CSCI 115 Project Report

Zeke Branham, Adrian Dalena, Jonathan Garza, Ahmad Kirkland, Ian Wilson, Dr. Matin Pirouz

---

1. Introduction

The purpose of this project was to conduct theoretical analyses of the performance of various sorting algorithms under specific testing scenarios. Specifically, the theoretical discussion focused on the best, average, and worst-case scenarios for Bubble, Selection, Insertion, Heap, Merge, Quick, Counting, and Radix sort. Our primary goal for these discussions was to see how and to what extent the properties of input data had with each algorithm; first-hand analysis of these scenarios should help us reach this goal by bringing insight into the nuance behind each algorithm beyond our typical notions of how each will perform.

2. Approach

In our approach to this project, we needed to break down the work into manageable parts in order to satisfy all of the requirements in a timely manner. We started with an analysis of our coding requirements. With this, we decided that to implement all the required functionality of this program, it would be best to use multiple files that would be shared amongst each other: one file to hold our sorting algorithms, one file to create our needed arrays for testing of best/worst/average cases, and a main file that created our arrays, called our sorting algorithms, then timed each of our algorithms case by case. This not only made our code portion more readable, but also easier to edit and add to without scrolling through a single unorganized section of code. After the coding portion was completed successfully as expected, we stored the experimental data with graphs in a separate excel file that will be discussed further in our results section.

3. Theoretical Analysis

3.1 Best Case

The best case situations for comparison sorts were similar across many of the sorting algorithms, except for one, Quick Sort. For most of the algorithms, what produced a best-case scenario was when the algorithm tried to sort a presorted list. This can be expected, as with how most of the algorithms work, presorted lists will require the least amount of swaps or actions performed. The one exception was Quick Sort, and this is because its performance depends more on how the list is partitioned. To get the best case for Quick Sort, the list of elements would need to be partitioned at the median of the first, middle, and last index, nearest to the middle of the list of elements.

Heap Sort is somewhat of a special case. In virtually any situation, Heap Sort will have consistent runtime performance due to how it sorts the elements; when it is called, it will always have to traverse the height of the tree to either the minimum or maximum element to delete it from the heap and to insert it into the list of sorted elements. Therefore, on every single pass of the algorithm it will traverse the heap in log(n) time for n elements to produce the same time complexity in every situation.

For linear sorting algorithms (Counting Sort and Radix Sort), the best case for these algorithms would be having a small digit range. This is attributed to how these algorithms sort, and the dependent nature they have on the amount of digits in the maximum value in the set of elements being sorted. The smaller the digit ranges, the less space is used in Counting Sort, and the less amount of passes are needed to sort in Radix Sort.

3.2 Average Case

The parameters that produce an average case time complexity is hypothesized to be a randomized input for the majority of the algorithms. This is understandable because, as randomized elements are being sorted, some will already be correctly positioned and thus, little to no action needs to be taken by the algorithm. Additionally, when the elements are randomized, it represents the most practical use case for each of the algorithms. However, with Quick Sort, its performance is dependent, once again, on the partition; using random partitioning in this case would increase the chances of an average case runtime, with the best-case partition and worst-case partition possibilities becoming less likely.

3.3 Worst Case

The worst case for the comparison sorts hypothetically occurs when the list of elements is reverse sorted, requiring the most amount of actions and checks to be performed by the algorithm to sort the elements. With the linear sorting algorithms, since they do not compare the elements between one another to sort, the organization of the elements in the list is irrelevant. To produce the worst case runtime with Counting Sort and Radix Sort is to have a large range of digits. Counting Sort will require a lot of memory and calculations when the digit range is large, and Radix Sort will require many passes depending on how many digits are in the maximum element in the list of elements to be sorted.

4. Experiment

4.1 Setup

The device that was used is a Microsoft Surface Pro 7 with an Intel I5-1135G7 processor (2.4 GHz) with 8 GB of RAM (7.85 Usable). We limited background applications such as adblockers, antiviruses, and Google Chrome in the cases we were not using Online GDB. Limiting the background applications allowed the computer to focus primarily on our testing thus limiting the chance of inaccurate data. The IDEs we used were Codeblocks 20.03, Visual Studio 2022, and Online GDB. Our method of timing was the C++ library “Chrono”, and we measured time in milliseconds using the high-resolution clock. Online GDB was our main IDE as we were all familiar with coding and sharing projects through this coding environment. Testing was done locally on Codeblocks to limit or decrease the chance of having inaccurate data due to internet issues.

