Term Project Report

Leon Kantikov, Mingzan Liu, Joel Ramirez-Mariscal, Harsh Mohan Sason, Gustavo Lita-Chaves
Matin Pirouz

Abstract—In this report, the theoretical time complexity and experimental time complexity of eight sorting algorithms are analyzed and compared. Namely, 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 sorting algorithms mentioned above. In the problem solving and analysis section, we 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 is solved first in a brute force method where every possible pair of numbers is checked indiscriminately. Our goal is to now solve the problem using a more efficient method. To do this we will be utilizing an open-addressed hash table and linear probing to insert and search for elements in the hash table.

Index Terms—algorithm, time complexity, sorting, hash table, load factor, linear probing, maximal prime gaps

1 Introduction

1.1 Part 1: Comparison of Sorting Algorithms

To compare theoretical time complexity and experimental time complexity of the sorting algorithms (insertion sort, selection sort, bubble sort, merge sort, quicksort, heapsort, counting sort, and radix sort); the theoretical time complexities (best, average, worst case) are recorded. Then, to get the experimental time complexities, different input array types and data sizes are passed into the sorting algorithms and run at a set amount of times. The results are then graphed to visualize the experimental time complexities.

1.2 Part 2: Problem Solving and Analysis

The problem that we solved was finding whether or not, given a set S of n amount of integers, there exists two elements in S whose sum is integer x. To solve this problem using the brute force method, similar to insertion sort, each value is compared to the other using a nested loop. One value i is compared to every other value in set S until there is an integer j where $i+j=x$. The problem that needs to be solved here is efficiently determining whether i has another value in S, j, that can sum to x. If this problem is solved, the need to compare i to every integer in S to find j would be eliminated. A solution to this problem is to use a hash table to determine whether j exists in S for every value in S since the location of where j is in S would be known.

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 exists two elements whose sum is x in set S

2 RELATED WORK

A concern when creating an efficient solution for **Part 2** was whether finding the next prime number after n would significantly impact the performance of the algorithm as n approached infinity. Fortunately, it is evident that as numbers get larger, the maximum gaps between prime numbers gets larger at a decreasing rate. For example, before 113 the largest gap between prime numbers is 14 which is roughly 9% of 113. But at 370261, the largest gap between prime numbers is 112 numbers apart which is roughly .0003% of 370261. As n increases it is clear that finding the next prime number would be insignificant when compared to the actual time complexity of the algorithm. [1]

3 Comparison of Sorting Algorithms

To compare the theoretical time complexity and the experimental time complexity of the sorting algorithms; the theoretical worst-cast, average-cast, and best-case complexities of the sorting algorithms are considered. In addition, the experimental time complexities have to be considered as well. To obtain the experimental time complexities, tests have to be run on each of the sorting algorithms. For the testing, the inputs and input data sizes have to be carefully chosen to avoid an unnecessary number of experiments and unnoticeable experiment results. Also, to get stable run times of the sorting algorithms, each input and input data size is run multiple times and the average is taken for each sorting algorithms.

3.1 Theoretical Time Complexities

Insertion sort, the best-case time complexity is $O(n)$, achievable with sorted input. The element before the key is always less than or equal to the key because it is already sorted. Therefore, the while-loop is never entered and the outer for-loop continues until input size n. The average-case time complexity is $O(n^2)$, with randomly sorted input. The element before the key is sorted for some, the while-loop is entered for some elements but not all elements; resulting in $n^2/2$ comparisons and exchanges. The worst-case time complexity is $O(n^2)$, with reverse sorted input. Each element has to compare with all the elements on the left

