Term Project Report

Mingkuan Pang, Harry Atulbhai Patel, Harshitha Vallabhaneni, Quan Hoang Nguyen Rashmitha Garige
Matin Pirouz

Abstract—In this report, we compared and analyzed the theoretical time complexity and experimental time complexity of eight sorting algorithms. Including insertion sort, selection sort, bubble sort, merge sort, quicksort, heapsort, counting sort, and radix sort. Time complexities in this report include the best-case complexity, average-case complexity, and worst-case complexity for each of the sortingalgorithms mentioned above. In the problem solving and analysis section, we solve the problem of finding whether, given a set $S$of n integers, there exists two elements in *S* whose sum is exactly *x*. This problem is solved first in a brute force method where every possible pair of numbers is checked indiscriminately. Our goal is to solve the problem using a more efficient algorithm. To do this, we will be creating an array to store every integer we went through, then every time when we meet an integer we search if there exists another integer, we went through to make their sum to be exactly x.

1 INTRODUCTION

1.1 Part 1: Comparison of Sorting Algorithms

The theoretical time complexity and experimental time complexity of sorting algorithms (insertion sort, selection sort, bubble sort, merge sort, quicksort, heap sort, counting sort and radix sort) are compared. The theoretical time complexity (best, average, worst case) are recorded. Then, to get the time complexity of the experiment, different input array types and data sizes are passed to the sorting algorithm, which runs within the set time. The results then graphically show the temporal complexity of the experiment.

1.2 Part 2: Problem Solving and Analysis

The problem that we solved was finding whether, given a set *S* of *n* of integers, if there exist two elements in *S* whose sum is integer *x*. To solve this problem using the brute force method, eachvalue is compared to the other using a nested loop. One solution is to create an array A to store every integer that went through, and check whether x – i exists in the A, for the next integer i in the S, which will decrease the time complexity from $O(n^2)\ to\ O(n)$.

1.3 Contributions

- compared the theoretical time complexities of multiple sorting algorithms to their experimental performance
- determined what the best, average, and worst-case inputs are for all algorithms tested
- implemented an efficient solution to solve whether there exist two elements whose sum is x in set S

2 COMPARISON OF SORTING ALGORITHMS

2.1 Theoretical Time Complexities

Insertion sort, the best-case time complexity is $O(n)$, it will be the input is already in order, therefore, for each element inserted, only the previous element needs to be examined.

For the general case, assuming that the input array is in the randomly order, the time complexity is the same as the worst case, which is

$$T(n) = O(n^2)$$

The worst case will be the input is totally reversed. In that case, the first element being considered when the second element is inserted, and the first two elements being considered when the third element is inserted…Then the time complexity on the worst case will be

$$T(n) = 1 + 2 + 3 + \cdots n - 1 = \frac{n^2}{2} = O(n^2).$$

Selection sort, because selection sort divides the array into two parts, one is ordered and the other is unordered, the algorithm selects elements from the unordered array and inserts them into the ordered array, so its time complexity is

$$T(n) = 1 + 2 + 3 + \cdots n - 1 = \frac{n^2}{2} = O(n^2).$$

For the three cases of the input, the time complexity is the same for the selection sort.

Bubble sort, the best-case time complexity is $O(n)$, achievable with sorted input. One pass through the input size n, since is already sorted, no further traversal is needed.

The average-case and worst-case time complexity is $O(n^2)$, achievable with randomly sorted input and reverse sorted, in those two cases each element will compare with the restof the input. Therefore, for those two cases, the time complexity will be

$$T(n) = 1 + 2 + 3 + \cdots n - 1 = \frac{n^2}{2} = O(n^2).$$

Merge sort, for merge sort, the time complexity will be

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(nlogn)$$

where $2T\left(\frac{n}{2}\right)$ is the algorithm divides the array into two parts, and $O(n)$ is the time complexity for the algorithm to conquer the divided parts. It will not be affected by the order of the input array. Therefore, the time complexity for those three cases will be the same.

Quicksort, because quick sort is a recursion algorithm, therefore, the time performance of quicksort depends on the depth of the quicksort recursion. Best Case, on the best case, partition evenly divides the array into two parts every time. Therefore, if the size of input is n, then the time complexity will be

$$T(n) = n * (T(1) + logn) = O(nlogn)$$

Average Case, on the average case, partition divides the array from somewhere in the middle, therefore, the time complexity will be the same as on the best case. Worst Case, on the worst case, the input is sorted or reversed, one of the divided parts is empty. on this case the time complexity will be

$$T(n) = 1 + 2 + 3 + \cdots n - 1 = \frac{N^2}{2} = O(n^2).$$

Heap sort is the continuous construction of the heap, after finish constructed the first heap, swapping head and tail, then adjust the heap to make it to be a max heap or min heap, then removing the head(tail) and reconstructing the heap with the rest part. The time complexity is

$$T(n) = n + n * logn = O(nlogn)$$

$O(n)$ is the time complexity for the heap construction. $O(nlogn)$ is the time complexity for adjusting the heap. No matter how the order of the input is, the time complexity will stay the same.

Counting sort, the time complexity for counting sort is $T(n) = O(n + k)$ where the k is the range of the input. For counting sort, no matter how the order of the input is, the time complexity will stay the same size.

Radix sort, the time complexity for counting sort is $T(n) = O(d(n + k))$ where the d is the number of passes and k is the radix used (In this project, it is 10). For radix sort, no matter how the order of the input is, the time complexity will stay the same.