4.2 Data Sets

Our data consists of four different sets of elements (25000, 50000, 75000, 100000) to display the increase in runtime as the input size was increased. Depending on the sort, we had a data range from 0-100000 or from 0-100 (linear sorts). Depending on the algorithm, the arrays may be filled to elicit sorted, unsorted, and random arrays. However, in Quick Sort, we manipulated the data to have the first index as the lowest number to cause a worst-case scenario. Each algorithm and dataset was tested 5 times to ensure we would not have outliers that would skew our results to be totally inaccurate.

4.3 Results and Discussion

In the best-case scenario, the majority of sorting algorithms performed as expected with very low runtimes. However, Selection Sort remains a very significant outlier:

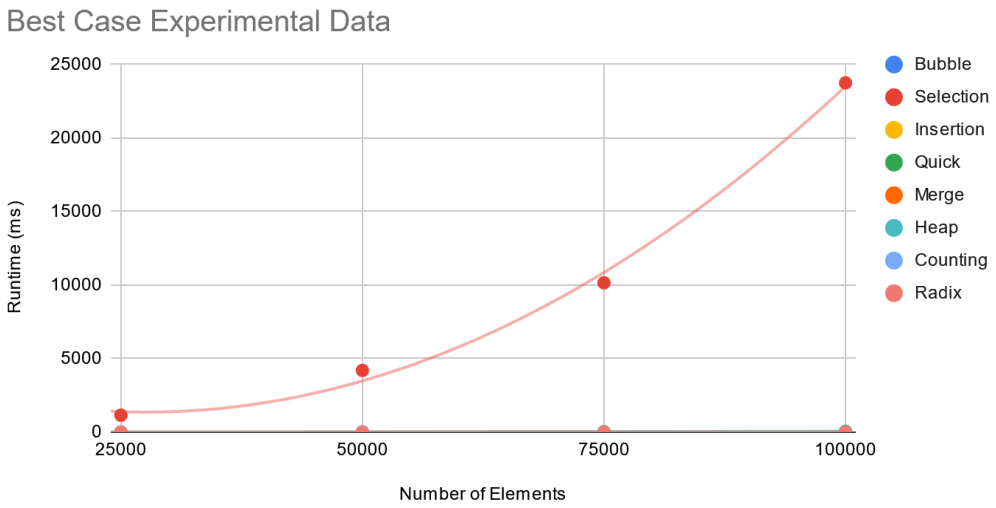

Figure 1: Best Case Data

Case	# Elements	Attempt	Bubble (ms)	Selection (ms)	Insertion (ms)	Quick (ms)	Merge (ms)	Heap (ms)	Counting (ms)	Radix (ms)
Best	25000	1	0	779	0	3	1	3	0	1
		2	0	748	0	3	2	5	0	0
		3	0	889	0	11	2	7	0	1
		4	0	1689	0	0	0	0	0	0
		5	0	1672	0	6	9	11	0	1
		Average	0	1155.4	0	4.6	2.8	5.2	0	0.6
	50000	1	0	4244	0	2	0	15	0	6
		2	0	4082	0	20	14	5	0	1
		3	0	4281	0	12	0	26	0	6
		4	0	4197	0	12	6	31	0	1
		5	0	4182	0	0	0	25	0	0
		Average	0	4197.2	0	9.2	4	20.4	0	2.8
	75000	1	0	12725	0	40	12	76	0	0
		2	0	9771	0	15	7	19	0	7
		3	0	9262	0	15	0	29	0	15
		4	0	9679	15	15	15	14	0	0
		5	14	9333	0	14	15	38	0	0
		Average	2.8	10154	3	19.8	9.8	35.2	0	4.4
	100000	1	0	30573	0	43	28	78	15	17
		2	0	26792	8	46	29	90	0	0
		3	0	26373	0	44	16	93	12	16
		4	0	16548	0	25	11	41	0	5
		5	0	18490	0	22	5	40	5	8
		Average	0	23755.2	1.6	36	17.8	68.4	6.4	9.2

Table 1: Best Case Data

Selection Sort's deviation may be attributed to the indiscriminate nature of its design where, regardless of the ordering of the input, the same number of swaps and comparisons occur. This is because Selection Sort involves swapping the largest value of the unsorted subarray with the end of the subarray; however, this can occur even if the last element is the largest element which is a redundant action.

