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 sorting algorithms 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, each value is compared to the other using a nested loop. One solution is to create an array A to record the integer in each pass by increasing its corresponding value in array A to 1, then for the integer $S[i]$ in every pass, check if its corresponding value for $x - S[i]$ is 1, if yes, that means $x - S[i]$ exists in the set S, otherwise record $S[i]$ to A. Continue the process in each pass until reach the end of set S. If it hit the end of set S, then return false. There are only two single for loop in this algorithm 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 rest of 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

```
1 Bubble-Sort (A)
2 for i = n to 1 do
3     key = i
4     for j =0 to i do
5         if A[j] > A[key] then
6             key = j
7 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

```
1  Merge (A, l, m, r)
2  if A.len > 1 then
3      let lv be a new array
4      let rv be a new array
5      len1 = m - l
6      len2 = r - m
7      for i = 0 to len1 do
8          lv.push_back(A[i+1])
9      for i = 0 to len2 do
10         rv.push_back(A[m+1])
11     lv.push_back(INT_MAX)
12     rv.push_back(INT_MAX)
13     j = 0, k =0, pos = l
14     for i = l to r do
15         if lv[j] <= rv[k] then
16             A[i] = lv[j]
17             j = j + 1
18         else
19             A[i] = rv[k]
20             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

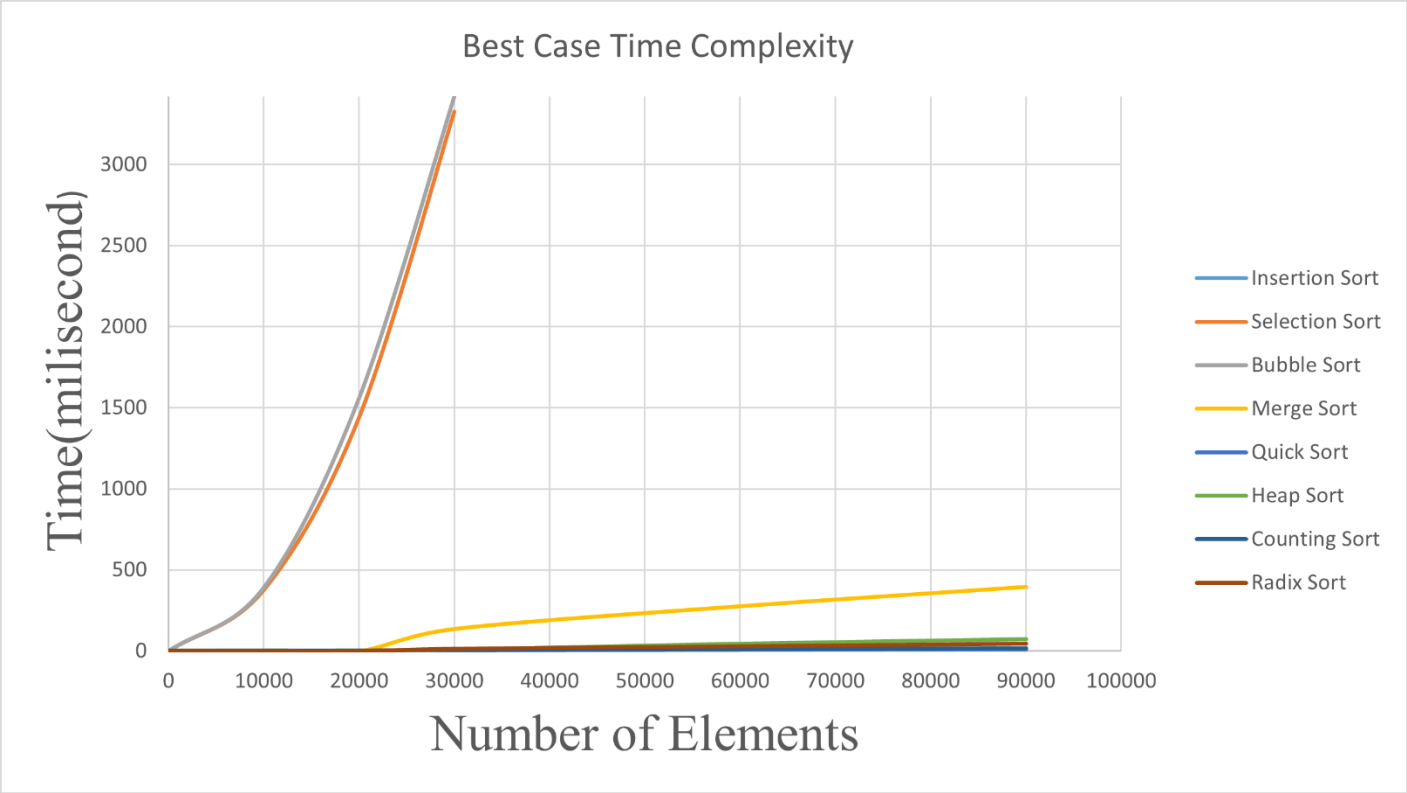
Heap sort the best, average and worst cases of experimental results in Fig.4-Fig.6 all conform to $O(nlogn)$ asymptotic analysis. Heap sort, regardless of input type, builds the heap and maintains the heap $n-1$ times after each swap.
The results show that the experimental results of the three experimental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