3 EXPERIMENT

3.1 Experimental Setup and Data Generation

The execution time of the sorting algorithms (insertion sort, selection sort, bubble sort, merge sort, quicksort, heapsort, counting sort, and radix sort) are run on a desktop computer with AMD Ryzen 7 5800H CPU which is 3.25G Hz, and the ram on the machine is 16 GB. The code is written in C++ and the Time.h library is used to time the sorting algorithms. The code is compiled and run using Microsoft Visual Studio. An array of the same value, sorted array, random array, and reverse sorted array is chosen as input types. These three array types account for the best-case, average-case, and worst-case inputs for the mentioned sorting algorithms. For data size, n = 10000, 20000, 30000 is chosen for the Insertion Sort, Selection Sort and the Bubble Sort. n = 30000, 60000, 90000 is chosen for Merge Sort, Quick Sort, Heap Sort, Counting Sort and the Radix Sort. For each input type and data size, the sorting algorithms are run 5 times and the average is recorded in milliseconds.

3.2 Experimental Data

The experimental results show that, as can be seen from Fig.1-Fig.3, radix sort, counting sort, merge sort, heap sort and quicksort have the best performance among all the sorting algorithms tested. Insertion sort, selection sort, and bubble sort performed worst considering their worst-case and mean time complexity of $O(n^2)$. Although the optimal time complexity of insertion sort in Figure1 is $O(n)$; In Figures 2 and 3, the overall performance of insert sort is worst than radix sort, counting sort, merge sort, heap sort, and quicksort using median pivot.

3.3 Experimental Data Analysis

Insertion sort, the experimental diagram in Fig. 4 conforms to the asymptotic analysis, and the experimental and theoretical time complexity in the best case is $O(n)$. As both the experimental and theoretical time complexity are, the average case and the worst case in Fig.5 and Fig.6 conform to the asymptotic analysis $O(n^2)$. The experimental diagrams in Fig.5 and Fig.6 show that the time increases significantly with the increase of data volume.

Selection sort, in Fig.4 - Fig.6. which conforms with the asymptotic analysis of $O(n^2)$ for best, worst, and average-case time complexities.

Bubble sort, the experimental diagram in Fig.4 does not match the asymptotic analysis $O(n)$ of time complexity in the best case. The experimental graph shows $O(n^2)$. The reason is in my implementation, there are two nested for loops. Whether or not the input array is sorted, each element must compare with the rest of input. Therefore, the time complexity is $O(n^2)$ across three cases.

Bubble Sort

```
Bubble-Sort (A)
for i = n to 1 do
    key = i
    for j =0 to i do
        if A[j] > A[key] then
            key = j
swap A[i],  A[key]
```

Merge sort, the experimental graphs in Fig4-Fig6 match with the asymptotic analysis as the best, average, and worst-case time complexity is significantly greater than $O(nlogn)$ Across best, average, worst case experimental graphs. The reason is there are lots of array access operations in the merge function. The reason is because it is running on the Microsoft Studio, In Microsoft Visual Studio, these algorithms with a lot of array access typically have more experimental time than their asymptotic analysis. From Fig.7, Merge Sort performances significantly differently when run on Microsoft Visual Studio and VS Code. Obviously, merge sort performances much better than it does on Microsoft Visual Studio. However, since all the previous testing was done in Microsoft Visual Studio, I decided to continue the rest of the experiments on the Microsoft Visual Studio.

Merge

```
Merge (A, l, m, r)
if A.len > 1 then
    let lv be a new array
    let rv be a new array
    len1 = m - l
    len2 = r - m
    for i = 0 to len1
        lv.push_back(A[i+1])
    for i = 0 to len2
        rv.push_back(A[m+1])
    lv.push_back(INT_MAX)
    rv.push_back(INT_MAX)
    j = 0, k =0, pos = l
    for i = l to r
        if lv[j] <= rv[k] then
            A[i] = lv[j]
            j = j + 1
        else
            A[i] = rv[k]
            k = k + 1
```

Quicksort, median pivot whose consider to be best version of quick sort seems to have conform the asymptotic analysis of $O(nlogn)$ across three cases. Because median Pivot greatly reduces the probability that another array will be empty in quicksort

Heap sort the best, average and worst cases of experimental results in Fig.4-Fig.6 all conform to $O(nlogn)$ asymptotic analysis. Heap sort, regardless of input type, builds the heap and maintains the heap $n - 1$ times after each swap. The results show that the experimental results of the three experimental graphs agree with the asymptotic analysis.

