Algorithm Analysis

Thuan Quach, Pratham Aggarwal, Jacob Masangcay, Amandeep Singh

Abstract—This paper cover the analysis of eight sorting algorithms: Insertion, Selection, Bubble, Merge, Quick, Heap, Counting, and Radix sort. This paper also covers a brute-force and a dynamic programming approach to finding pairs in a set that add up to a target value.

Index Terms—N/A

1 Introduction

The purpose of this project is to analyze the actual performance of different sorting algorithms to their theoretical time complexities. The sorting algorithms experimented on are the following: Insertion, Selection, Bubble, Merge, Quick, Heap, Counting, and Radix Sort. We will also cover a brute-force and a dynamic programming approach to solving a problem related to finding pairs (Section 3.1).

1.1 Contributions

Thuan's contributions were primarily focused on code development and presentation. He worked on part 1's code and output, developed the experimental setup and information sets, refined the theoretical sections of the report, handled the theory part about whether comparison is a good choice of basic operation for analyzing these algorithms, and wrote code for part 2's brute force technique. Pratham's contributions focused primarily with visual representation and theoretical analysis. He designed all of the project's graphs, helped write the sections of report, handled the experimental vs. theoretical parts of the report, and the conclusion of the report. He additionally provided code for the efficient algorithm in part 2 of the project. Amandeep mostly helped in the project's report writing. He helped calculate average values for half of the sorting methods and gave theoretical analysis for those algorithms as well. He also helped with the analysis of efficient algorithm for the project's second phase. Jacob's contributions were primarily on report writing. He helped in calculating the average values for the other half of the sorting algorithms and provided theoretical analysis for those algorithms in part 1 and the brute force algorithm in part 2. Working together had both positive and negative aspects. Our ability to divide the project into manageable tasks and assign them based on individual strengths led to efficient task completion. The exchange of different points of view encouraged creative solutions and improved our understanding of the project. However, one negative aspect was the schedule problem, everyone was not available at the same time which created a problem in getting everyone's opinion at the same time, and some people ended up contributing more than others. Everyone gave the best they could to the project.

Manuscript received December 4, 2023

2 Part 1: Comparison of Sorting Algorithms

2.1 Theoretical Analysis

2.1.1 Insertion Sort

For Insertion Sort, the best case O(n) occurs when there is only one comparison per element which can occur when the input array is already sorted (ascending order). The worst case O(n^2) occurs when every element must be compared to the previous elements which can occur when input array is reversed (descending order). The average case O(n^2) occurs when every element must be compared to roughly half of the previous element which can occur when the array is roughly half sorted.

2.1.2 Selection Sort

For Selection Sort, it is O(n^2) in all cases. This is because there is always the same number of comparisons for every iteration. However, the number of swaps can differ depending on how sorted the input is. In the best case where the unsorted element of the iteration is the minimum, there are no swaps, and this occurs when the input array is sorted (ascending order). In the average case where there are roughly half the number of swaps compared to comparisons, it occurs when the array is roughly half sorted. In the worst case where there are swaps for every comparison, it occurs when the input array is reversed (descending order).

2.1.3 Bubble Sort

For Bubble Sort, the time complexity varies across different scenarios. In the best case, which occurs when the array is already fully sorted, Bubble Sort exhibits a linear time complexity of O(n). This is because it only requires a single pass to check if any elements in the array need to be swapped, and the sorting process concludes if no swaps are necessary. The average case time complexity, O(n), is encountered when the array is approximately half sorted. In this situation, roughly half of the number of passes are needed to arrange the array properly. The worst-case time complexity, O(n^2), occurs when the array is in reverse order. In this scenario, Bubble Sort faces the maximum challenge of sorting the array, accumulating comparisons equal to the number of the elements in the array and incrementally increasing until it completes the sorting process.

2.1.4 Merge Sort

For Merge Sort, it is O(nlogn) in all cases. Merge sort consistently performs well, regardless of the initial order of the input data. However, the merging process can be less complex if the array is already sorted. This is because if the array is unsorted, the array would require rearrangement. Therefore, the best case would occur when the array is already fully sorted. The average case would occur when the array is roughly half sorted. The worst case would occur when the array is reversed.

2.1.5 Quick Sort

In the worst-case scenario, Quick Sort exhibits a time complexity of O(n^2), while in both the average and best cases, it achieves O(n log n). In the best-case situation, Quick Sort strategically divides the array into well-balanced segments during each recursive call. This efficient data splitting results in a balanced tree structure, leading to a time complexity of O(n log n). Conversely, the worst-case scenario arises when repeated use of a poorly chosen pivot leads to unbalanced partitions, such as in the case of a reversed array where the first index is chosen as the pivot. The average case occurs when the partitions are somewhat sorted balanced; this can occur with a randomized array.