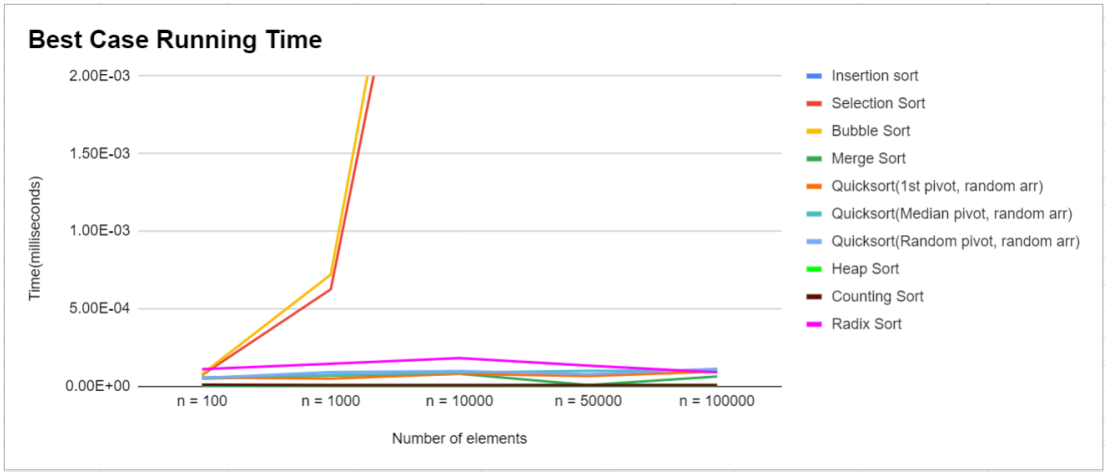

Fig. 1

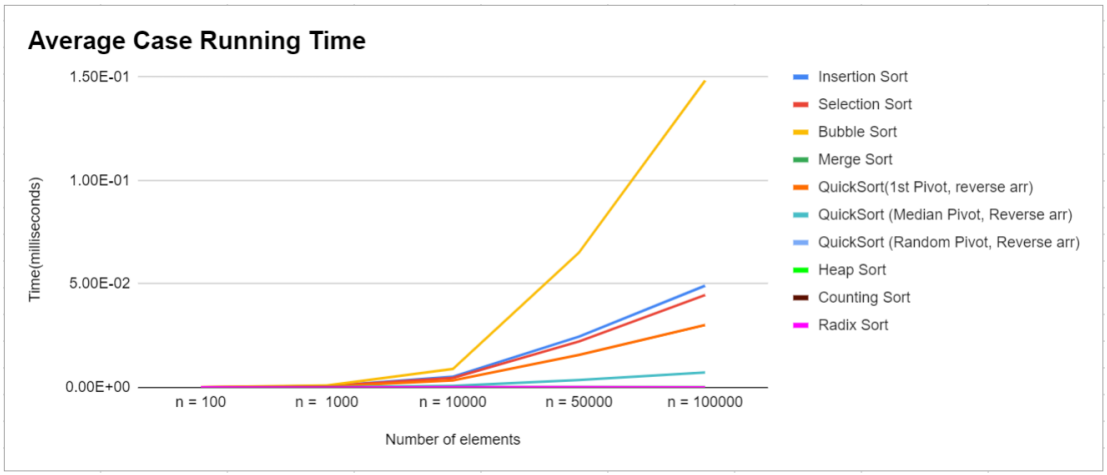

Fig. 2

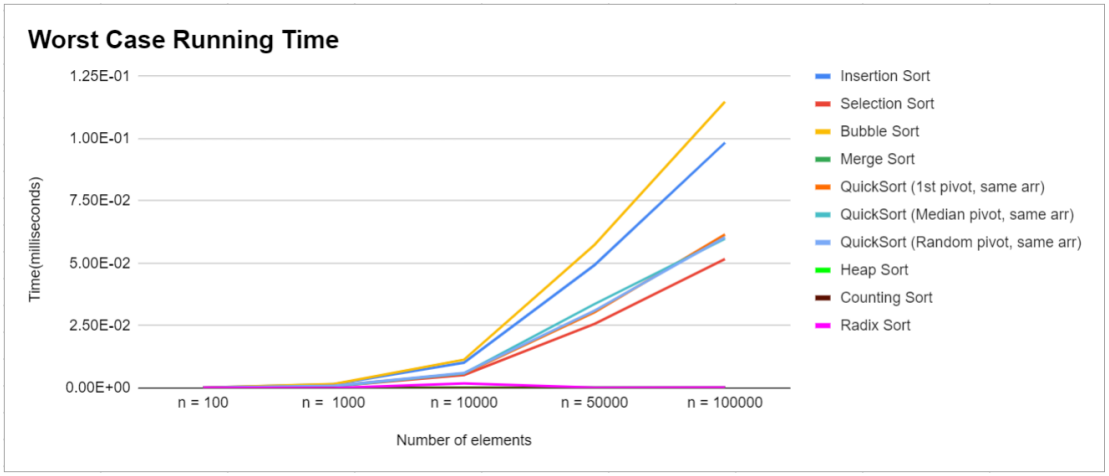

Fig. 3

side of the array.

Selection sort, the best-case, average-case, and worst-case time complexity is $O(n^2)$, achieved with any input. Each element i makes comparisons with elements $i+1$ until input size n.

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 input respectively. Each element will compare with the rest of the input. Since one element is pushed to the end of the array that is in the correct order, elements needed to be compared decrease by 1 for each traversal through the array.

Merge sort, the best-case, average-case, and worst-case time complexity is $O(nlogn)$, achieved with any input. The input will be broken in half recursively and merged together regardless of input.

Quicksort, for random pivot, median pivot, and first element pivot, the best-case and average-case time complexity is $O(nlogn)$, achievable with randomly sorted input. When the pivot allows the input to be partitioned $n/2$ evenly on both sides. The worst-case time complexity is $O(n^2)$, achieved with reverse sorted or sorted input. This creates a partition where one side will have one element and the other with $n-1$ elements. Different pivot changes the likelihood of uneven partition.

Heap sort, the best-case, average-case, and worst-case time complexity is $O(nlogn)$, achieved with any input. Each element i makes comparisons with elements $i+1$ until input size n. Building a max heap takes $O(n)$ time, swapping the first element with $i = input\ size\ n$, where n will decrease by 1 for each iteration. Maintaining heap property after each swap is $O(logn)$ for $(n-1)$ times.