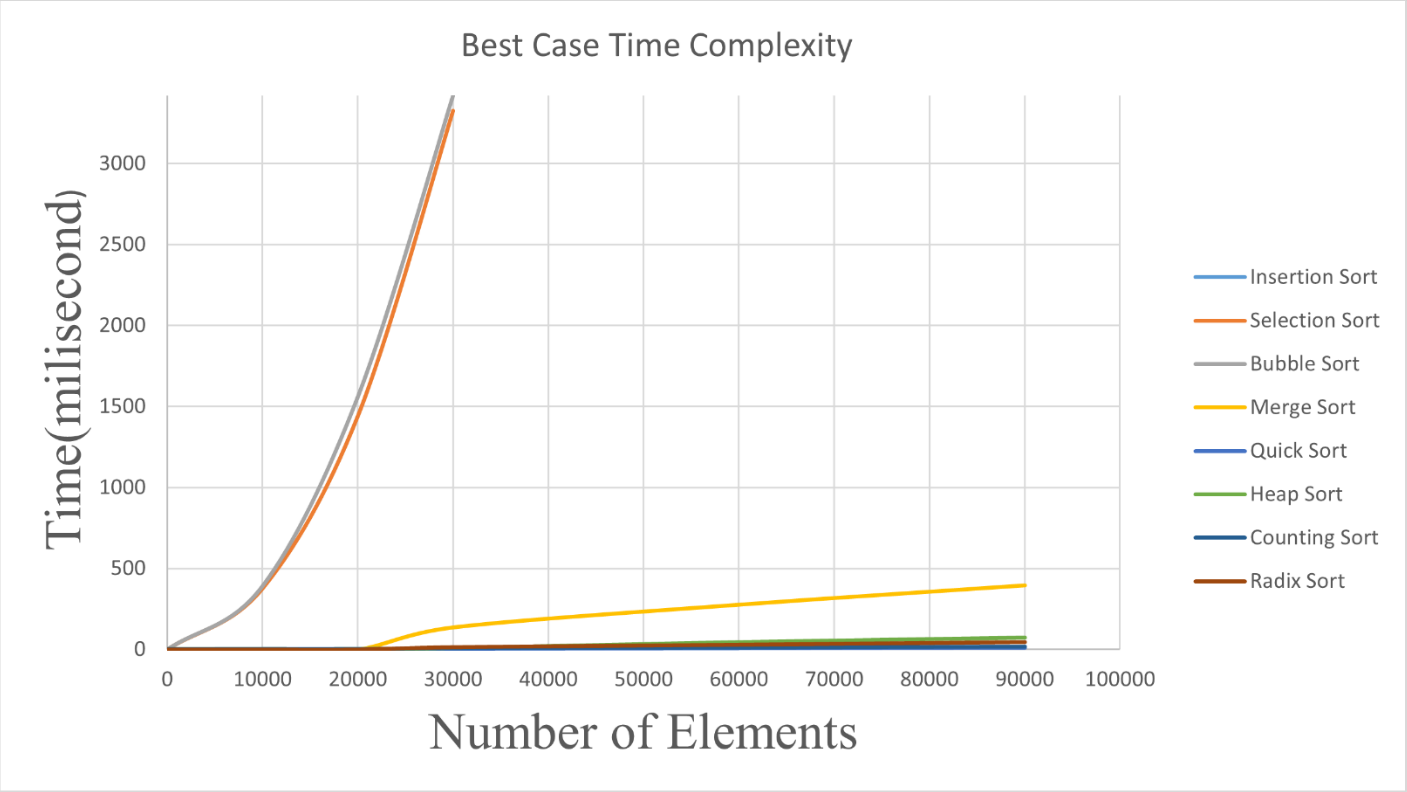
Counting sort, the experimental results in Fig.4 - Fig.6 for best, average, and worst-case time complexity conform with the asymptotic analysis of $O\,(n + r)$; where n is the input size and r is the range of the input. The experimental results showed linear run time, which corresponds with the asymptotic analysis $O(n)$ or $O(r)$ depending on whether the size of the input is greater, or the range of the input is greater.

Radix sort, Fig. 4 asymptotic analysis of best-case experimental results $O(d * (n + k))$. The number of digits is equal for all elements of an input, so no additional counting sort needs to be performed to accommodate elements with additional digits. The average experimental results in Fig. 5 conform with the asymptotic analysis of $O\ (d\ * (n\ +\ k))$. The number of digits for all elements doesn't differ much. However, it takes longer than the best-case in Fig. 4 because additional counting sorts need to be performed to accommodate more numerically large elements. The worst-case experiment does not accord with $O\ (d * (n\ +\ k))$ asymptotic analysis. The error could be that, since the best-case input is randomly sorted, some of the running numbers may have changed considerably due to the randomly generated input, resulting in the graph shown in Fig 4.