2.1.6 Heap Sort

Heapsort's efficiency arises from the use of a binary heap, a specialized tree-based data structure. The heapify operation, performed during each extraction of the maximum element, takes O(nlogn) time. The worst, best, and average case can be determined based on how high an element has to traverse during the heapify process. The worst case occurs when the array is reversed; the best case occurs when the array is sorted; and the average case occurs when the array is half sorted or randomized.

2.1.7 Counting Sort

Counting Sort linear time complexity is based on counting occurrences in the input array rather than element comparisons. It generates a second array for frequency representation. Therefore, Counting Sort has a time complexity of O(n+r) in all cases where r represents the range. It demonstrates the best case when the input array has a low range. The worst case occurs when the range is extremely high. The average case typically has a range that is less than or equal to the number of elements.

2.1.8 Radix Sort

For Radix Sort, it is O(d(n+r)) in all cases. For Radix sort, the best case would occur when the array elements have a low number of digits within them and it would start with the least significant digit and work its way up to the more significant numbers in the array. The average case is would be when the highest number of digits is constant and less than the size of the array. As for the worst case, it would occur when the array has elements with a high number of digits such as equaling the size of the array.

2.2 Setup

The experiments utilize a program that asks the user for an input size and a sorting algorithm, and then displays the execution time in console. The timing mechanism that calculates the execution time for all three cases (worst, average, best) is implemented in the following way: the time is taken before the algorithm call using "clock()" from the ctime library, and stored in the variable called "time_taken" and then the time is taken again after the algorithm call and subtracted from the "time_taken" variable. The resulting value is converted to seconds and displayed in console. The experiments were performed on a Windows 11 Machine with a Ryzen 3 5300G CPU and 20 GB of DDR4 RAM in Visual Studios 2022. For every sorting algorithm, the experiment was repeated five times.

2.3 Data-Sets

There were seven types of data sets generated: randomized arrays, fully unsorted arrays, fully sorted arrays, half sorted arrays, arrays with a high range, arrays with a low range, and arrays with high number of digits. These data sets were chosen because they exhibit at least one of the case scenarios (worst, average, best) for each of the sorting algorithms, which is determined theoretically in Section 4. Each algorithm test began with an input of 1000 and then increased in the following order: 10000, 50000, 100000, and 200000. For the worst case scenarios, the fully unsorted arrays were used for Insertion, Selection, Bubble, Merge, Quick and Heap Sort; the array with a high range was used for Counting Sort; and the array with high digits was used for Radix Sort. For the average case scenarios, the half sorted arrays were used for Insertion, Selection, Bubble, Merge and Heap Sort; and the randomized arrays were used for Quick, Counting, and Radix Sort. For the best case scenarios, the fully sorted arrays were used for Insertion, Selection, Bubble, Merge, Quick and Heap Sort; and the arrays with low digits were used for Counting and Radix Sort.

2.4 Results and Discussion

2.4.1 Five best, average, and worst case Sorts

Figure 1 represents the five best case running time algorithms. These algorithms are Insertion Sort, Merge Sort, Quick Sort, Radix Sort, and Counting Sort. Figure 2 represents the the five average case running time algoorithms. These algorithms are Heap sort, Merge sort, Quick sort, Radix sort, and Counting sort. Figure 3 represents the five worst case running time algorithms. These algorithms are Bubble sort, Insertion sort, Selection sort, Quicksort, and Counting sort.

2.4.2 Experimental vs. Theoretical

Figures 4, 5, and 6 show the relationship between the experimental results and the theoretical time complexity for the eight sorting algorithms. The worst case graph (Figure 1) shows that the experimental results of each sorting algorithm match their expected time complexity. However, it is important to know that for an input size of 1000 for Heap sort, Merge sort, and Radix sort, there appears to be a deviation from the constant steady line that follows