Truncating **Figure 1** allows us to much more clearly see the performance disparity between the remaining sorts:

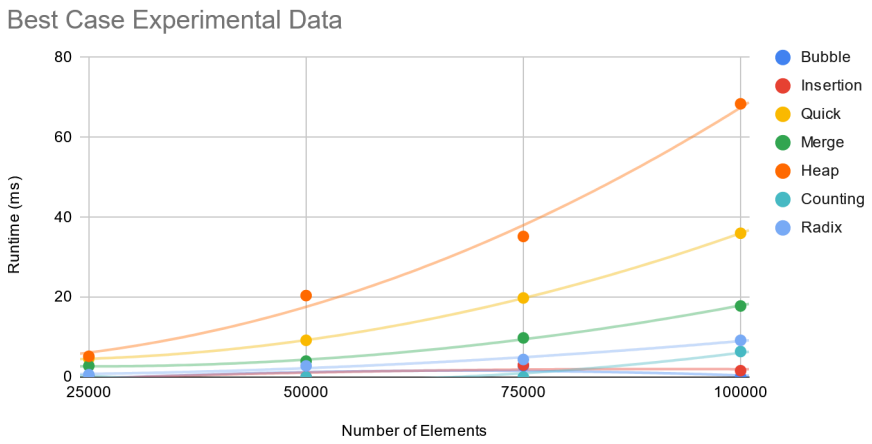

Figure 1.1: Best Case Data (Truncated)

Here, Insertion Sort stands out with a notable lead over the other sorts despite our typical associations with its $O(n^2)$ performance. Although, in the best case, Insertion sort will actually perform in O(n) time with minimal overhead, as it only needs to compare the elements without any swapping (effectively righting the redundancy issue present in Selection Sort). Additionally, Bubble Sort's surprising result may be attributed to our modification of the algorithm that causes early termination when no swaps are made through the initial comb through the array – thus, the elements were already in sorted order.

The average case results more readily align with our typical notions of how each sort should perform:

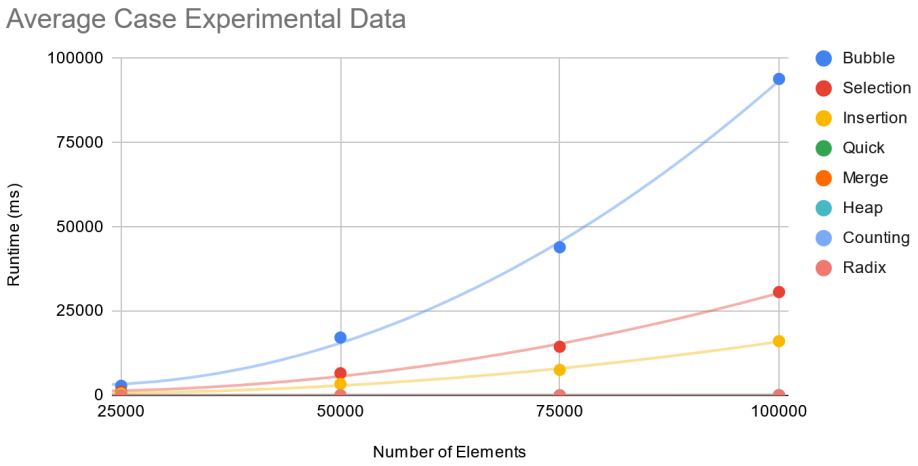

Figure 2: Average Case Data

Case	# Elements	Attempt	Bubble (ms)	Selection (ms)	Insertion (ms)	Quick (ms)	Merge (ms)	Heap (ms)	Counting (ms)	Radix (ms)
Average	25000	1	2027	752	473	3	3	5	1	2
		2	1963	735	357	3	2	3	0	2
		3	2964	1395	702	15	15	0	0	0
		4	2614	1041	515	16	0	0	0	0
		5	4675	1507	805	15	7	15	1	4
		Average	2848.6	1086	570.4	10.4	5.4	4.6	0.4	1.6
	50000	1	12257	3993	2123	0	7	10	0	0
		2	23397	10072	5399	23	25	42	0	11
		3	25643	10528	5312	2	24	37	0	13
		4	12068	4246	1973	15	15	14	0	0
		5	12541	4111	2138	15	10	15	0	15
		Average	17181.2	6590	3389	11	16.2	23.6	0	7.8
	75000	1	68571	22484	11840	27	29	65	11	12
		2	27332	9067	4838	16	13	25	1	0
		3	67660	22334	11598	32	25	62	14	15
		4	28829	9316	4750	17	15	15	0	8
		5	27457	8996	4748	15	17	31	0	0
		Average	43969.8	14439.4	7554.8	21.4	19.8	39.6	5.2	7
	100000	1	122108	40033	21322	43	47	88	12	20
		2	122667	39945	20736	40	56	86	11	13
		3	122002	40299	21100	32	44	90	0	23
		4	50547	16430	8727	15	29	47	0	2
		5	52340	16690	8626	16	26	41	0	11
		Average	93932.8	30679.4	16102.2	29.2	40.4	70.4	4.6	13.8