3.4 Is the Number of Comparisons a Good Predictor ofExecution Time?

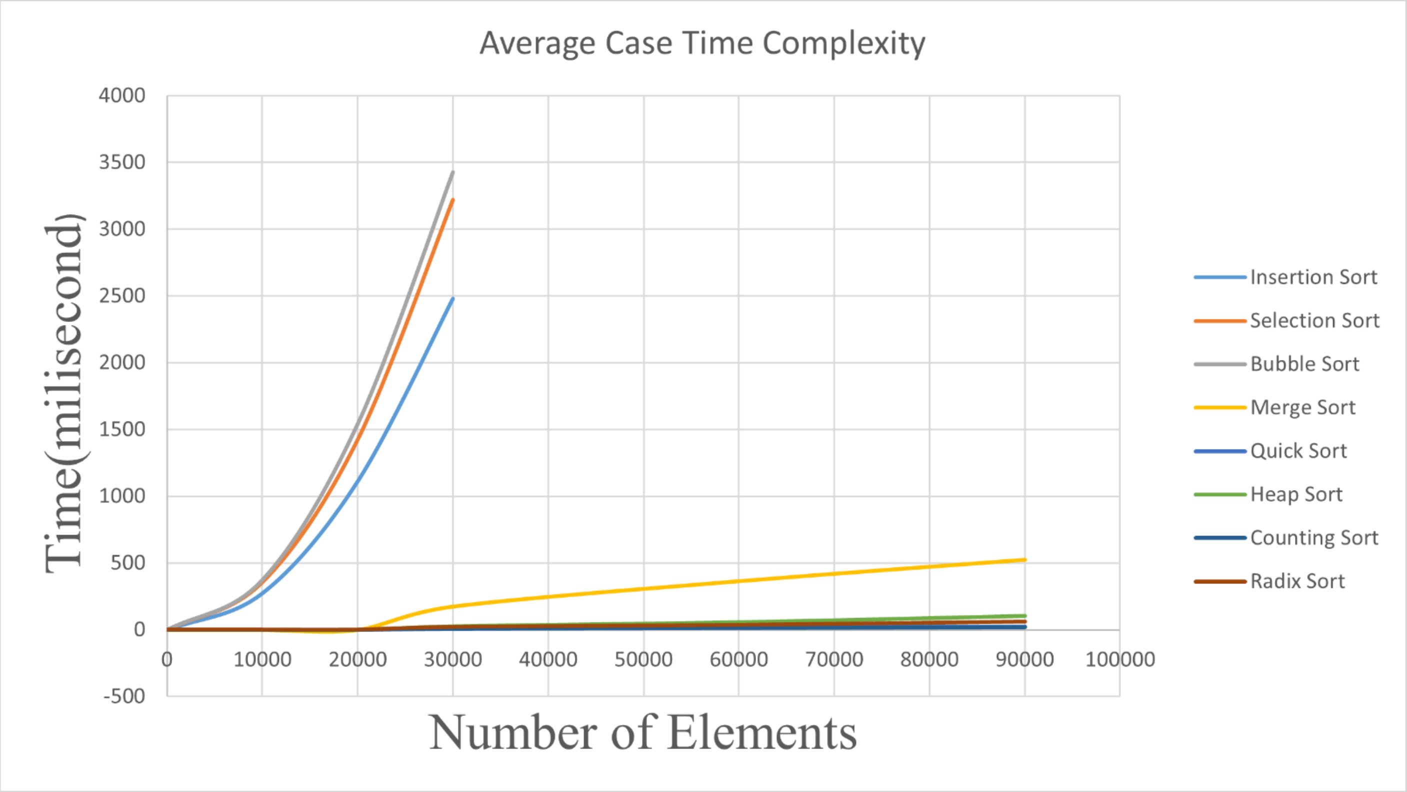
Experimental results show that for sorting algorithms (insertion sort, selection sort, bubble sort, merge sort, quick sort, heap sort, count sort and radix sort), the comparison times can predict the execution time well. As shown in Fig.2, figure $O(n2)$ Sorting algorithms, such as insertion sort, selection sort, Bubble sort takes longer than $O(nlogn)$ algorithms such as merge sort, heap sort, and quicksort. In Figure 3, a similar trend appears, $O(n2)$ worst-case algorithms such as insertion sort, selection sort Bubble sort is slower than worst-case $O(nlogn)$ algorithms such as merge sort and heap sort. Counting sort and radix sort algorithms are not suitable for this situation because they are not comparison-based sorting algorithms.

Figure1



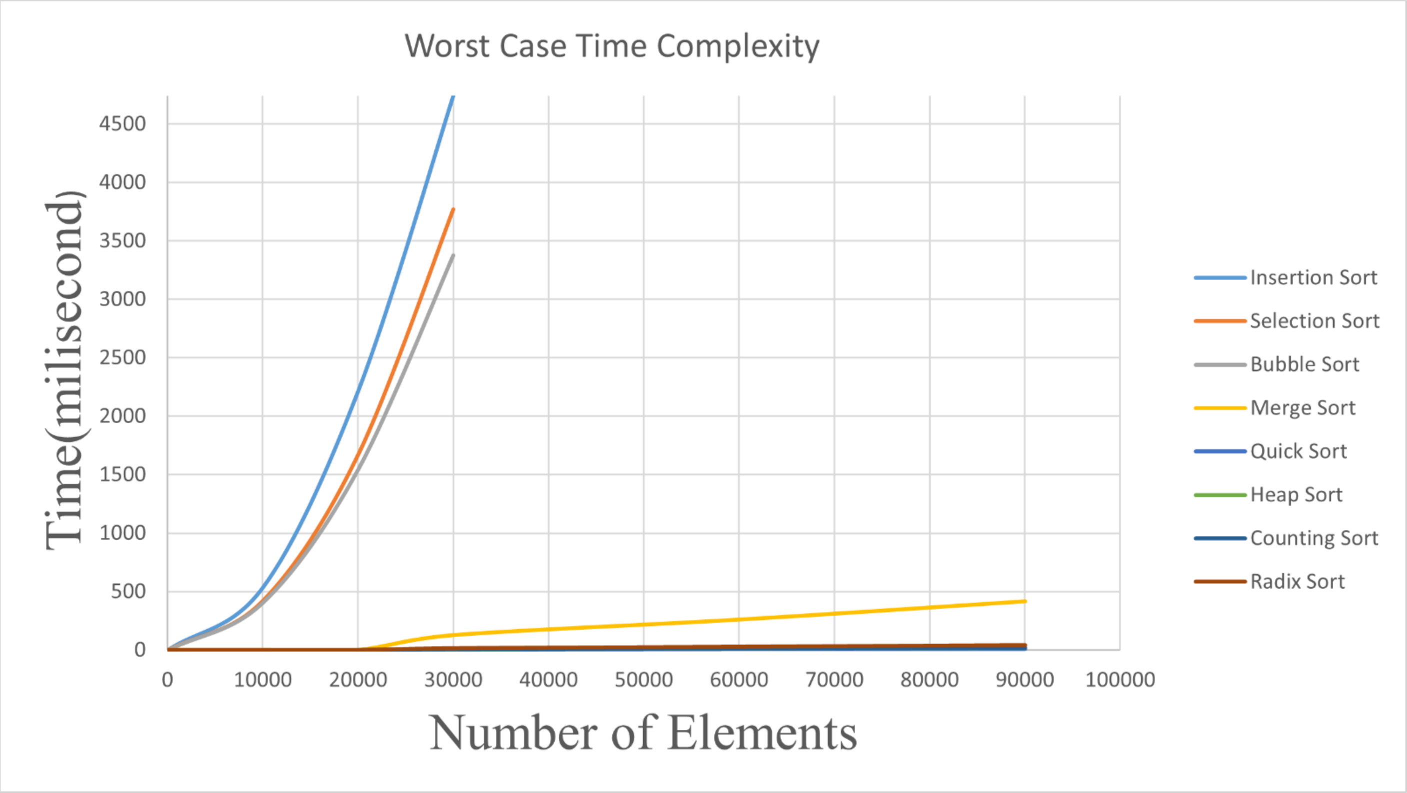
(Jump back to experimental data)