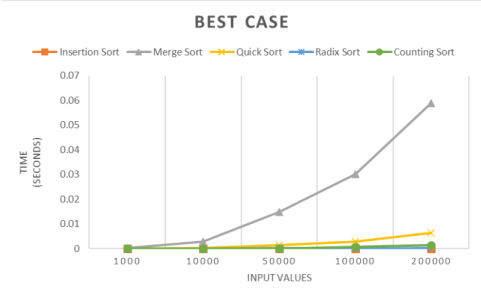

Fig. 1: Five Best-Case Running Times

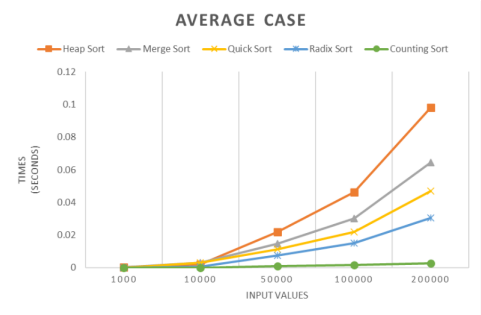

Fig. 2: Five Average-Case Running Times

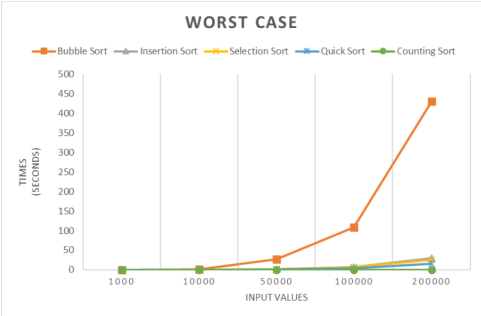

Fig. 3: Five Worst-Case Running Times

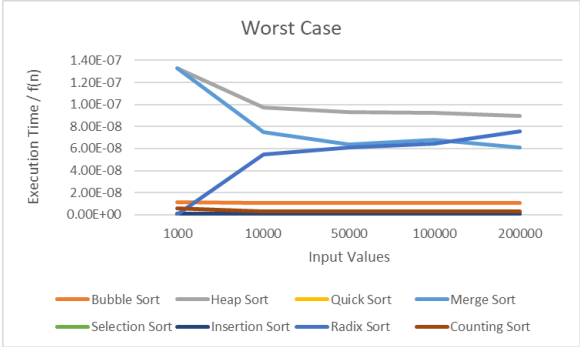

Fig. 4: Experimental vs. Theoretical: Worst Case

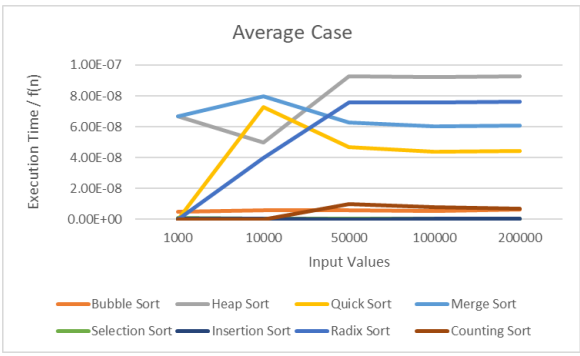

Fig. 5: Experimental vs. Theoretical: Average Case

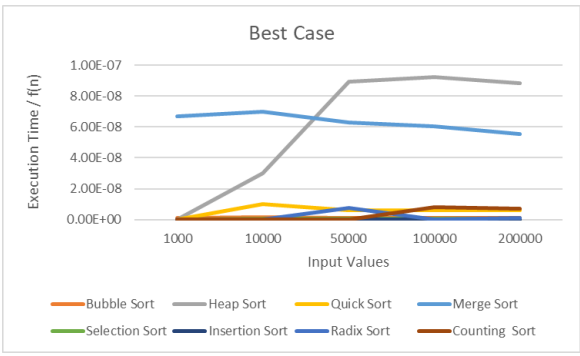

Fig. 6: Experimental vs. Theoretical: Best Case

the rest of the inputs. We believe the reasoning for this is that since the algorithms perform extremely fast for small input sizes, most of the execution time is likely dominated by background processes unrelated to the algorithm. As the input size grows, the algorithm's execution time dominates other background processes that are occurring at the same time, and thus, that is why there is not much deviation from the line for large input sizes. The average case and best case graphs also follow this trend. In the average case graph (Figure 2), once the input size is 50,000, then the line for every algorithm begins to stabilize and remain constant and steady. However, before 50,000 elements, there is some deviation in the graph. In the best case, it appears that the deviation occurring is for Heap Sort for an input size of 1000 and 10000. We believe that this might be to due to the precision of the timer. Since we got 0 seconds for the execution time, it caused the line to start at the bottom.

2.4.3 *Comparisons and Execution Time*

Comparisons are a great predictor of execution time for comparison algorithms. Figure 7 represents the correlation between the number of comparisons and execution time in the worst case for every algorithm. This graph is plotted as (execution time)/(number of comparisons) vs. input size. We defined the number of comparisons for Bubble sort and Selection sort as $(n^2)/2$, Quick Sort and Selection as (n(n-1))/2, Heap Sort as 2nlgn, and Merge Sort as n/2. Figure 8 represents the average case and Figure 9 represents the best case. Since every algorithm has a almost constant line for every input size, it can be concluded that comparisons have a great impact on the execution time, and thus are a great way to predict the execution time. However, it is

important to note that there is some deviation for smaller input sizes, and we believe that has to with background processes dominating the execution time for those input sizes.

