Furthermore, when comparing the reversed line to the $O(n^2)$ line on the Big O Time Complexities graph, the reversed line has a dramatic increase as the input size grows which corresponds to $O(n^2)$. It can safely be said that in the worst case of Bubble sort, the experimental results agree with the worst case analysis.

Moving on to the average case, something strange happens. The half-sorted line, which corresponds to the average case, and the sorted line, which corresponds to the best case, are very close in their execution times over input sizes. Similar to Insertion Sort, the average case appearing faster than the expected $O(n^2)$ can be attributed to the fact that the input was a half sorted array. Again, an argument can be made that a half sorted array is a possibility in random arrays. For the average case, the experimental results do not appear to agree with the average case analysis. However, if each execution time was doubled to account for the half of the array that was sorted, it is certain that the line would be very close to that of the worst case. Thus, it can be said that the experimental results may agree with average case analysis if the array was in a truly randomized order, but even then the results may vary in the way that they could be even faster or slower.

As for the best case, once again something strange happens. When comparing the sorted line from the Bubble Sort graph to the Big O Time Complexities graph, it appears to correspond more similarity to the O(nlogn) line rather than the expected O(n) line. The sorted line should also be similar to the best case sorted line seen in the Selection Sort graph. Once again this is not the case. So what did happen? To explain this the Pseudo-Code of the utilized Bubble Sort algorithm will be shown below.

```
SKbubbleSort (A, n)
    for i ← 1 to n
        for j ← 0 to n - i
            if A[j] > A[j + 1]
                Do A[j] ⟷ A[j + 1]
```

In order to get the best case from an already sorted array, Bubble Sort needs some sort of flag to remember if any swapping has actually occurred. With such a flag in place, if no swapping occurred then the algorithm will quickly know that the array is already sorted and will quit after at most the

second iteration. If the algorithm utilized in the experiment had something like a boolean to check if swapping has occurred then the experimental results would probably agree with the best case analysis, however in the case of the Bubble Sort algorithm above, the experimental results do not agree with the best case analysis of O(n) and seem to more closely correspond to O(nlogn).

4.4 Merge Sort:

Best Case: O(n log n) - Sorted Array
Average Case: O(n log n) - Random Order Array
Worst Case: O(n log n) - reversed Array

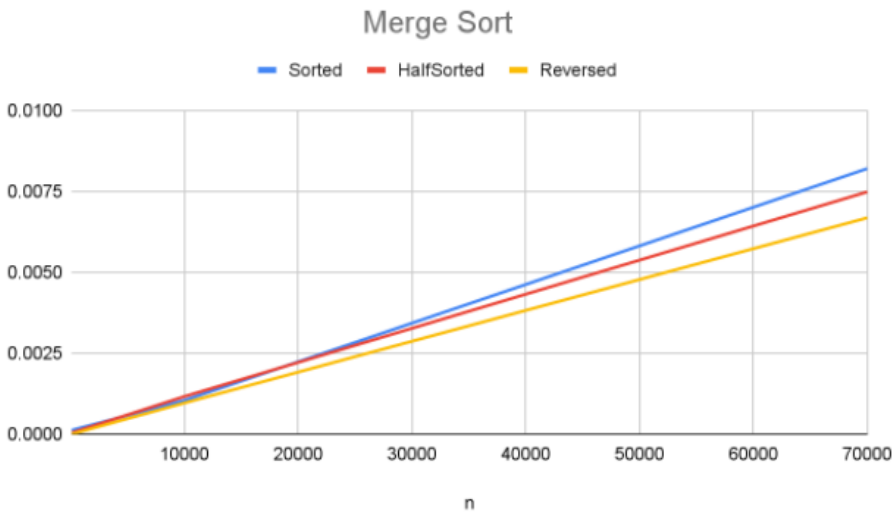

Figure 16: Merge Sort Graph

Regarding Merge Sort, the best, worst, and average case analysis are all O(nlogn). This means that one can reasonably expect that the Sorted, Half Sorted, and Reversed lines will appear very similar when graphed in their relation of execution times over increasing input sizes.. As can be seen in the

Merge Sort graph above, this appears to definitely be the case. Furthermore, when comparing each line to the O(nlogn) line in the Big O Time Complexities graph, they appear to perfectly coincide. In the case of Merge Sort it is overwhelmingly clear that the best, average and worst-case analyses agree with the experimental results.