Figure2



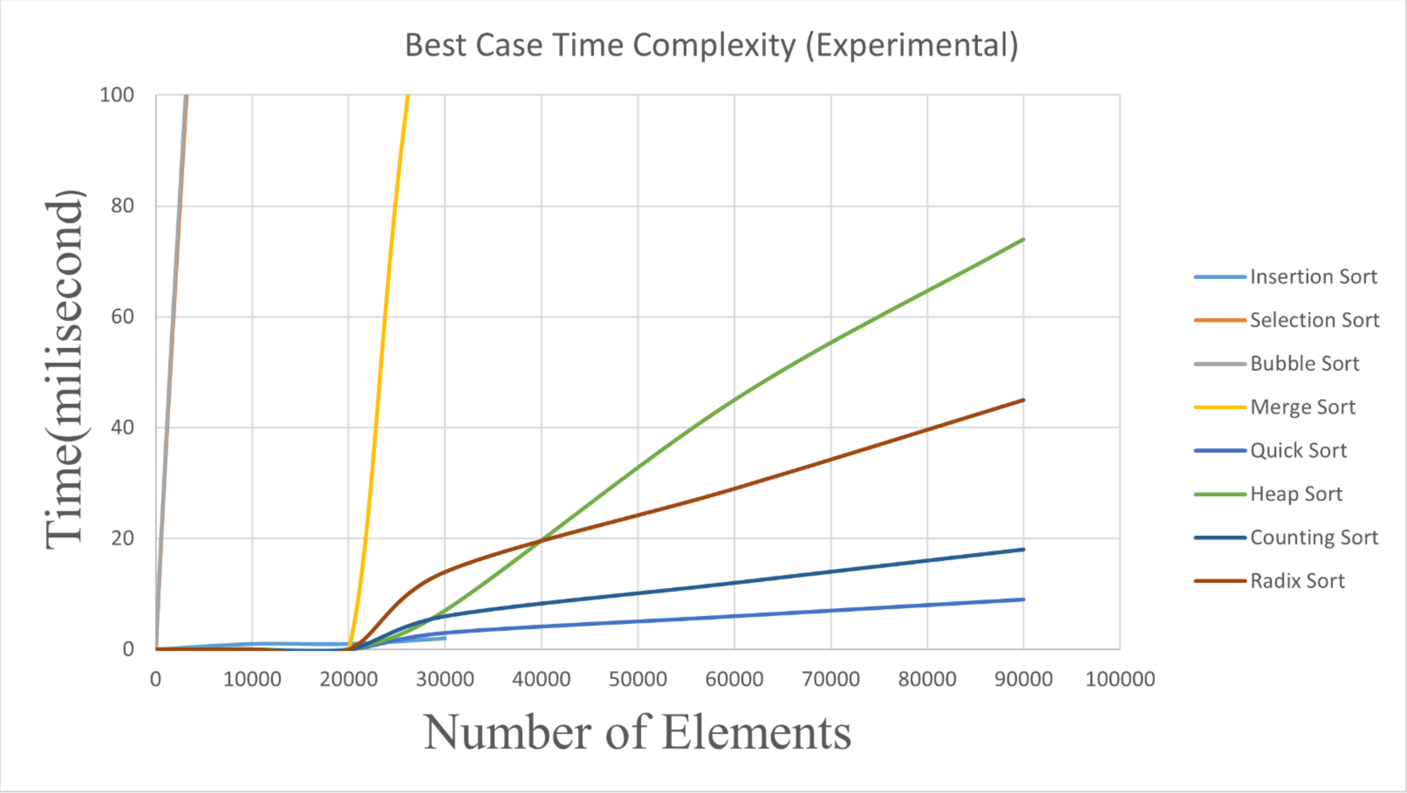
(Jump back to experimental data)