Fig. 7: Comparisons and Execution Time: Worst Case

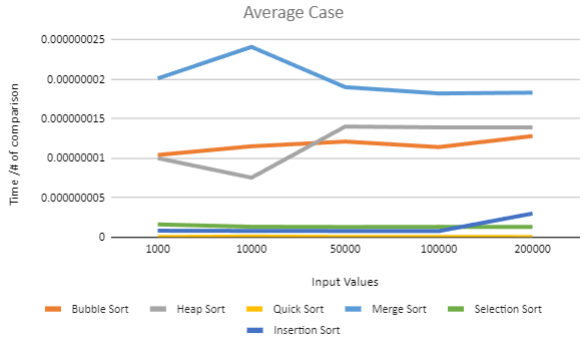

Fig. 8: Comparisons and Execution Time: Average Case

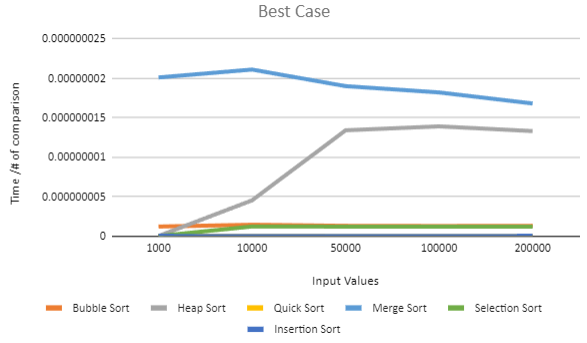

Fig. 9: Comparisons and Execution Time: Best Case

3 Part 2: Problem Solving and Analysis

In this part, we will cover an inefficient algorithm (brute-force approach) and an efficient algorithm (dynamic programming) that solves the following problem:

Given a set S of n integers and another integer x, determine if there exists two elements in S whose sum is exactly x.

3.1 Brute Force Approach

The running time of the brute force approach (Algorithm 1) is O(n^2). This can be determined by analyzing the costs at every line of the algorithm. Line 3 takes n+1 steps. Line 4 takes $\sum_{i=1}^{n} n - i$ steps. Line 5 and 6 takes $\sum_{i=1}^{n} n - i - 1$ steps. Lines 7, 8, and 9 have no costs since they just represent the structure of the for loop. Line 10 takes 1 step. It makes sense that the running time is O(n^2) because the algorithm implements a double nested for loop.

Calculation of the running time:
T(n)
= n + 1 + $\sum_{i=1}^{n} n - i$ + $\sum_{i=1}^{n} n - i - 1$ + $\sum_{i=1}^{n} n - i - 1 + 1$
= n + 3($\sum_{i=1}^{n} n - i$) = n + 3($\sum_{i=1}^{n} n$ -($\sum_{i=1}^{n} i$)
= n + 3(n^2 - n)
= O(n^2)

Algorithm 1: Brute Force Approach Pairs

```
1  Input: S, n, x
2  Output: True or False
3  for i = 1 to n do
4      for j = i+1 to n do
5          if S[i] + S[j] = x then
6              return True
7          end
8      end
9  end
10 return False
```

3.2 Dynamic Programming Approach

The running time of the dynamic programming approach is O(n). This can be determined by analyzing the costs at every line of the algorithm. Line 3 takes 1 step. Line 4 takes n+1 steps. Line 5 and 6 take n steps. Line 7 takes 1 step. Line 9 takes n steps. Line 11 takes 1 step. It makes sense that the running time is O(n) because there is just a for loop that iterates through the input set.
Calculation of the running time:
T(n)
= 1 + n + 1 + 1 + n + 1 + 1 + 1 = 2n + 6 = O(n)

Algorithm 2: Dynamic Approach Pairs

```
1  Input: S, n, x
2  Output: True or False
3  seen = create_empty_set()
4  for i = 1 to n do
5      complement = x - S[i]
6      if complement in seen then
7          return True
8      end
9      add S[i] to seen
10 end
11 return False
```

4 CONCLUSION

In this report, we analyzed the performance and complexity of eight sorting algorithms on various data sets. We measured the number of comparisons and swaps and created graphs that demonstrated the relationship between input size and execution time. We also compared the experimental findings to the theoretical time complexity of each algorithm. Our results showed that the initial order and distribution of the input data affected the performance and complexity of the sorting algorithms. In terms of time efficiency and scalability, some algorithms, such as merge sort, quick sort, heap sort, counting sort, and radix sort, outperformed others, such as insertion sort, selection sort, and bubble sort. However, in certain cases, certain algorithms outperformed others. For significantly sorted data, for example, insertion sort was the fastest method, whereas counting sort and radix sort were the most efficient algorithms for data with a limited range of values. The second section looked at two techniques to addressing a given problem: a brute-force approach with an O(n^2) time complexity and a more efficient dynamic programming approach with an O(n) time complexity. Each algorithm's execution time was looked at, highlighting the efficiency gains of the dynamic programming approach. Our analysis proved the significance of selecting an appropriate sorting method for various types of data and difficulties. It also demonstrated the trade-offs between the algorithms' time and space complexity, stability, and adaptability.