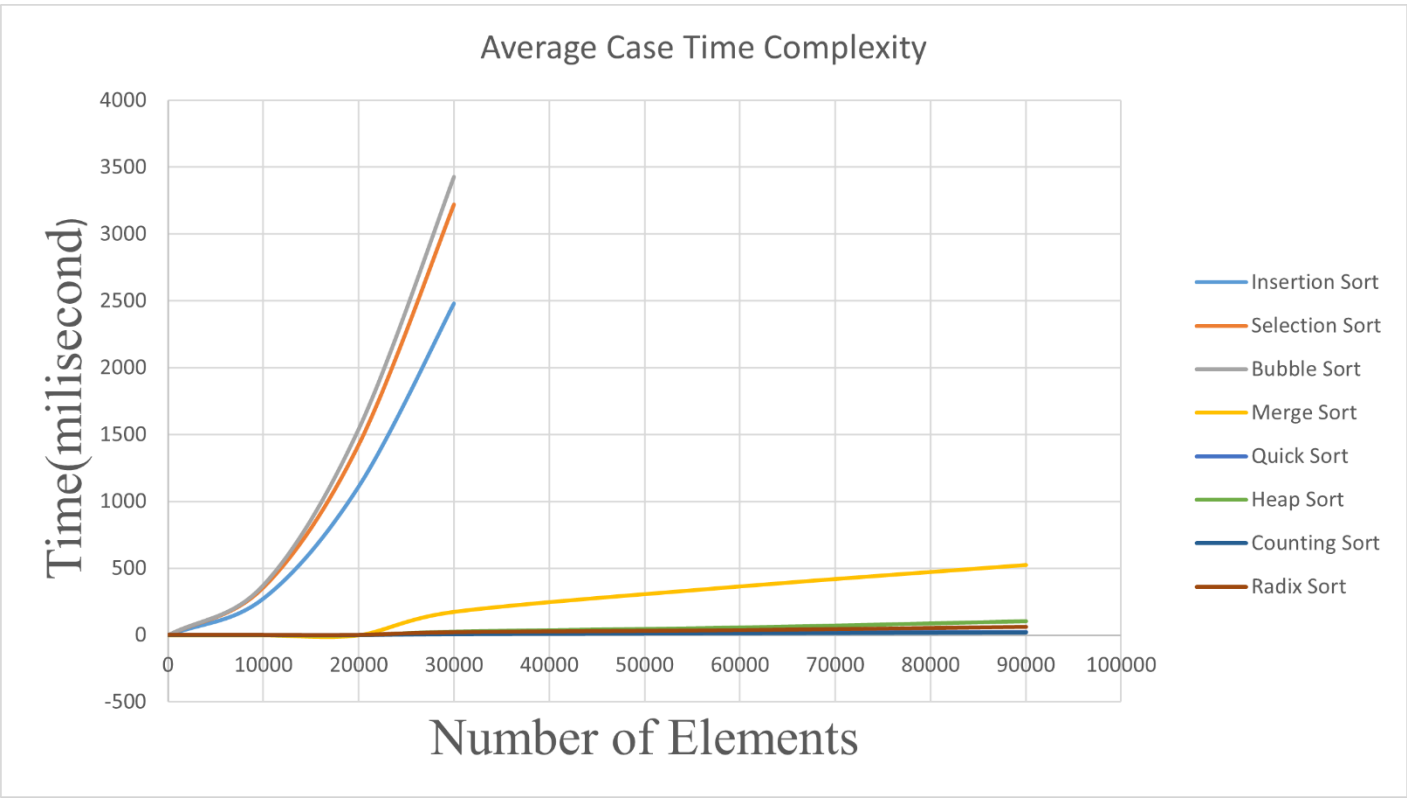
Experimental results show that for sorting algorithms (insertion sort, selection sort, bubble sort, merge sort, quick sort, heap sort, count sort and radix sort), the comparison times can predict the execution time well. As shown in Fig.2, figure $O(n2)$ Sorting algorithms, such as insertion sort, selection 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