Figure3



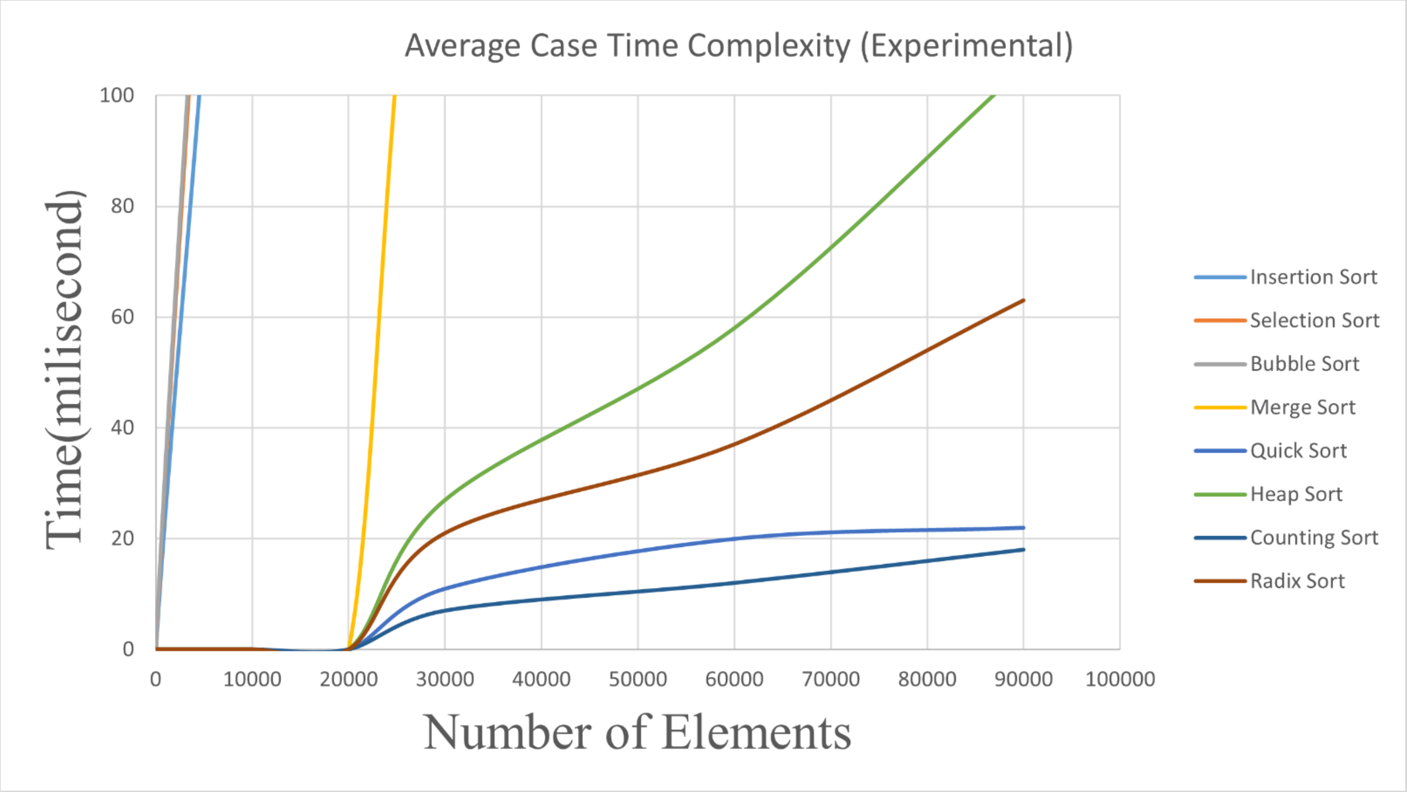
(Jump back to experimental data)

Figure4



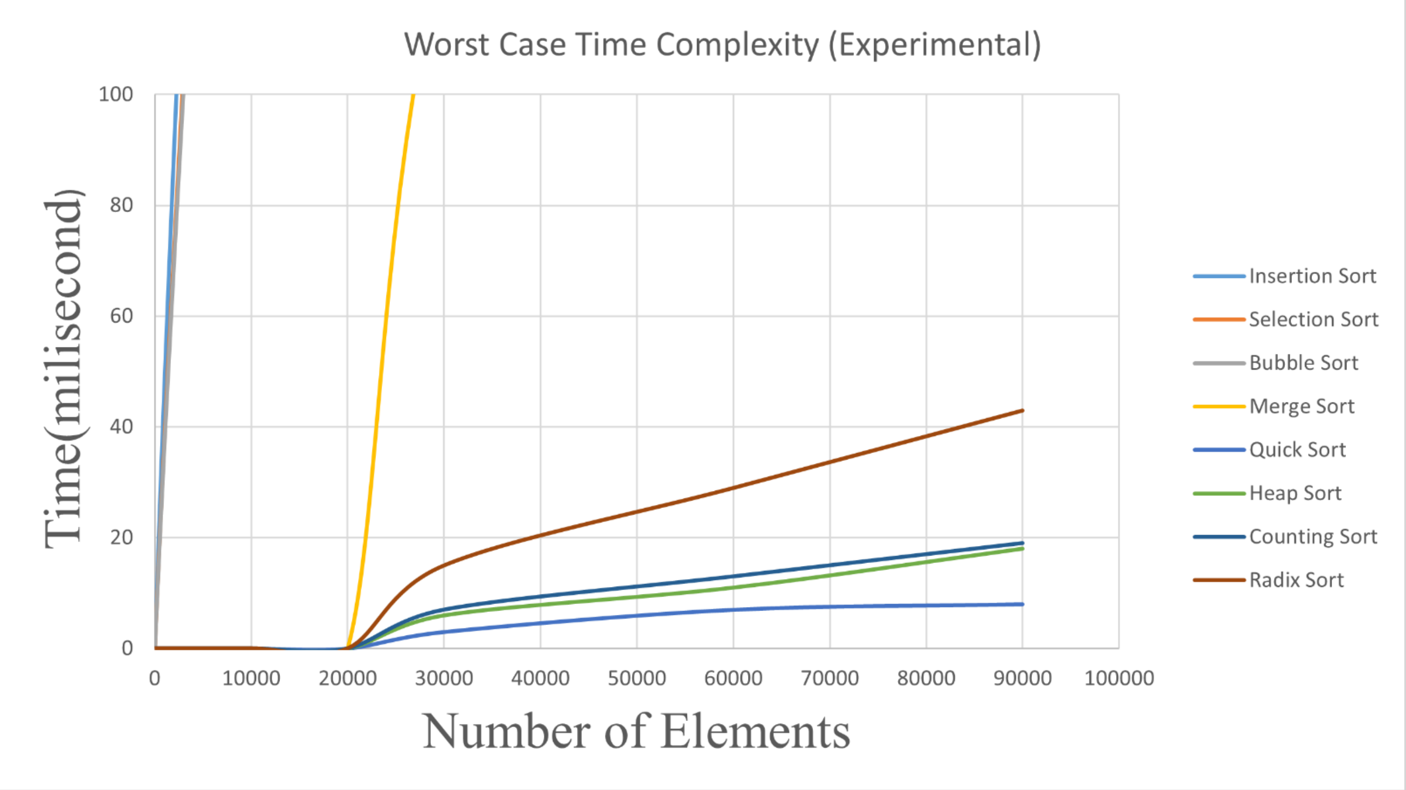
(Jump back to experimental data)