Counting sort, the best-case, average-case, and worst-case time complexity is $O(n+r)$, where n is the input size n and r is the range of the input. Despite the worst-case time complexity is $O(n+r)$, run time is slow when the range of input is significantly larger than the input size due to an array of size r has to be created and traversed through. The best-case time complexity is also $O(n+r)$, it occurs when the range of the input r is significantly smaller than the input size n.

Radix sort, the best-case, average-case, and worst-case time complexity is $O(d*(n+k))$, where d is the number of digits of the largest element in an input, n is the size of the input, and k is the base used (10). Best-case input is when the number of digits for each element is the same. Average-case input is when the number of digits for each number of the element is not significantly larger or smaller than each other. The worst-case input is when the number of digits for each element is significantly larger than each other. The number of digits is the number of times the counting sort will be run.

4 EXPERIMENT

4.1 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 Ryzen 5 5600x CPU. The code is written in C++ and the Chrono library is used to time the sorting algorithms. The code is compiled and run using VScode with GCC compiler (g++). An array of the same value, random array, and reverse sorted array is chosen as input types. These three array types account for the worst-case, average-case, and best-case inputs for the mentioned sorting algorithms. For data size, $n = 100,\ 1000,\ 10000,\ 50000,\ 100000$ is chosen. For each input type and data size, the sorting algorithms are run 10 times and the average is recorded in milliseconds.