5 Is The Number of Comparisons A Good Predictor of the Execution Time?

5.1 Insertion Sort Correlation

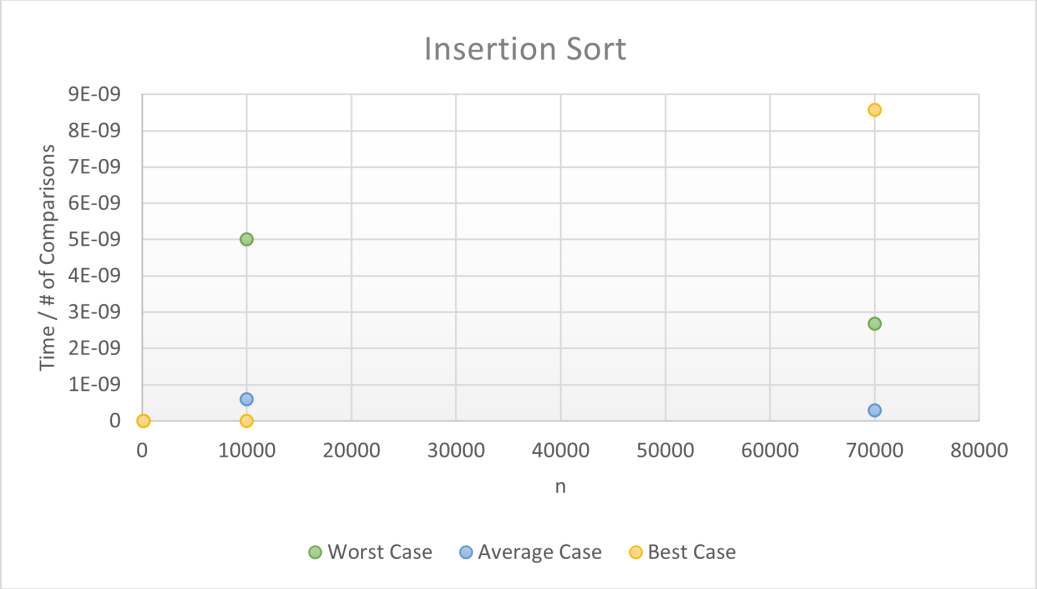

Figure 17: Insertion Sort Data Correlation

Regarding Insertion Sort, in Figure 12, there appears to be a small positive correlation between the execution time divided by the number of comparisons vs n when n is less than or equal to 10000. However that correlation becomes a weak positive correlation as the n grows.

5.2 Selection Sort Correlation

Figure 18: Selection Sort Data Correlation

Regarding Selection Sort, in Figure 13, there does appear to be a Curvilinear correlation between the execution time divided by the number of comparisons vs n.