Table 2: Average Case Data

Of the $O(n^2)$ sorts, Bubble clearly performs the worst; this is likely due to a similar redundancy issue faced by Selection Sort as Bubble Sort combs through the entire array multiple times in order to do swaps on adjacent elements.

Similarly, truncating **Figure 2**, we again see results more aligned with known information:

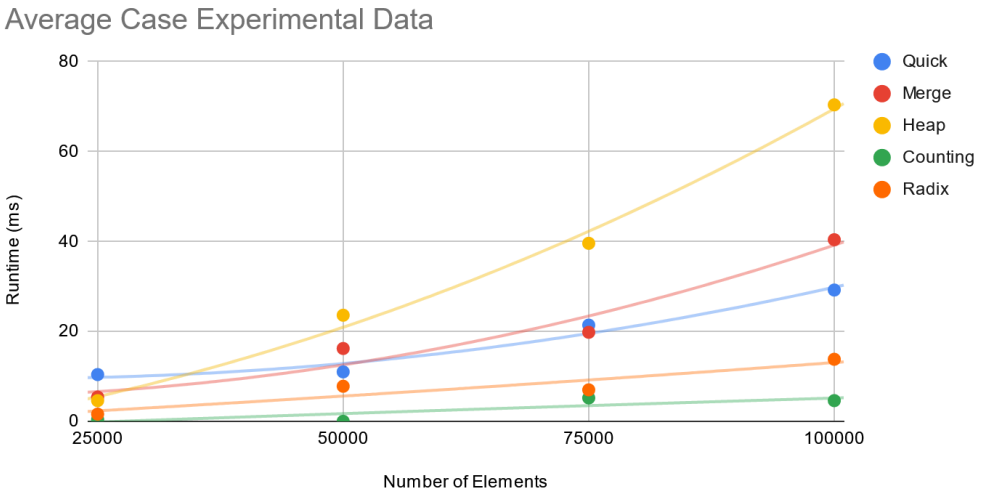

Figure 2.1: Average Case Data (Truncated)

We can see that, nevertheless, Heap sort performs notably worse than the other O(nlogn) sorts which are more tightly bound. This difference is likely due to the overhead created by building the heap prior to the actual sorting.

Finally, analyzing our worst-case experimental data yields surprising findings:

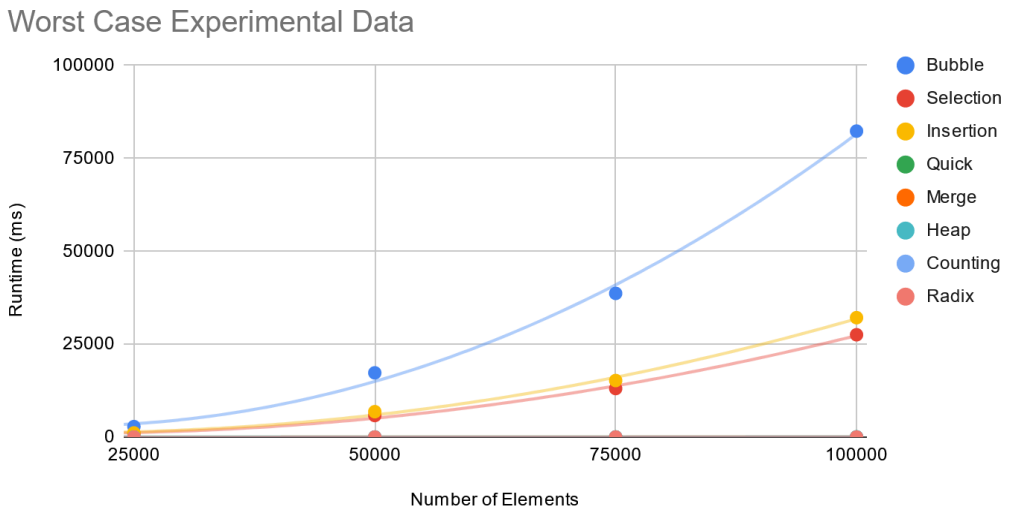

Figure 3: Worst Case Data

Case	# Elements	Attempt	Bubble (ms)	Selection (ms)	Insertion (ms)	Quick (ms)	Merge (ms)	Heap (ms)	Counting (ms)	Radix (ms)
Worst	25000	1	1878	659	803	2	0	5	1	1
		2	1756	628	744	3	2	3	1	3
		3	3715	1054	1258	15	0	15	0	15
		4	2641	1187	1340	0	0	0	10	0
		5	4110	1229	1478	15	3	15	15	0
		Average	2820	951.4	1124.6	7	1	7.6	5.4	3.8
	50000	1	10681	3585	4261	0	0	10	0	7
		2	26836	8905	10521	16	10	42	0	0
		3	27124	9173	10845	33	15	37	3	0
		4	10777	3711	4251	0	13	14	0	0
		5	11010	3633	4292	0	0	15	0	0
		Average	17285.6	5801.4	6834	9.8	7.6	23.6	0.6	1.4
	75000	1	60835	20329	23769	27	12	65	0	10
		2	24962	8582	9556	12	9	25	0	0
		3	59514	20126	23544	28	10	62	0	18
		4	24297	8232	9401	15	15	15	0	0
		5	23715	8051	9812	15	11	31	0	0
		Average	38664.6	13064	15216.4	19.4	11.4	39.6	0	5.6
	100000	1	111156	35734	42153	45	19	88	0	22
		2	106638	36375	41734	32	0	86	0	7
		3	109071	36284	42524	40	25	90	9	23
		4	42569	14729	17241	24	2	47	0	14
		5	42234	14560	17024	16	15	41	2	0
		Average	82333.6	27536.4	32135.2	31.4	12.2	70.4	2.2	13.2

Table 3: Worst Case Data

Comparing **Figure 3** with **Figure 2**, we can see that Bubble Sort performs better in our worst case versus our theoretical average case. Further analysis reveals that a reverse-order input may actually give Bubble Sort an advantage: because the larger elements appear first, they are moved all the way to their final position; the unsorted subset effectively shrinks with each pass as those values are never moved again (consequently saving time).

Truncating **Figure 3** again suggests that many of our worst-case scenarios actually facilitate better performance than our average-case scenarios:

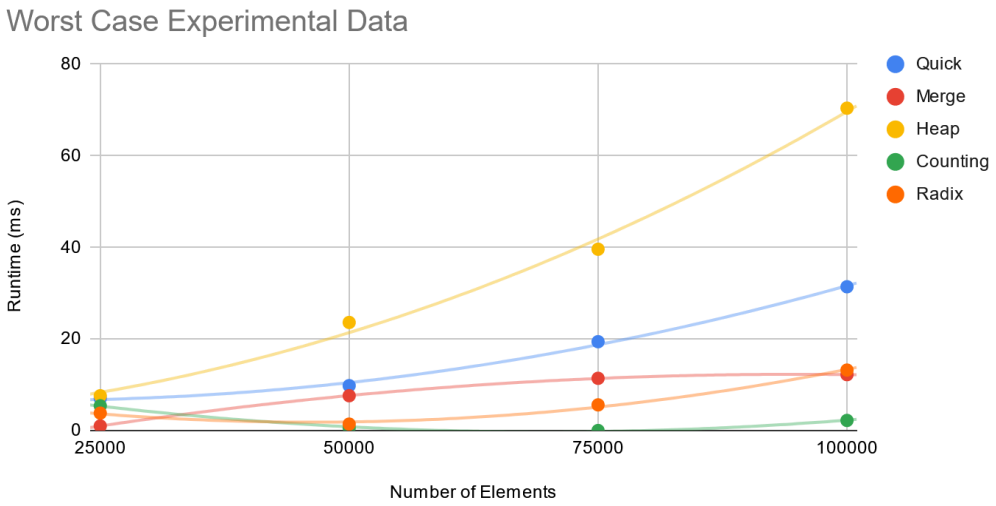

Figure 3.1: Worst Case Data (Truncated)

The most apparent difference between **Figure 2.1** and **Figure 3.1** is the Merge Sort result. It is not entirely clear why there was such a large difference in favor of the worst case (as Merge Sort should theoretically perform consistently regardless across same-sized inputs), but it is possible that this is simply a very severe case of run-to-run variance.