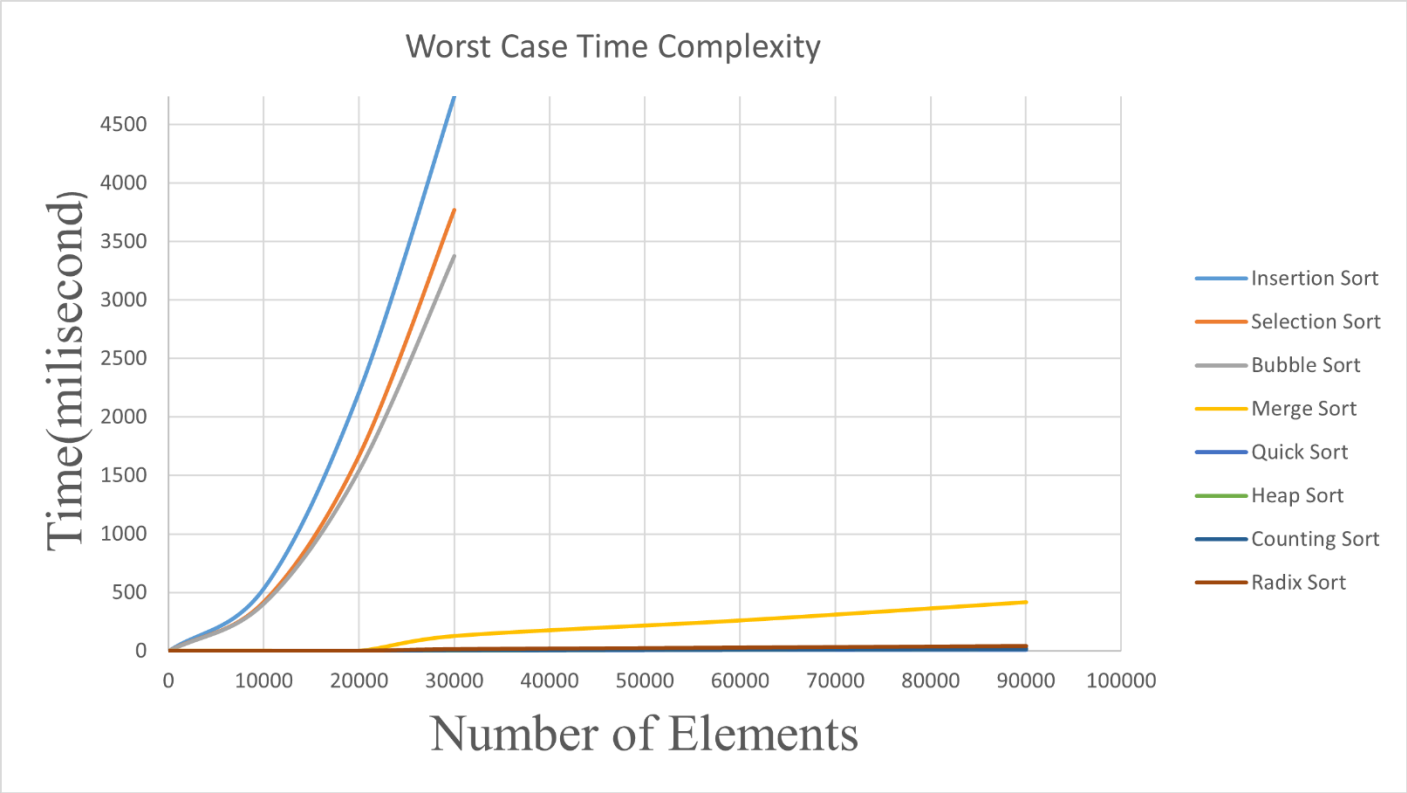
(Jump back to experimental data)