5.3 Bubble Sort Correlation

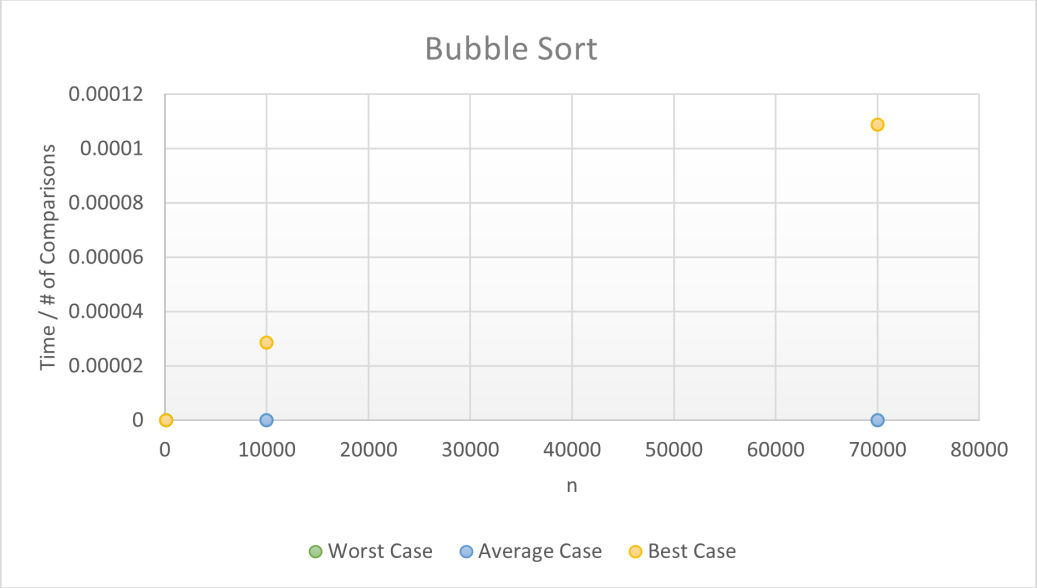

Figure 19: Bubble Sort Data Correlation

Regarding Bubble Sort, in Figure 14, there does appear to be a weak positive correlation.

5.4 Merge Sort Correlation

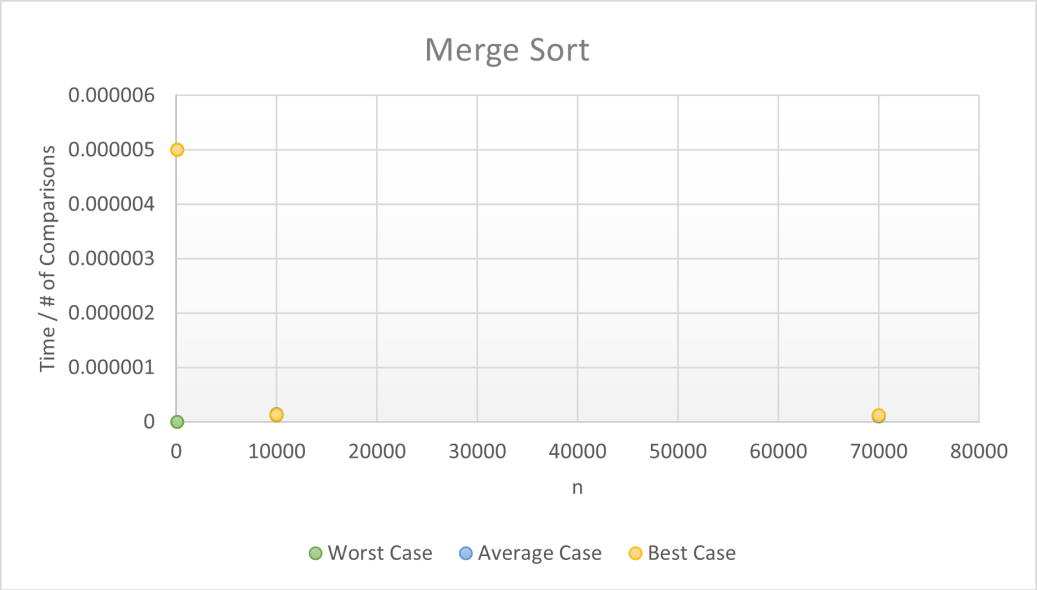

Figure 20: Merge Sort Data Correlation

Regarding Merge Sort, in Figure 15, it could be argued that there is a very weak negative correlation. However, there also appears to be no correlation at all.