Figure5



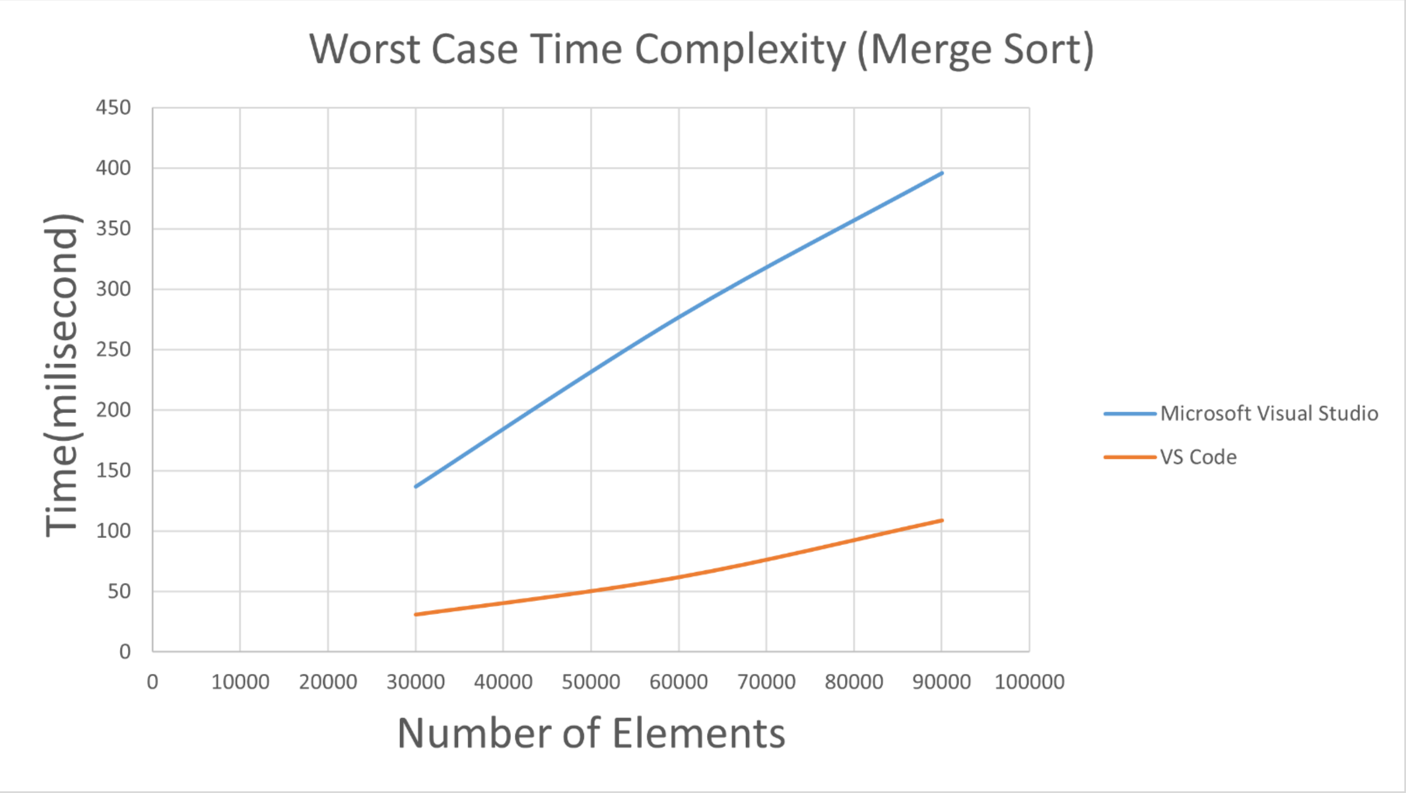
(Jump back to experimental data)

Figure6



(Jump back to experimental data)

Figure7



(Jump back to experimental data)

4 PROBLEM SOLVING AND ANALYSIS

There are other factors to consider implementing the more efficient solution to the problem mentioned in 1.2:

4.1 Brute Force Algorithm and Analysis

For **Algorithm 1** the input is array A storing the values of set S, size of array n, and value x. $num1$ is initialized to $A[i]$ and $num2$, in a nested loop, is initialized to $A[j]$ where $j = i + 1$. The solution is found when $num1 + num2 = x$ and the output is *true*. The output is *false* when the outer loop reaches it's end condition.