Figure2



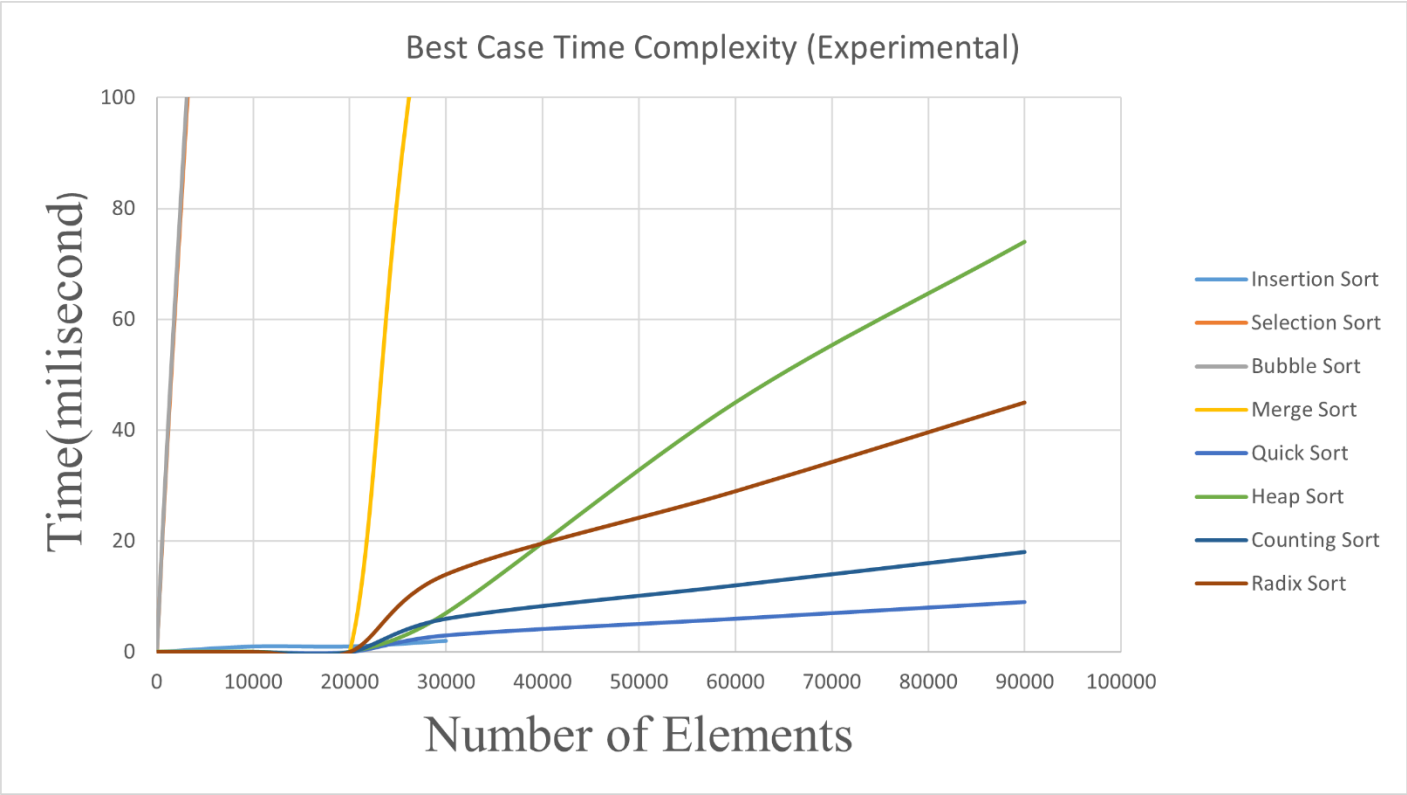
(Jump back to experimental data)

Figure3



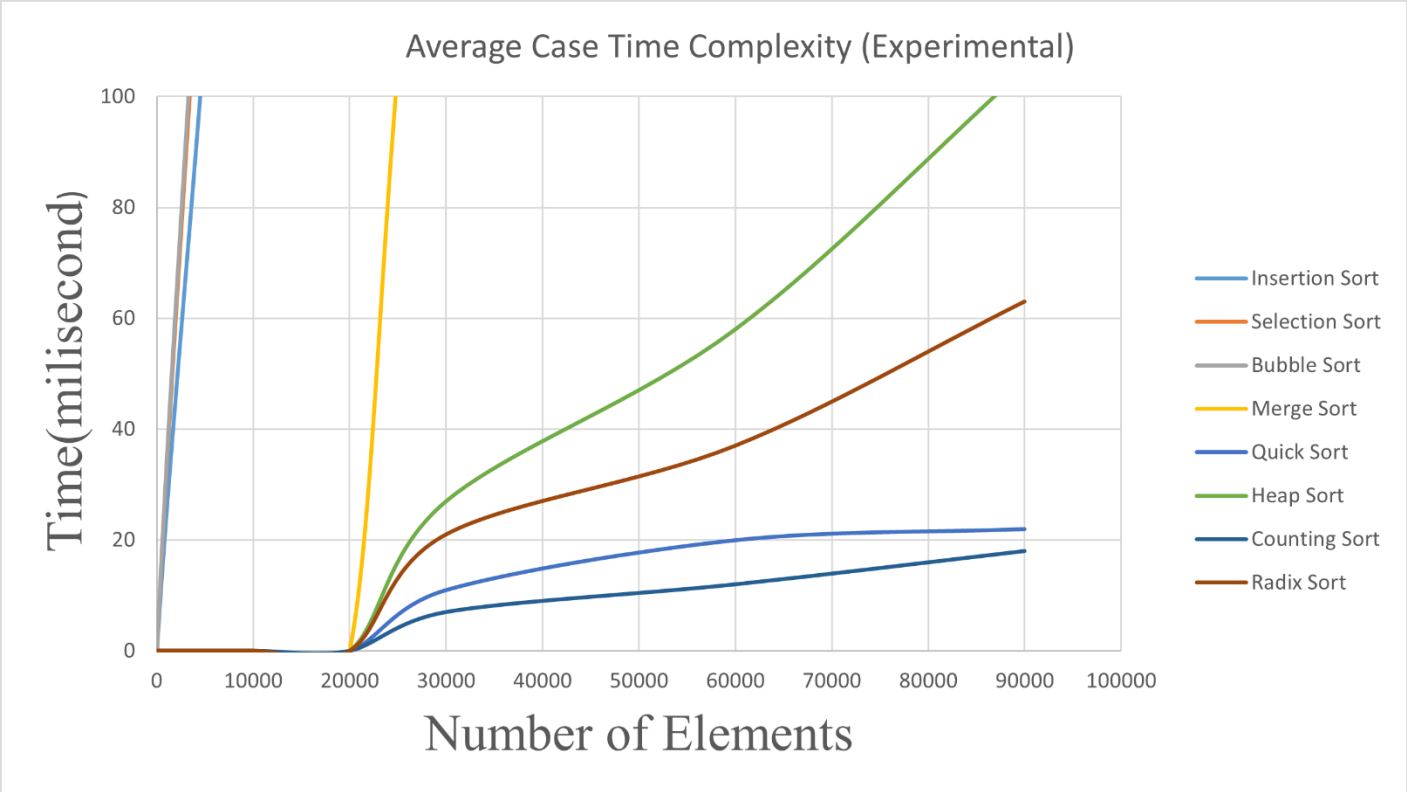
(Jump back to experimental data)