Q: Which of the five sorts seems to perform the best (consider the best version of Quick Sort)?

Counting Sort performed the best across all the different sorting algorithms as expected with its theoretical time complexity being linear. In the best case with the largest element set, it performed by far the best at 6.4ms, other than Bubble and Insertion Sort, which both had special cases that allowed them to perform better than their theoretical values. In the average case, it had the best numbers over all the other sorting algorithms at 4.6ms. This is more representative of the performance for the algorithms since in the average case each algorithm will have to do the same amount of operations and comparisons and do not have any special situations that allow an algorithm to perform better than its theoretical time complexity. In the worst case, Counting Sort performed the absolute best across all sorts at 2.2ms. Counting Sort performing so well, and so consistently across all cases is due to the non-comparison nature of the algorithm, thus, the order of elements in the array to be sorted is irrelevant.

Q: To what extent do the best, average, and worst-case analyses (from class/textbook) of each sort agree with the experimental results?

The graphs comparing different sorting methods show something pretty interesting: they match up well with what we learned in this class about how fast these methods are supposed to work. Simple sorts like Bubble and Selection can get really slow when they have a lot of items to sort, and the graphs that clearly show sharp upward curves for bigger data sets. This fits the idea that these methods take a lot more time when they are dealing with more things to sort.

Then there are the more advanced sorts like Quick, Merge, and Heap, which are known for being quicker on average. The experiments show that as you increase the number of items to sort, these methods remain consistent. This means they are better at handling big piles of data and confirms what we think about their performance. For Radix and Counting Sort, which do not compare items in the traditional way and are known for being fast no matter how the data is arranged, the graphs show a steady time to sort. This would make sense because these sorts are known for their consistent speed. So, overall, the real-world tests support what the theory says and give us a clear picture of how these sorting methods perform in practice.

However, in testing, there were a few interesting results from two algorithms to make note of that performed in unexpected ways that divert from what theoretical analysis would suggest. One of these algorithms is Bubble Sort which performed better in its worst case with a reversed list, than its average case with a randomized list. This would seem to contradict intuition, albeit, there is an explanation for this. When Bubble Sort is being used on a reverse-sorted list, it will have a unique advantage given how the program works. Since the largest elements will be at the front, they will immediately get bubbled all the way to their correct position in the list, and therefore never move again. This means the amount of elements that need to be swapped continuously decreases as every element will get put in its proper position on the first loop through the list, resulting in less swaps on average, than in a randomized list. The second algorithm of note was Merge Sort. It also had the same situation as Bubble Sort, which was better performance in the worst case than its average case. The issue with this finding was there is no real apparent reason for this result. From what we could conclude, the mismatch in performance is due to unknown causes.

Q: For the comparison sorts, is the number of comparisons really a good predictor of the execution time? In other words, is a comparison a good choice of basic operation for analyzing these algorithms?

The number of comparisons is a good predictor of the execution time since the runtime rises with the increase in comparisons. Given that some comparison sorting algorithms have a worst case of $O(n^2)$, showing that the number of comparisons would be exponential. Putting this complexity against a different comparison sorting algorithm like Merge Sort or Quick Sort with a possible complexity of O(nlogn), shows that the number of comparison with an algorithm with an $O(n^2)$ complexity would increase much faster than that of a complexity of O(nlogn).

5. Problem Solving and Analysis (Part 2)

5.1 Prompt

The problem-solving and analysis portion of our project involves the following prompt: *"Design and implement an efficient algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x."* This prompt was to be answered with two approaches: a brute-force algorithm that checked every possible pair of elements and an efficient algorithm that could avoid doing so.

5.2 Solutions

To solve the problem through a brute-force approach, the algorithm we designed makes use of nested for loops in a naive implementation to make comparisons of each integer in the set against every other one:

```
Alg BruteForce(set[], x)
1. n ← length(set)
2. for i ← 0 to (n - 1) do
3.   for j ← (i + 1) to (n - 1) do
4.      if set[i] + set[j] = x then
5.         return true
6.   return false
```

Figure 4: Brute Force Algorithm

The general idea of this algorithm is to test each integer S[i] in the set S of size n against all other integers S[j] where 0 <= i < (n - 1) and j > i. In each pass, we test if S[i] of S[j] sums to our target value; j begins from (i + 1) because there is no need to test an integer

against itself and because checks against previous values will have already been done with lesser values of i. If we find some pair S[i] + S[j] that satisfies the query during our iterations, then we return true; if all possible pairs of checks are exhausted without satisfying our query, then return false as there were no pairs found that sum to our target value.

