CSCI 115 Term Project Report

Dr. Matin Pirouz

Fall 2020

Group 3:

Gurleen Kaur

Yii Choong Kaw

Long Chang

Samuel Vannarith Om Mercado

Yet Chun Fong

Vi Nguyen

Part 1

1. **Theoretical question**
2. **Insertion Sort**

The time complexity for best case is O(n), when the vector is already sorted because the program will run through the whole vector from the 1 element to the nth element to make comparison between a[i] and a[j] (n comparisons, O(n)), only one comparison is done and it does not move at any stage and will not get into the inner while loop to do the swap and decrement i (O(1)). The worst case and average case time complexity of insertion sort is O(n2). The worst case occurs when the vector is sorted reversely because the program will have to run the outer for loop n times and inner while loops n-1 times to swap and decrement i, the elements are moved and compared in all the stages. For average case, the inputs are randomly sorted. The program will run through the whole for loop and sometimes get into the inner while loop to do the swap. Insertion is an in-place sorting algorithm.

1. **Selection Sort**

Selection Sort being an in-place comparison sorting algorithm, it sorts the input by comparing one element with the other to search for minimum, then swap it with the leftmost element. The time complexity for selection sort is O(N2) for its best, average, and worst case because the selection sort algorithm performs the same number of comparisons under all conditions. Selection sort algorithm divides the list into left and right parts which left is sorted and right is unsorted. The algorithm will iterate through the list to find the smallest element and swap with the leftmost element and that element will be on the left side also known as  the sorted part. This process will continue until the unsorted part has only 1 element left, which is the largest element in the list. Therefore, the number of comparisons is always the same regardless of the order of inputs. Selection sort does not take up extra memory because it is an in-place sorting algorithm therefore it is memory efficient but not time efficient.

1. **Bubble Sort**

Best case: Theoretical time: O(n). We can see this when the list is already sorted. No need of any swaps. Only the loop of comparisons is really being used.

Average case: O(nlogn). This is the most common case, when the members of the list are randomly sorted. For this case, it is for sure that a for loop is going to pass over each member. However, the use of the second loop might vary depending on how the elements are accommodated originally.

Worst case: Theoretical: O(n^2). We can see this when the members of the array are reversely sorted. Bubble sort has to do a comparison and a swap for each member. In other words, Bubble sort is going to make use of the two loops yes or yes for each key number.

1. **Merge Sort**

Best case: O(nlogn), merge Sort recursively splits the list into two halves until there is just one element and it can’t split anymore and no break.

Worst case: O(nlogn), merge Sort recursively splits the list into two halves until there is just one element and it can’t split anymore and no break.

Average case: O(nlogn), merge Sort recursively splits the list into two halves until there is just one element and it can’t split anymore and no break.

1. **Quick Sort**

The best and average case time complexity for quicksort is O(n log n). The worst case time complexity for quicksort is O(*n*2).  The worst case occurs when the pivot point is the largest or smallest element from the array. In this case, one of the arrays after the partition is always empty and the second array contains size-1 elements. Worst case occurs because quicksort is recursively called on one of the arrays. The best case occurs when the pivot is the middle element from the array. The best case is when both of the arrays have somewhat the same amount of the elements after partition. Quicksort is an in-place sorting algorithm.

1. **Heap Sort**

Heap sort is an in-place sorting algorithm, but it is not stable. The heap sort algorithm guarantees a running time of O (nlogn). Asymptotically, there is no difference between best, average, and worst-case scenarios for heap sort algorithms; it performs equally well under all scenarios. The basic idea for heap sort algorithm is that it will start by heapify function, then the largest element with push to the root node. After that, swap the last node with the root node which will take (log n) time because the root is being sunk all the way down to the leaf, and heapify the heap recursively for (n) times. The process results in running time of O(nlogn) in total. However, heap sort is notably slower than quicksort on average in most of the cases because the heap sort algorithm has more constant factors than quicksort such as number of swaps. The main reason why heap sort performs equally under any scenarios is because the heap sort algorithm swaps the data and puts them in order, even if the data is already sorted. In short, heap sort does not have best, average, worst scenarios like other sorting algorithms because the steps to sort in heap sort are not reduced on the order of the inputs.