Figure4



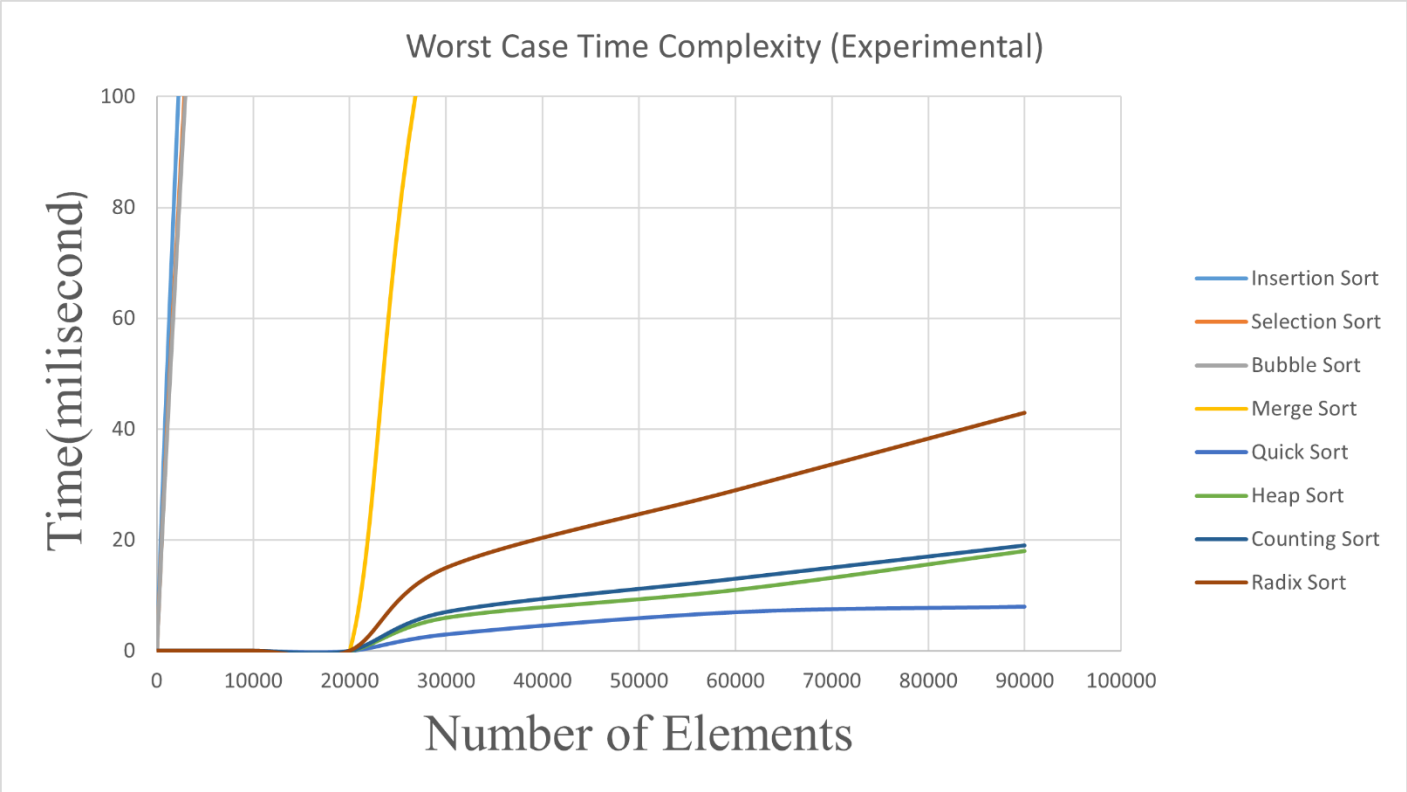
(Jump back to experimental data)