We attempted to approach an efficient solution through a variety of methods, including dynamic programming methods such as tabulation. Despite this, through some discussion of our prototypes against the working principles of dynamic programming, we concluded that none of our approaches satisfied all of the conditions in which dynamic programming is a valid choice (namely, we could not determine how this problem can be represented as a series of overlapping sub-problems that could be combined into a complete solution).

However, the attempted use of tabulation inspired the use of a dictionary-type data structure that we could use to query specific elements in constant time; as a result, we then designed an algorithm that made use of a hash-table data structure in order to benefit from near-constant time access and manipulation:

```
Alg Efficient(set[], x)
1. hashTable[n]
2. for i = 0 to n do
3.    if set[i] <= x then
4.         InsertHashLinear(hashTable,set[i])
5. for i ← 0 to (n-1) do
6.    if set[i] <= x then
7.         temp ← x - set[i]
8.         if SearchHash(temp) then
9.              return true
10. return false
```

Figure 5: Efficient Algorithm

This algorithm operates as follows: it first initializes an empty hash table of size n and then fills it with all elements from the set S which are less than or equal to the target sum through linear collision resolution. This is done because it is not possible for any value larger than the target sum to sum to it; it is assumed that all input values are positive (including zero). Next, for each integer S[i] in the set (except for the last), we initialize a temporary variable that holds the target sum minus S[i] and then search the hash table for this value — by specifying only the exact value we need for each integer and using a dictionary-like data structure to search for it, we forgo the need to compare it against every other element in the set. If we find this value in the hash table, then it

returns true. If every element is tested without any valid pairing found, then it returns false.

5.3 Solution Analyses

Through line-by-line analysis, we can see the efficiency disparity between both algorithms:

Algorithm	Line-by-line Analysis
Alg BruteForce(set[], x) 1. n ← length(set) 2. for i ← 0 to (n - 1) do 3. for j ← (i + 1) to (n - 1) do 4. if set[i] + set[j] = x then 5. return true 6. return false	1. 1 2. n 3. $\sum_{j=1}^{n-1} j + 1 = \frac{n(n-2)}{2} + 1$ 4. $\frac{n(n-2)}{2}$ 5. 1 6. 1 $T(n) = c + cn + \frac{cn(n-2)}{2} + c + \frac{cn(n-2)}{2} + c + c$ $= c(n^2 + 2n + 4)$ $= O(n^2)$
Alg Efficient(set[], x) 1. hashTable[n] 2. for i = 0 to n do 3. if set[i] <= x then 4. InsertHashLinear(hashTable,set[i]) 5. for i ← 0 to (n-1) do 6. if set[i] <= x then 7. temp ← x - set[i] 8. if SearchHash(temp) then 9. return true 10. return false	1. 1 2. n + 1 3. n 4. O(n) 5. n 6. n - 1 7. n - 1 8. O(n) 9. 1 10. 1 $T(n) = c + c(n+1) + c + O(n) + c + c(n-1) + c(n-1) + O(n) + c + c$ $= c(3n + 4) + O(n)$ $= O(n)$

Table 4: Brute Force vs. Efficient Algorithm Runtime Analysis

As we can see from **Table 4**, our brute force implementation operates with a runtime of $O(n^2)$ whereas the efficient algorithm can operate in linear O(n) time. The brute force algorithm's efficiency disadvantage primarily stems from the use of nested for loops

(lines 2. and 3.) to reiterate through the array multiple times; this is due to the algorithm's rudimentary approach of finding a suitable pairing by testing each element in the set against every other. While the local subarray test-space does shrink, that does not necessarily reduce the runtime in a meaningful way as any lower-order efficiency gains we achieve become less and less apparent as the input scales.

On the other hand, our efficient algorithm's advantageous runtime stems from its use of linear-time hash functions while avoiding a similar try-all testing approach. In particular, both the hash insertion and search functions operate in linear-time on average but can operate in constant O(1) time should there be no collisions. Limiting the inputs to only values that are less than the sum theoretically reduces the total number of collisions by reducing the total number of entries overall.

6. Conclusion

From a theoretical standpoint, sorting algorithms like Bubble, Selection, and Insertion were expected to perform efficiently when dealing with particularly small data sets. Also, while dealing with data sets that are particularly unsorted or reversed they would be completely inefficient and useless. On the contrary, more advanced algorithms like Quick Sort, Merge Sort, and Heap Sort were projected to maintain consistent performance regardless of the data sets.