1. **Counting Sort**

Counting sort is a sorting algorithm that sorts the inputs by counting the occurrence of each unique input and stores the count in an auxiliary array. Counting sort is stable but is not in place sort. To sort the input, it uses the count as the index of the auxiliary array to store the input. Assume the n input elements are integer in the range from 0 to n-1, the best, worst or average case for counting sort are O(n + r). The ‘n’ is for the original array while ‘r’ is for the auxiliary array. There are 4 for loops in counting sort. Whenever it needs to iterate until ‘n’, the time complexity for that For loop is O(n) and the same goes to ‘r’, whenever it iterates until ‘r’, the time complexity is O(r). So, all those up will become the time complexity for counting sort. Pseudocodes for each for loop and the function of each for them are shown below:

**for i ← 0 to r**

**do C[i] ← 0**

The first for loop is to set all the elements in the auxiliary array to be 0. As it iterates to ‘r’, the time complexity for this loop is O(r).

**for j ← 1 to n**

**do C[A[j]] ← C[A[j]] + 1**

The second for loop is to count the frequency of each element of the original array and store it to their respective index in the auxiliary array. The loop iterates until ‘n’, so the time complexity for this loop is O(n).

**for i ← 1 to r**

**do C[ i ] ← C[ i ] + C[i - 1]**

The third for loop is to get the cumulative sum of the element in the auxiliary array as the cumulative sum is the key part to place the elements to the correct index to sort the array. Similar to the first for loop, the time complexity for this for loop is O(r).

**for j ← n down to 1**

**do B[C[A[ j ]] - 1] ← A[ j ]**

As for the last loop, there is an array B[] which is the output array to store the elements and ultimately be the sorted array. First, get the index from the original array, then the index in the auxiliary array provides the value which is the cumulative count. Minus the value by 1 and that is the index where the element should be in the output array. This loop iterates from the last element to the first element from the original array, so the time complexity is O(n).

Get the time complexity of each for loop and add them together to get the total time complexity for counting sort which is O(n + r). This sorting algorithm may have an issue. Assume the n input elements are integer in the range from 0 to n-1 and there is an element with 6 digits number while the rest of the elements have a maximum 3 digits number. The time complexity for it does not change but the time to create the auxiliary number drastically increases. So, it is not efficient when handling this kind of case.

1. **Radix Sort**

Radix sort not being a comparison, but a linear sorting algorithm gives itself the time complexity of O(n), or more specifically O(d\*(n+k)) for the best, worse, and average case. D is the highest number of digits the largest element has, n is the size of the array, and k is the maximum possible value of a digit of the elements. K will not surpass 9 since the value of a digit can only go up to 9. The idea of radix sort is start sorting from the least significant digit to the most significant digits, it uses counting sort as the subroutine to sort the inputs. Radix sort itself takes advantage of other sorting algorithms and uses them to sort the digits of within the elements one by one instead of the whole element at once.

**Best Case:** All elements are 1 digit, meaning Radix sort only has to sort 1 time.

**Worst case:** An element has N2 digits, meaning Radix sort has to go and sort N2 time.

1. **Data generation and experimental setup**
2. **Insertion Sort**

* What kind of machine did you use?
* Dell, Intel ®️ CORE (™️) i5-10210U CPU @1.60 GHz 2.11 GHz
* What timing mechanism:
* chrono.
* How many times did you repeat each experiment?
* 3
* What times are reported:

Insertion: size = 50000

* Best case time: 0.05 milliseconds
* Worst case time: 95.475 milliseconds
* Average case time: 48.8886 milliseconds
* How did you select the inputs?
* For insertion sort, for the best case, I assigned the values in order and already sorted them into the array. For the worst case, I reversed the values of the array and stored them in reverse order. For the average case, I randomized the array.
* Did you use the same inputs for all sorting algorithms?
* No

1. **Selection Sort**

* What kind of machine did you use?
* Intel ® Core ™ i7-8565U, 16 GB RAM.
* What timing mechanism did you use?
* Chrono (Milliseconds)
* How many times did you repeat each experiment?
* 3
* What times are reported?
* Random: 64.731ms
* Pre-Sorted: 85.198 ms
* Reversed: 110.307 ms
* How did you select the input?
* Vector size 50000 (Orders: Random, pre-sorted, reversed)
* Did you use the same inputs for all sorting algorithms?
* No