4.2 Experimental Data

From the result of the experiment, radix sort, counting sort, merge sort, heap sort, and quicksort seem to perform the best out of all the tested sorting algorithms considering from Fig.1-Fig.3. Median quicksort, in particular, performed the best out of the three versions of quicksort (first pivot, median pivot, and random pivot). Insertion sort, selection sort, and bubble sort performed the worst considering their worst-case and average-case time complexity is $O(n^2)$. Despite insertion sort's best-case time complexity being $O(n)$ in Fig1; in Fig.2 and Fig.3, the overall performance of insertion sort is worst than radix sort, counting sort, merge sort, heap sort, and quicksort using median pivot.

4.3 Experimental Data Analysis

Insertion sort, the experimental graph in Fig.4 conform with the asymptotic analysis as the experimental and theoretical time complexity is $O(n)$ for the best-case complexity. The average case and worst case in Fig.5 and Fig.6 for the experimental graph conform with the asymptotic analysis as both experimental and theoretical time complexity is $O(n^2)$. The experimental graphs in Fig.5 and Fig.6 show a significant increase in time as data size increase from $n = 1000$ to $n = 10000$.

Selection sort, in Fig.4 - Fig.6. as the input increases to $n = 10000$, a sudden jump in the time taken by the sorting algorithm. As increase the input more than $n = 10000$, the time taken increased significantly again, which conforms with the asymptotic analysis of $O(n^2)$ for best, worst, and average-case time complexities.

Bubble sort, the experimental graph in Fig.4 did not match with the asymptotic analysis of $O(n)$ for the best-case time complexity. The experimental graph showed $O(n^2)$, this is due to the array that was used to test the time taken for bubble sort. The array of the same element was used, as a result, the way bubble sort was written in this experiment will continue to traverse until $n-1$ times, unless it is a strictly sorted array, where the elements are not equivalent. The average and worst-case experimental results conformed with the asymptotic analysis of $O(n^2)$ in Fig.5 - Fig.6. There are significant spikes in time as the data size increase more than $n = 1000$.

Merge sort, the experimental graphs in Fig4-Fig6 match with the asymptotic analysis as the best, average, and worst-case time complexity is $O(nlogn)$. The experimental graphs spike in time minimally as the data size increased more than $n = 1000$, across best, average, worst case experimental graphs. Since merge sort break input in half