The outcomes of our tests largely aligned with our theoretical predictions. The advanced sorting algorithms like Quick, Merge, and Heap Sort demonstrated their abilities and efficiently handled most data sets, reflecting their superior design and operational logic; the simpler sorting algorithms like Bubble Sort, Selection Sort, and Insertion Sort found difficulties with larger and more disorganized data sets like reversed arrays.

As we saw, however, some of our findings did not line up with our predictions. In particular, some algorithms performed better in their theoretical worst case scenarios compared to their average. While this may have been due to run-to-run variance, further analysis revealed how our theories may have yielded unexpected advantages. This highlights how intuition is not necessarily the most accurate and may lead to undesirable results in real-world applications where the data set has known properties.

As for the problem-solving and analysis section, our rudimentary and optimized approaches to the problem proved both successful and insightful. While creating a brute-force algorithm proved straightforward, it was not difficult to surmise its shortcomings which motivated our discussion into an efficient algorithm. Designing an algorithm that could improve upon it, however, proved somewhat challenging as

prototyping a successful design required arduous trial-and-error, as well as, a degree of collaborative creativity. We will take this newfound understanding of the intricacies behind scratch algorithm design with us in our computer science careers moving forward.

With our findings in this project, we have gained valuable insight into the practical application of sorting analysis. This exemplifies the importance of selecting the right sorting algorithm based on the nature of the data set to get optimal results. The results of this project serve as an example to validate computational theories and real-world scenarios and enhance our understanding of algorithms as a whole.

7. Statement of Contributions

Each member of our group was required to present a sorting algorithm and their implementation of it. Each member of our group was assigned a section to work on at the bare minimum, but each member would still help with other sections. Every member was able to fulfill assigned tasks and communicated frequently regarding the project. We had two weeks during the project where communication was not as frequent due to unhealthy group members.

1. Jonathan - Collaborative Coding, Sorting Algorithms, Testing, Experimental Setup
    a. In charge of testing as this was the group members device we chose to run our programs. Discussed the experimental setup during the presentation since it was their device and their environment. Assisted with other sections such as the Part 2 Algorithm Analysis, Theoretical Analysis, and the Experimental Data Set Parameters.
2. Ian - Collaborative Coding, Sorting Algorithms, Theoretical Analysis, Comparison of Theory vs Actual
    a. Key organizer for in-person group collaboration. Discussed the theoretical analysis of each sorting algorithm by providing information about the best, average, and worst cases and certain scenarios that may invoke those cases. Analyzed and compared the theoretical findings with our actual findings after testing. Assisted with result analysis, Part 2's algorithm, and designing the experimental data set parameters.
3. Adrian - Collaborative Coding, Sorting Algorithms, Analysis of Results, Discussed Results, Part 2 Algorithm Analysis
    a. In charge of generating figures and analyzing the results gathered from testing. Discussed disparities and factors that may have skewed our results to seem inaccurate. In charge of the Part 2 Algorithm Analysis as this

member felt the most comfortable with doing line-by-line analysis of the algorithm.

4. Ahmad - Initial Organization, Collaborative Coding, Sorting Algorithms, Intro/Conclusion
    a. Responsible for creating the initial roadmap for completing the project. Devised the testing methodology for considering which member's implementations of each algorithm should be chosen for the final code, as well as, establishing baseline requirements that would evolve into the overall experimental parameter setup.
5. Zeke - Collaborative Coding, Sorting Algorithms, Experimental Data Set Parameters and Reasoning
    a. Key contributor to the troubleshooting stage of development. Diagnosed stack overflow issue with initial code prototype; enabled the use of large data sets for testing through dynamic memory allocation. Assisted with designing the experimental data set parameters as well as the initial program design for Part 2 using dynamic programming.

Overall, this group demonstrated excellent synergy with each other in a way that made the research and experimentation process enjoyable while still maximizing productivity. Each individual member demonstrated strong initiative; there were few, if any, points during our research where there was major stagnation in progress as each member delegated themselves to any open tasks that they could. Constant communication played a key factor in the group's efficiency as well — check-ins and collaboration occurred on a daily basis with plans for in-person collaboration established well in advance.

8. References

Cormen, Thomas H., and Charles E. Leiserson. *Introduction to Algorithms, 3rd Edition*. 2009.