1. **Bubble Sort**

* What kind of machine did you use?
* Processor: AMD Ryzen 7 2700X Eight-Core Processor 3.70 GHz
* RAM: 16 GB
* What timing mechanism?
* We decided to use chrono since it is very simple to measure time in milliseconds, which is convenient to see differences between performances.
* How many times did you repeat each experiment?
* 3 times for this algorithm code (randomly sorted, sorted, and reversed sorted vector).
* What times are reported?
* Average case: 1803194 ms.
* Best case: 353 ms.
* Worst case: 2570890 ms.
* How did you select the inputs?
* Vector with a size of 5,000. Random generated numbers in the vector. However, already sorted numbers for best case, and reversed sorted numbers for worst case.
* Did you use the same inputs for all sorting algorithms?
* Since we used a random generation of numbers as our input, the input was not the same. Plus, the size assassinated for each algorithm might vary depending on the running time.

1. **Merge Sort**

* What kind of machine did you use?
* Dell, Intel ® CORE (™) i5-10210U CPU @1.60 GHz   2.11 GHz
* What timing mechanism?
* Chrono
* How many times did you repeat each experiment?
* 3
* What times are reported?
* Merge: size = 1,000,000
  + - Best Case: 7467.99ms
    - Worse Case: 6549.21ms
    - Average Case: 6931.13ms
* How did you select the inputs?
* I selected the inputs according to the advantages and disadvantages of the algorithms.
* Did you use the same inputs for all sorting algorithms?
* No, I didn’t

1. **Quick Sort**

* What kind of machine did you use?
* Memory: 8 GB 2133 MHz LPDDR3
* Processor: 3.1 GHz Dual-Core Intel Core i5
* What timing mechanism?
* chrono
* How many times did you repeat each experiment?
* 5
* What times are reported?
* Input size = 100,000

Quick sort (First element as Pivot)

* Best case: 0.41015 milliseconds
* Worst case: 10.1908 milliseconds
* Average case: 10.1908 milliseconds

Quick sort (Random element as Pivot)

* Best case: 0.109114 milliseconds
* Worst case: 0.109114 milliseconds
* Average case: 0.109114 milliseconds

Quick sort (Median element as Pivot)

* Best case: 0.106594 milliseconds
* Worst case: 4.82168 milliseconds
* Average case: 0.106594 milliseconds
* How did you select the inputs?
* For quicksort, we know that if we use the largest element as pivot, we get the worst case so for quicksort, I picked 1 as the pivot point. For the best case, I just used the random array and the first element as the pivot point.
* Did you use the same inputs for all sorting algorithms?
* No

1. **Heap Sort**

* What kind of machine did you use?
* Intel ® Core ™ i7-8565U, 16 GB RAM.
* What timing mechanism?
* Chrono (Milliseconds)
* How many times did you repeat each experiment?
* 3 times
* What times are reported?
* Random: 113.081 ms
* Pre-Sorted: 99.305 ms
* Reversed: 111.120 ms
* How did you select the inputs?
* Vector size 10000000 (Orders: random, pre-sorted, reversed)
* Did you use the same inputs for all sorting algorithms?
* No

1. **Counting Sort**

* What kind of machine did you use?
* Intel Core i7-856 CPU @ 1.80GHz   1.99GHz
* x64-based PC
* 16GB RAM
* What timing mechanism?
* Chrono (milliseconds)
* How many times did you repeat each experiment?
* 3 times
* What times are reported?
* Quick Sort size = 100000
* Best case: 181 ms
* Worst case: 196 ms
* Average case: 183 ms

* How did you select the inputs?
* The number of inputs is 10 million. For single digits inputs, they are inserted by a for loop function to get the modulus 10 of the random generated inputs. For multiple digits inputs, they are inserted by a for loop function to randomly generated inputs.
* Did you use the same inputs for all sorting algorithms?
* No

1. **Radix Sort**

* What kind of machine did you use?
* Intel(R) Core(™️) i7-8565U CPU 1.80GHz
* x64-based PC
* 16GB RAM
* What timing mechanism?
* Chrono (milliseconds)
* How many times did you repeat each experiment?
* 3 times
* What times are reported?
* Quick Sort size = 100,000
* Best case: 4.263 ms
* Worst case: 14.453 ms
* Average case: 7.233 ms

* How did you select the inputs?
* The size of inputs is 100,000 for all, but for best case only tens place digits, worst case is all hundred million digits, and average case is all hundred thousand digits.
* Did you use the same inputs for all sorting algorithms?
* No