5.5 Heap Sort Correlation

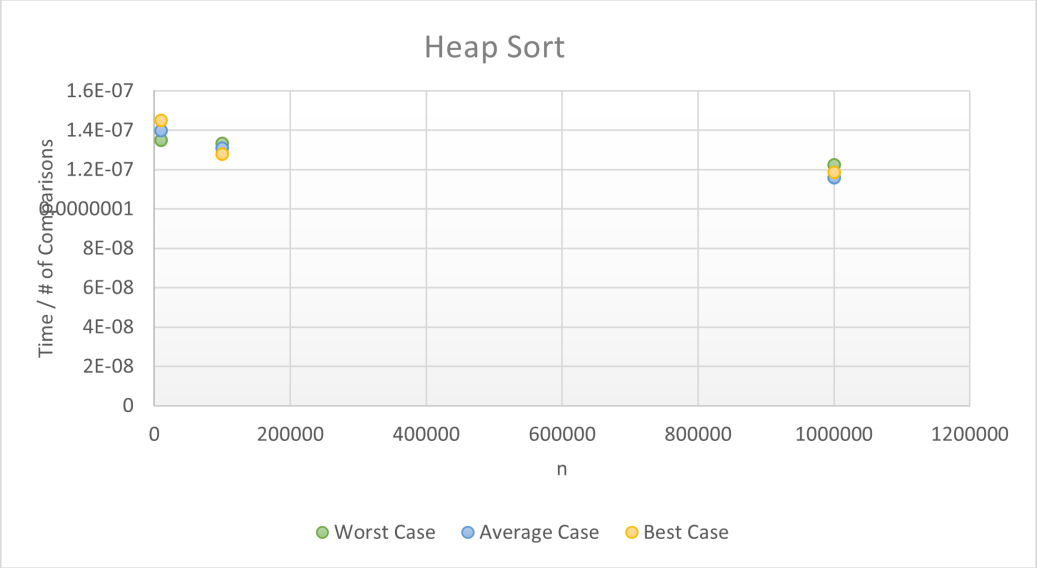

Figure 21: Heap Sort Data Correlation

Regarding Heapsort, in Figure 21, similar to Selection Sort there appears to be a Curvilinear correlation.

5.6 Quick Sort Correlation

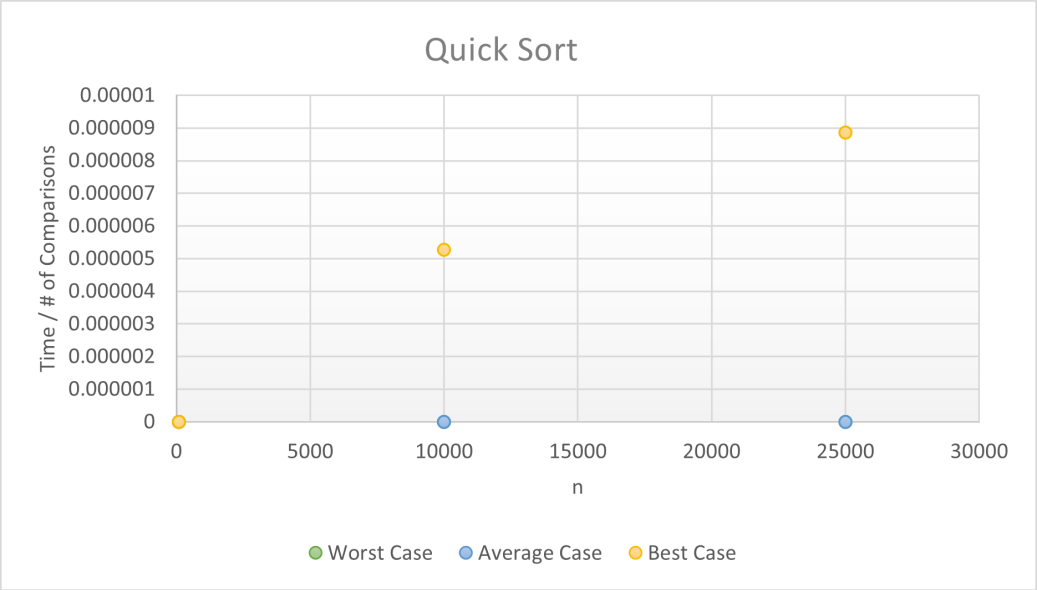

Figure 22: Quick Sort Data Correlation

Regarding Quicksort, in Figure 22, there appears to be a weak positive correlation. Although it is a weak correlation, the data points are evenly spread indicating that a relationship does exist.

5.7 Conclusion

To answer the question, is the number of comparisons truly a good predictor of execution times? For each comparison sorts scatter plot of the execution time divided by the number of comparisons vs n, there appears to be some form of correlation. For Insertion Sort, there was a weak positive correlation, for Selection Sort there was a curvilienar correlation, for Bubble Sort there a weak positive correlation, and finally for Merge Sort there was a very weak negative correlation. Although these correlations are weak they are present. These correlations would also most likley become stronger with more data sizes if they follow the trend of each comparison sorts correlation. To conclude, the number of comparisons is a good predictor of execution times.