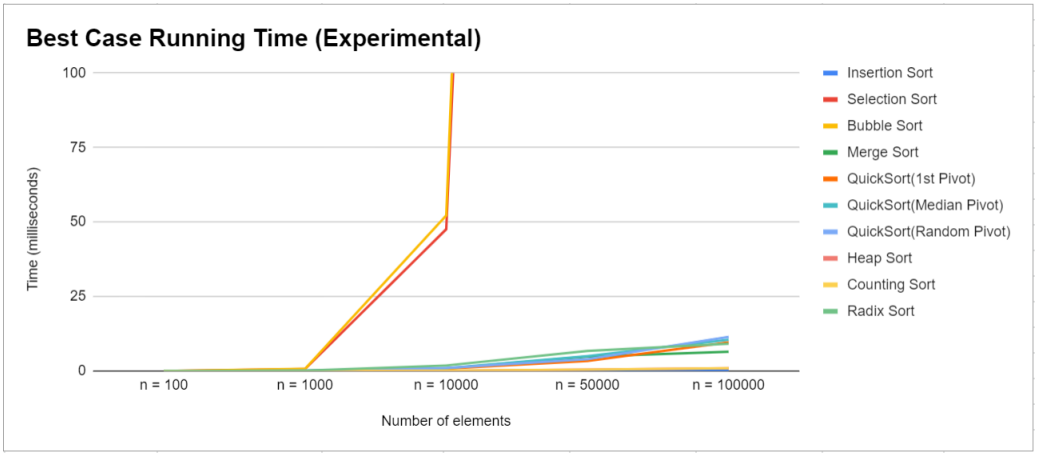

Fig. 4

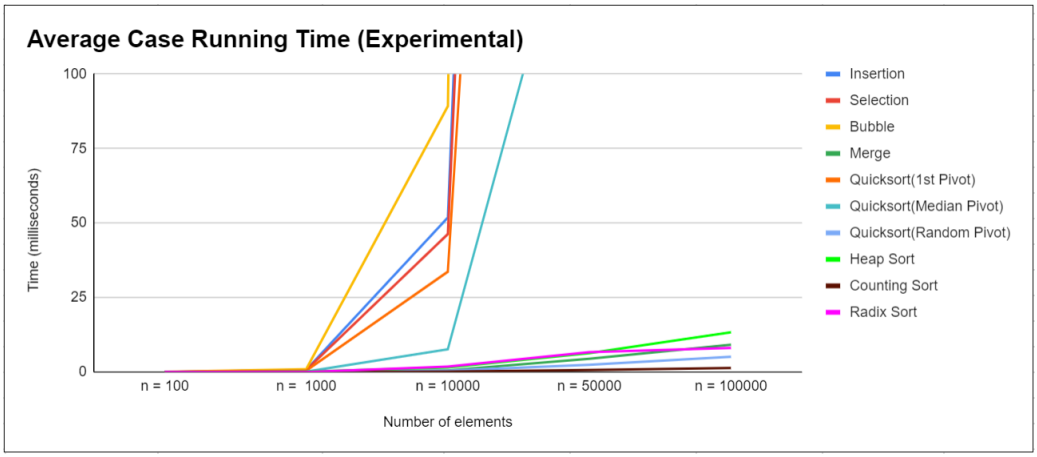

Fig. 5

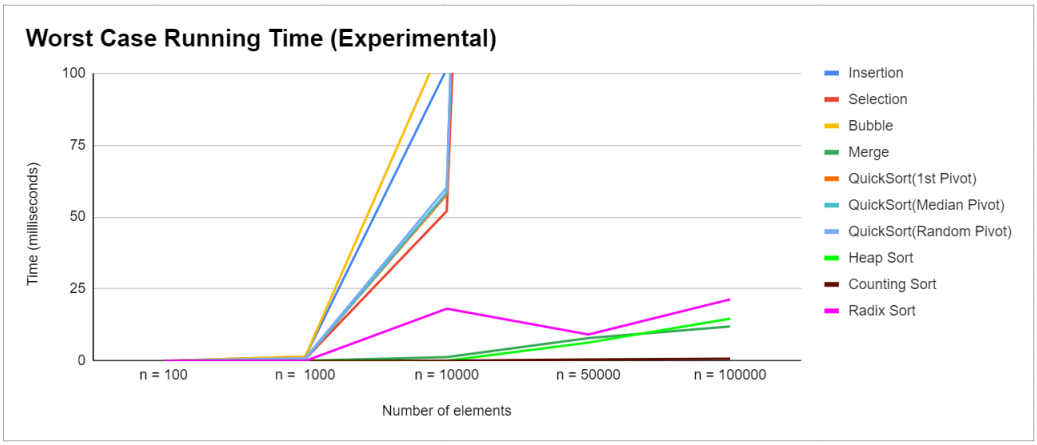

Fig. 6

recursively regardless of input, the experimental result conforms with the asymptotic analysis.

Quicksort, first element pivot, median pivot, and random pivot have best-time complexity conforms with the asymptotic analysis of $O(nlogn)$. In Fig.4, the time taken spikes minimally as the input increase more than $n = 10000$ and $n = 50000$. The experimental average time complexity in Fig.5 did not conform with the asymptotic analysis of $O(nlogn)$. The experiment result revealed $O(n^2)$ average-case time complexity. Due to the pivot choice, the input is partitioned unevenly by a significant margin, resulting in the average-case running time of $O(n^2)$. The pivot choice affects the chance of uneven partitions. The experimental worst-case time complexity in Fig.6 conforms with asymptotic analysis of $O(n^2)$ as the reverse sorted input causes significant uneven partitions of the input. As the input size increase more than $n = 1000$ and $n = 10000$, the time taken spiked significantly.