1. **Which of the five sorts seems to perform the best ?**

* **We picked Counting sort, Heap sort, Quick Sort (3 version), Insertion sort and Merge sort.**

Best Case (Time In milliseconds)

***Table

Description automatically generatedChart, line chart

Description automatically generated***

Worst Case (Time In milliseconds)

Table

Description automatically generated

Chart, line chart

Description automatically generated

Average Case (Time In milliseconds)

Table

Description automatically generated

Chart, line chart

Description automatically generated

1. **To what extent does the best, worst-se analyses of each sort agree with the experimental results?**

* **Insertion Sort**

Table

Description automatically generated

Based on the graph for insertion sort, the experimental results don’t match with its predicted theoretical times (Dotted trendline), because the running time for sorting algorithm might be highly affected by the external factors including the hardware environment such as CPU speed, amount of running processes at that particular time which caused the actual experimental results to be different with predicted theoretical time. However, the patterns of running time for best, worst, average case agree with the asymptotic analysis which insertion sort has time complexity of O(n) for its best case and time complexity of O(n2) for its average and worst case which matched with the pattern on the graph.

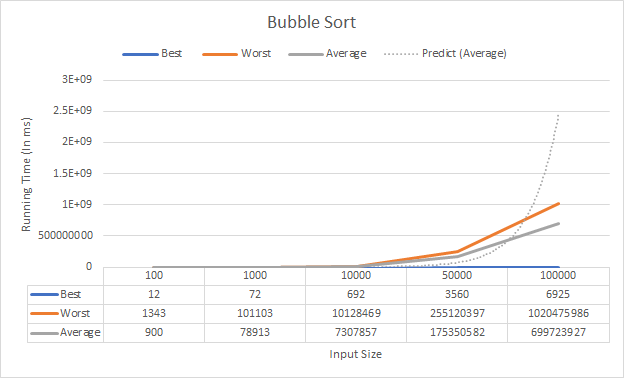
* **Selection Sort**

Table

Description automatically generated

According to the graph for selection sort, the experimental result doesn’t match with the predicted theoretical time as the pattern trendline doesn’t match with the lines graphed with actual experimental results. One of the possible reasons why the actual results don’t align with the predicted theoretical times might be the external factors including hardware specifications such as CPU speed, usage of RAM at that time, SSD or HDD and more which will directly affect the running time of the algorithm. However, the selection sort has time complexity of O(n2) for its best, average and worst case, and the experimental results matched with the asymptotic analysis for its best, average and worst case because the pattern of lines grows similarly for all three cases even though the lines do not match with the predicted theoretical times which is indicated by the dotted trendline.

* **Bubble Sort**



As we can see in the graph, the numbers do not fit exactly as dictated by the theory of time complexity. Although more than one factor may be causing this result, the main factors are still the components that make up the equipment used. That is, the processor, RAM, internal storage, etc. Let's remember the internal components are ultimately the ones in charge of making the possible calculations. As a result, we will have variations depending on the individual characteristics of each component of the equipment. Even so, we can observe the characteristic curvature that a function with n^2 has, which is the average time complexity of Bubble Sort O(n^2).

* **Merge Sort**

**Table

Description automatically generated**

As we learn from the lecture, the Merge Sort algorithm running time is NlogN for every case including best case, worst case and average case, because there is no break in the loop and it recursively splits the parent lists into half until there is just one element in the list and merge them back together. As in the graph, we see that the running time between cases are not much different from each other, especially for sorted input and reversed input. It shows that the three lines are on the same path and almost on the same path line. It proves that the experiments are similar to that the lecture has stated and proved. However, in the graph, the random input list takes a little bit more time than the other two cases when the input size increases. Because the difference is not much, we think it's just because of the running system. The running time changes every time we run the program and the running time also depends on other factors including systems.

* **Quick Sort**

Table

Description automatically generated

According to the graph above for quicksort, the graph does not match the predicted theoretical results. Our experimental values are much lower than the theoretical values. At first the values do match with the theoretical values but as the input size increases the theoretical results grow much higher and faster than the experimental  result. We can see that our experimental results also do start growing at a lower speed after a point. One the reasons why the results don’t match could be the computer speed and the type of hardware we are using. So the external factors can be the reason for why our experimental and theoretical results don’t match. For quicksort the worst case is n^2 and n^2 is presented by the trendline (dotted line) in the graph and the average case and best case is O(nlogn).