6 Part 2: Brute/Inefficient method

The brute/inefficient method of finding all the pairs/elements equal to the sum.

6.1 PseudoCode

```
ALG: BruteSum(A,n,x)
        total←0
        for i←0 to n
            for a←i to n
                do sum←A[i]+A[a]
                do if sum==x
                        total ++
```

6.2 Running time analysis

```
ALG: BruteSum(A,n,x)
        total←0                          -1
        for i←0 to n                      -n+1
            for a←i to n                  -n+1
                do sum←A[i]+A[a]          -n
                do if sum==x             -1
                        total ++          -1
```

1+n+n+n+1+1=
Removing Constants:
(n)+(n)+n
Keeping the highest order term which there are two:
The time complexity is O(n^2)

6.3 Report of brute/inefficient method

The algorithm takes A for the list of array and n is taking the size of it while x is the value that we need to find the sum for. The total is to figure out if there is at least one sum that is found and the running time would be constant. The first for loop would start at the first element of the array list and then would go into the next for loop which would be considered a nested loop. The first for loop running time would be n+1 as it will go through the whole list. As soon as the second loop starts going through the function it will check to see if there is a sum that we are looking for.

The function is added from the first element and while the second element is going through the list and being added with the first element of the array to see if the sum is found. The time for this would n as it will continue to be active as the second for loop keeps going through the list. The if list is to see if the sum is found and would just be a constant as it isn't always executed and within the if function the total is being added by one which is also a constant. After the second for loop is done then the first loop will go onto the next element and repeat the second for loop from a new starting point from the first for loop. Which will give the time complexity of the function o(n^2) because of the two for loops.

7 Part 2: Efficient method

The efficient method of finding all the pairs/elements equal to the sum.

7.1 PseudoCode

```
ALG: SUM-CHECK(A, x, n)
        found_pair←false
        unordered_set HashTable
            for i←0 to n
                do y← x - A[i]
                do if HashTable.find(y) != HashTable.end()
                        then do if A[i] + y = x
                            then found_pair ← true
                            return found_pair
            HashTable.insert(i)
```

7.2 Running time analysis

```
ALG: SUM-CHECK(A, x, n)
        found_pair←false                                          -n
        unordered_set HashTable                                   -1
            for i←0 to n                                          -n + 1
                do y← x - A[i]                                    -n
                do if HashTable.find(y) != HashTable.end()        -1
                        then do if A[i] + y = x                   -n
                                return true                       -n
                HashTable.insert(i)                               -1
        return false                                              - n
```

= n + 1 + (n + 1) + n + 1 + n + n + 1 + n
= n + 1 + n + 1 + n + 1 + n + n + 1 + n
= 6n + 4
Removing Constants:
= n
Keep the highest order term: n
The time complexity is O(n) or Linear time

7.3 Report of Efficient method

The algorithm SUMCHECK takes a vector or array A, an integer x, which is the sum, and an integer n which is the size of A. A is a vector of type integer that holds a list of numbers. x is the desired sum that the algorithm will check if vector A has a pair that sums to x. First a boolean called "found_pair" is initialized to false. This boolean will be used to return true if a pair is found or return false if a pair is not found. Then an empty unorderd_set named "HashTable" is created. Next, a for loop will go through each element in vector A starting from element at index 0 to element at index n. A new integer y will then be set to the difference of x and the current element at element A[i]. At this point, y is a possible half of the pair needed to sum to x. The next part of the algorithm will check if the HashTable contains the other needed number. Moving on, An if statement will then check if the unordered_set contains this half of the possible pair by utilizing the .find() function which either returns an iterator to the found element or an iterator to .end(). Then the algorithm will then add y to the current element A[i]

from the for loop to check if y and A[i] sums to x. If it does sum to x then a pair exists in A and the algorithm will set found_pair to true and return. If y and A[i] does not sum to x then the algorithm will insert A[i] into the undordered_set. This process will repeat until each element in A[i] has been checked and a pair has not been found or until a pair is found. This results in a time complexity of O(n).