Figure5



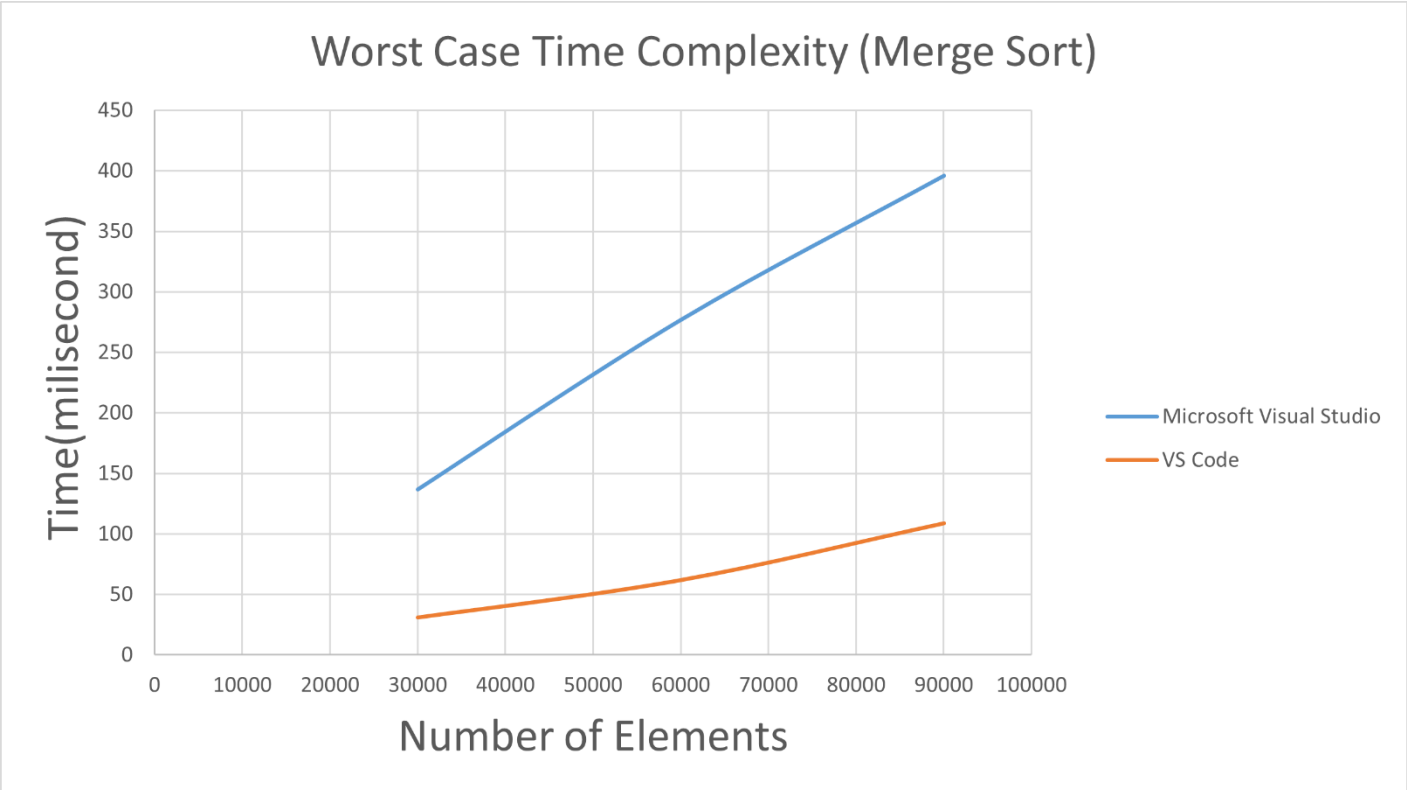
(Jump back to experimental data)

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: One concern is how to design the size of the array A. In order to cover all the integers in the set S, what we do is we search the whole input array just for finding the minimum and maximum integers in the set S. For include every number in the set S, the array A should be able to store every number in the range from min to max. In order to calculate the range, there are two cases should be considered: one is the maximum is positive, another is the maximum is negative. in the case 1- case 2, the maximum is positive, the size of the array A should be

$$maximum - minimum + 1.$$

In another case, case 3, if the maximum is negative, then the size of the array A should be

$$(-maximum) - (-minimum) + 1$$

Case 1:

Set S:

1	3	3	-2	2	5

From the set S above, the minimum is -2, the maximum is 5. Therefore, in order to contain every number in the input array, the size of the array used for storing should be $5 - (-2) + 1 = 8$

Array A:

Index	0	1	2	3	4	5	6	7
Number	0	0	0	0	0	0	0	0

Case 2:

Set S:

1	3	3	5	2

From the Set S above, the minimum is 1 and the maximum is 5. Therefore, for being able to contain every number in the input array, the size of the array used for storing should be

$5 - 1 + 1 = 5.$

Array A:

Index	0	1	2	3	4
Number	0	0	0	0	0

Case 3:

Set S:

-2	-1	-3	-3	-5

In this case, the minimum integer is -1 and the minimum integer is -5. Therefore, the size of the A should be

$$(-5) - (-1) + 1 = 5$$

Array A:

Index	0	1	2	3	4
Number	0	0	0	0	0

Another factor to concern about is how to determine the corresponding index on array A for every integer in set S. Since we divided the array A into two parts when created.

min	negative part	positive part	max

We noticed that the minimum integer in the set S will be insert into array A on the first position where the index is 0. Therefore, the index for the minimum is $minimum - minimum$. Therefore, for the integer that next to the minimum which is $minimum + 1$ will be

$$(minimum - minimum) + 1 = (\text{minimum} + 1) - \text{minimum}$$

Therefore, the index for n^{th} integer will be

$$(minimum + n) - minimum$$

Therefore, for the integer num which is greater than minimum n: $num = minimum + n$, the index will be

$$num - minimum$$

Example 1:

Set S:

1	3	3	5	2

In this case, min is 1, therefore, for every integer in the set S, $S[i]$, the corresponding index in the array A will be $S[i] - 1$.

Representing integer in the set S		**1**	**2**	**3**	4	**5**
Index		0	1	2	3	4
Number		0	0	0	0	0

Example 2:

Set S:

-1	3	3	1	-2

In this case, min is -2. Therefore, for every integer in the set S, $S[i]$, the corresponding index in the array A will be $S[i] - (-2) = S[i] + 2$.

Representing integer in the set S	**-2**	**-1**	0	**1**	2	**3**
Index	0	1	2	3	4	5
Number	0	0	0	0	0	0

After considering those two factors, we are now able to implement the efficient algorithm to solve the problem. Like what Fig.8 describes, first find out the minimum number in the set S: min , and the maximum number in the set S: max. Then create an array A with size $max - min + 1$, then initialized A with 0 on every position. For every integer in the set S: $S[i]$ Since we know the minimum integer and maximum integer in the set S, if $x - S[i]$ is greater than max or smaller than min, then that is not necessary to record $S[i]$ on A, because $x - S[i]$ is not in the set S absolutely. For other cases check if $A[x - S[i]]$ is equals to 0. If yes, record the current integer on A by using $A[S[i] - min]$++, then continue to check next integer in the set S.

If $A[x - S[i]]$ is not 0, that means $x - S[i]$ exists in the set S. in another word, there is another integer S[j] in set S: $S[j]$ to make $S[i] + S[j] = x$, therefore, return true. If we reach the last integer of set S and still not able to satisfy the condition to return true, that means it does not exist the pair whose sum is x in the set S, therefore, return false.

4.1 Efficient Algorithm Test

Test Case 1:

Set S:

1	3	3	-2	2	5

x: 6
Min: -2
Max: 5
Size of array A: 5 - (-2) + 1= 8

Array A:

Index	0	1	2	3	4	5	6	7
Number	0	0	0	0	0	0	0	0

iteration 1
Current Integer S[i]: 1
x- S[i] = 6 – 1 = 5
Corresponding Index: S[i] - min = 1 – (-2) = 3
Corresponding Index of x- S[i] = x – S[i] – min = 5 -(-2) = 7
If A [7] = = 0: yes, then A [3] ++
Array A:

Index	0	1	2	3	4	5	6	7
Number	0	0	0	1	0	0	0	0

iteration 2
Current Integer S[i]: 3
x- S[i] = 6 – 3 = 3
Corresponding Index: S[i] - min = 3 – (-2) = 5
Corresponding Index of x- S[i] = x – S[i] – min = 3 -(-2) = 5
If A [5] = = 0: yes, then A [5] ++
Array A:

Index	0	1	2	3	4	5	6	7
Number	0	0	0	1	0	1	0	0

iteration 3
Current Integer S[i]: 3
x- S[i] = 6 – 3 = 3
Corresponding Index: S[i] - min = 3 – (-2) = 5
Corresponding Index of x- S[i] = x – S[i] – min = 3 -(-2) = 5