The worst case for **Algorithm 1** would be if there are no values in A whose sum is x. In that case, for every value of A, the inner loop would execute n times which would check the sub-array of $A[i..n]$ for the solution. The total running time of this can be described as $\Theta(n^2)$. The best case would be if the first two values of A have the sum x. In that case, total running time would be $\Theta(1)$ since no loops would occur; $num1$ and $num2$ would both be initialized and their sum would be x and the return statement is triggered. In the average case, the solution would be found in the middle of A but an accurate representation of the time complexity is still $O(n^2)$.

Algorithm 1: Brute Force Algorithm

```
Input: $A, n, x$
Output: true or false
for $i$ $0 < n$ do
  $num1 = A[i]$
  for $j$ $i + 1 < n$ do
    $num2 = A[j]$
    if $num1 + num2 = x$ then
      return true
    end
  end
end
return false
```

4.2 Efficient Algorithm and Analysis

For **Algorithm 2** the input is array H, size of the array m, and value x. Array H is the hash table storing the elements of set S.The elements themselves are actually objects storing both the number and the frequency that they occur in S. If n is the amount of elements in set S, m is the first prime number greater than n 1.3 which is the amount of slots in the hash table. The algorithm iterates through every slot of H. If there is an nonempty slot (line 4) then y is initialized to x minus the number at $H[i]$. The variable *ind* contains the hash key for y. The hash function used in this case is y modulo m. The while loop at line 7 is used for linear probing to determine whether y is present in the hash table. If y is present in H the output is true. Otherwise, after probing is complete i is incremented and the next slot in H containing a number is checked. Once all the slots in H are checked that means that there were not two numbers present whose sum is x and the output is false. The if statement at line 9 is used to check in case the two integers whose sum is x are equivalent e.g. 3 + 3 = 6. In that case the frequency of y is checked to make sure that number is present in set S more than one time.

The worst case for **Algorithm 2** would be if every integer in set S had the same hash key. This would mean that this algorithm would act similarly to **Algorithm 1**; the loop used for probing would instead execute n times and check the sub array $H[i..n]$ to find if y is present in H. This instance would lead to a running time of $\Theta(n^2)$. However, unless specifically induced this is not likely to occur. If uniform hashing is assumed, the amount of probes in an unsuccessful search is constant and is at most $1/(1\ \alpha)$ where α is the load factor n/m [3]. The load factor of H is at most .75 so the maximum amount of probes for an unsuccessful search would be $1/(1\ .75)$ or 4 assuming uniform hashing. Since probing time complexity is constant $O(4)$, the time complexity of the whole algorithm would be $O(m)$, m being the size of the hash table.

efficient solution to the problem mentioned in 1.2: what the size of the hash table should be, should certain values be omitted from insertion into the hash table, and how to eliminate duplicates. The size of the hash table should be enough to accommodate the n amount of integers in the set, be a prime number for the hash function, and be large enough to maintain around a 75% load factor [2]. A simple way to determine what the size would be the next prime number after n 1.3. An early idea that we considered in reducing the amount of numbers that need to be searched was to remove numbers that are $<\ x$ since two positive integers could not be the sum of x if one of them were larger than x. This however would not take into consideration negative numbers; in that case one of the values could be larger than x if the other were negative e.g.$x = 8$, $i = 10$, $j = \ 2$, $i + j = x$. Due to this, all numbers of the set should be included in the hash table. However, if a number is repeated multiple times, this would cause unnecessary searches. To combat this, the integers are stored in an object that contains the number and the frequency of the number in the hash table. This insures that each integer is only inserted once into the hash table. This approach is illustrated in Fig.7.

Algorithm 1: Brute Force Algorithm

```
Input:  $A, n, x$
Output:  *true or false*
for $i$ 0 < $n$ do
  $num1 = A[i]$
  for $j$  $i + 1$ < $n$ do
    $num2 = A[j]$
    if $num1 + num2 = x$ then
      return *true*
    end
  end
end
return *false*
```

5 CONCLUSION

To compare the performance of sorting algorithms, we first predicted the performance based on the time complexities of the algorithms as a function of n. Different input sizes were chosen and the theoretical predictions were represented on a graph. The worst-case, best-case, and average-case input values and time complexities were identified. Now we tested all the sorting algorithms on one machine with DECEMBER 2021 7 the same input sizes and recorded the time it took for them to run. Each algorithm’s running time was compared to its predicted theoretical performance. From our analysis we concluded that a good predictor of execution time is the number of comparisons an algorithm needs to sort. To solve the problem of finding whether or not, given a set S of n amount of integers, there exists two elements in S whose sum is integer x. This problem was successfully solved using a brute force method and with a more efficient algorithm. The more efficient algorithm incorporated open addressed hash tables and linear probing. The time complexity of the brute force algorithm was O (n 2) and the efficient algorithm had a time complexity of O (m)

ACKNOWLEDGMENTS

We would like to express our special thanks to Professor Matin Pirouz as well our University who gave us this opportunity to do this project. We offer sincere appreciation for the learning opportunities provided by the university. Our completion of this project could not have been accomplished without the support of each one our group members: Mingzan Liu, Leon Kantikov, Harsh Mohan Sason, Gustavo Lita-Chaves and Joel Ramirez-Mariscal. All of our dedication and determination helped us in the completion and submission of this project.

REFERENCES

[1] Young, J. & Potler, A. First occurrence prime gaps. Mathematics of Computation 52, 221–224 (1989). URL http://www.jstor.org/stable/ 2008665.

[2] (2015). URL https://docs.oracle.com/javase/6/docs/api/java/ util/HashMap.html.

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. Introduction to algorithms (MIT press, 20