Heap sort, the experimental results in Fig.4 - Fig.6 best, average, worst-case conformed with the asymptotic analysis of $O(nlogn)$ for all three cases. Heap sort disregard input type, build a heap and maintain heap after each swap $n - 1$ times. As a result, the experimental result across three experiment graphs are conforming 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, the best-case experimental result in Fig.4 conforms with the asymptotic analysis of $O(d * (n + k))$. The number of digits are equivalent for all the elements of an input, resulting in no extra counting sorts to be performed to accommodate for elements with extra digits. The average-case experimental result in Fig.5 conforms with the asymptotic analysis of $O(d * (n + k))$. The difference in the number of digits across all elements is not significantly apart. However, since extra counting sort will be performed to accommodate the elements with more digits, the time taken is longer than the best case shown in Fig.4. The worst-case experimental does not conform with the asymptotic analysis of $O(d * (n + k))$. The error might be that since the input for the worst case is a randomly sorted input, the digits for some run might have varied by a significant amount due to a randomly generated input, causing the graph shown in Fig.6.

4.4 Is the Number of Comparisons A Good Predictor of Execution Time?

Based on the experimental results, the number of comparisons is a good predictor of execution time for the sorting algorithms (insertion sort, selection sort, bubble sort, merge sort, quicksort, heapsort, counting sort, and radix sort). As shown in Fig.2, the average case running time graph, $O(n^2)$ sorting algorithms like insertion sort, selection sort, and bubble sort takes longer to sort a input array compared to $O(nlogn)$ algorithms like merge sort, heap sort, and quicksort. In Fig.3, the similar trend occurs, the $O(n^2)$ worst-case algorithms like quicksort, insertion sort, selection sort, and bubble sort are slower than the $O(nlogn)$ worst-case algorthms like merge sort and heap sort. The counting sort and radix sort algorithm does not apply in this case since they are not comparison based sorting algorithms.

5 Problem Solving and Analysis

There are other factors to consider to implement the more 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

```
1  Input: A, n, x
2  Output: true or false
3  for i ← 0 < n do
4      num1 = A[i]
5      for j ← i + 1 < n do
6          num2 = A[j]
7          if num1 + num2 = x then
8              return true
9          end
10     end
11 end
12 return false
```

5.1 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

n = 10

size of hash table = next prime # < (n * 1.3)

size of hash table = 17

S = 15, 10, 19, 2, 16, 6, 20, 8, 9, 13

Hash function: |S[i]| % size of hash table

X = 8

HashTable

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
NULL	NULL	19	2	20	NULL	6	NULL	8	9	10	NULL	NULL	13	NULL	15	16

0	1	2
NULL	NULL	19

y = x - 19

y = -11

|-11| mod 17 = 11

HashTable[11] = NULL

0	1	2	3
NULL	NULL	19	2

y = x - 2

y = 6

|6| mod 17 = 6

HashTable[6] = 6

return true

Fig. 7: The algorithm loops until an nonempty slot is found, calculates y and it's hash key, and determines whether y is in the hash table or not. If yes, return true if not iterate until end of hash table.

Algorithm 2: Efficient Algorithm

```
1  Input: H, m, x
2  Output: true or false
3  for i ← 0 < m do
4      if H[i].frequency ! = 0 then
5          y = x − H[i].n
6          ind = hashkey(y, m)
7          while H[ind].frequency ! = 0 do
8              if H[ind].n = y then
9                  if y = (x/2) and
                      H[ind].frequency < 2 then
10                     ++ind
11                     break
12                 end
13                 return true
14             end
15             if ind >= m − 1 then
16                 ind = 0
17             end
18             else
19                 ++ind
20             end
21         end
22     end
23 end
24 return false
```

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 $O(n^2)$.

5.2 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.

6 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

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, 2009).