If A [5] = = 0: no, return true.
Array A:

Index	0	1	2	3	4	5	6	7
Number	0	0	1	0	0	1	0	0

Test Case 2:

Set S:

-1	-2	-1	0	-3	-4

x: -6
Min: -4
Max: 0
Size of array A: 0 - (-4) + 1 = 5

Array A:

Index	0	1	2	3	4
Number	0	0	0	0	0

iteration 1
Current Integer S[i]: -1
x- S[i] = (-6) – (-1) = -5
because **x - S[i]** is already smaller than the Min which is -4, therefore, continue to check next integer and array A will not be changed.
Array A:

Index	0	1	2	3	4
Number	0	0	0	0	0

iteration 2
Current Integer S[i]: -2
x- S[i] = (-6) – (-2) = -4
Corresponding Index: S[i] - min = -2 – (-4) = 2
Corresponding Index of x- S[i] = x – S[i] – min = (-4) -(-4) = 0
If A [0] = = 0: yes, then A [2] ++
Array A:

Index	0	1	2	3	4
Number	0	0	1	0	0

iteration 3
Current Integer: -1
x- S[i] = (-6) – (-1) = -5
because **x - S[i]** is already smaller than the Min which is -4, therefore, continue to check next integer and array A will not be changed.
Array A:

Index	0	1	2	3	4
Number	0	0	1	0	0

iteration 4
Current Integer: 0
x- S[i] = (-6) – 0 = -6
because **x - S[i]** is already smaller than the Min which is -4, therefore, continue to check next integer and array A will not be changed.
Array A:

Index	0	1	2	3	4
Number	0	0	1	0	0

iteration 5
Current Integer: -3
x- S[i] = (-6) – (-3) = -3
Corresponding Index: S[i] - min = (-3) – (-4) = 1
Corresponding Index of x- S[i] = x – S[i] – min = (-3) -(-4) = 1
If A [1] = = 0: yes, then A [1] ++
Array A:

Index	0	1	2	3	4
Number	0	1	1	0	0

iteration 6
Current Integer: -4
x- S[i] = (-6) – (-4) = -2
Corresponding Index: S[i] - min = (-2) – (-4) = 2
Corresponding Index of x- S[i] = x – S[i] – min = (-2) -(-4) = 2
If A [2] = = 0: no, then return true.
Array A:

Index	0	1	2	3	4
Number	0	1	1	0	0

4.2 Brute Force Algorithm and Analysis

For **Algorithm 1,** the input will be an array that with size of n, and an integer x. The goal is to find whether there exists a pair of number whose sum is exactly x in the array. The idea of **Algorithm 1** is to try all possible pairs in the array using two nested for loop, if the sum of the current pair is exactly x then return true immediately, otherwise continue checking next pair. If finishing travelling through the whole array, it means not able to find the pair that whose sum is x, therefore return false. Due to **Algorithm 1** using double nested for loop to check pairs, the worst case of the time complexity will consider to be $\Theta(n\text{^}2)$, that is when the pair whose sum is x is located at the end of the array. The time complexity for the best case is considered to be $\Theta(1)$ which is in the case that the pair whose sum is x is the first pair in the array, For the general case, the pair is right on somewhere between the begin and the end of the array. Therefore, the time complexity for the general case should be $O(n\text{^}2)$.

Algorithm 1: Two-Sum

```
1 Two-Sum (S, x)
2    for i = 1 to S.len – 1 do
3         for j = i+1 to S.len do
4              if S[i]+S[j] == x then
5                   return true
6  return false
```

4.3 Efficient Algorithm and Analysis

For **Algorithm 2**

Algorithm 2: Efficient Algorithm

```
1  Two-Sum-Efficient(S,x)
2    if S.len < 2 then
3         return false
4    else
5       if S.size mod 2 ! = 0 then
6          let max = S[1], min = S[1], i = 1
7       else
8          let i = 2
9          if S[1] > S[2] then
10            let max = S[1], min = S[2]
11         else
12            let max = =S[2], min = S[1]
13      for i to S.len – 1 do
14         if S[i] > S[i+1] then
15            if S[i] > max then
16               max = S[i]
17            if S[i+1] < min then
18               min = S[i]
19         else
20            if S[i+1] > max then
21               max = S[i+1]
22            if S[i] < min then
23               min = S[i]
24          i = i + 2
25      let size = max – min + 1
26      let  A[1...size] be a new array
27      for i = 1 to size do
28         let y = x – A[i]
29         if y > max or y < min then
30            continue
31         else if A[y – min] == 0 then
32            A[i-min] = A[i-min] + 1
33         else
34            return true
35   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

就是写学到了啥还有感谢各个组员们的帮助

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,

Figure8 (Go Back)