* **Heap Sort**

**Table

Description automatically generated**

Based on the graph above, the experimental results do not match with the predicted theoretical results, as the predicted theoretical times are much lower than the actual experimental results. However, the pattern of best, average, worst case agrees with the asymptotic analysis which heap sort algorithm has the same time complexity for all of its cases, O(n log n). According to the predicted theoretical times (trendline), the running time should grow much slower, but the actual results have a higher running time. The reason behind this might be that the external factors like hardware and software performance of the PC including CPU speed, processors used in the PC, usage of RAM at the particular time which will directly affect the running time and caused the pattern of predicted theoretical time and actual experimental results doesn’t match.

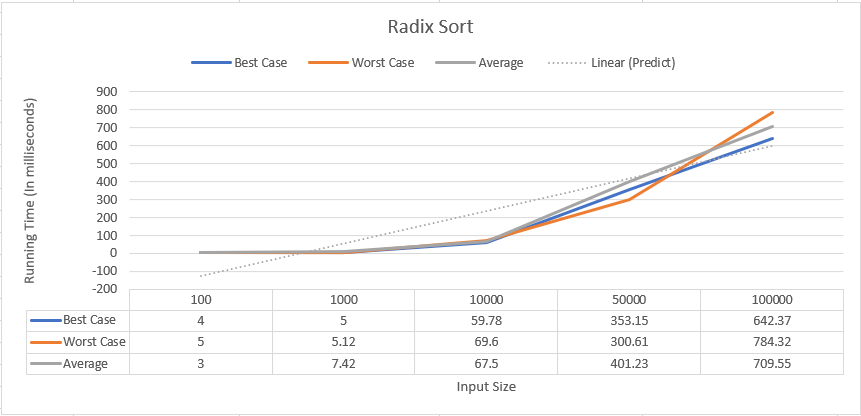
* **Counting Sort**

**Table

Description automatically generated**

According to the Counting Sort graph above, the predicted theoretical results which is the equation of the dotted trendline do not really match the experimental results. The possible reason why this is happening is because it is affected by external factors such as computer hardware which include the CPU speed, the RAM, SSD and so on. This computer hardware will affect the running time directly which cause the experimental results either running faster or slower.

* **Radix Sort**

****

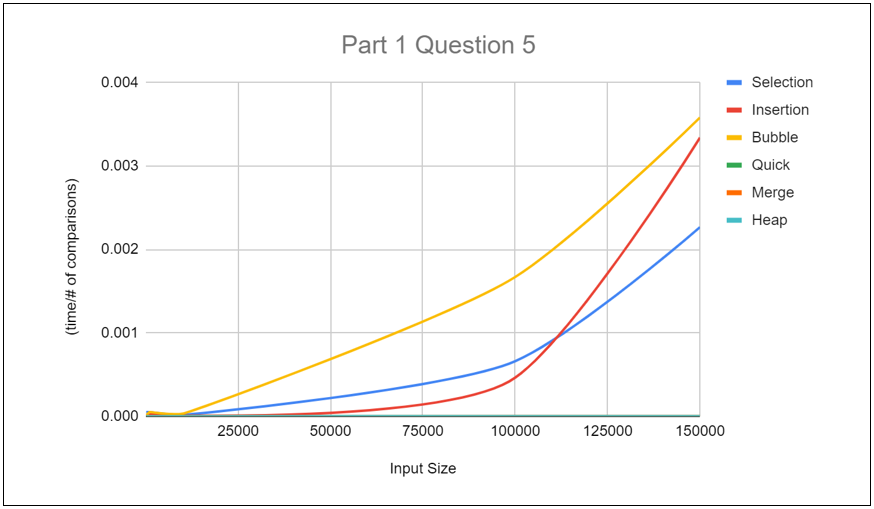
The linear line for the graph seems to be going in a constant line. This predicts the time of radix sort being O(d\*(n+k)) amount of time, as the graph is a linear graph. The graph also supports radix sort having the same time complexity for best worst and average case.

Summary for Part 1 Question 4

When analyzing theoretical vs experimental, theoretical analysis ignores a lot of factors and focuses on a few main things for the time analysis. For comparison sorts the only thing really being measured is the amount of comparisons being made, and factors such as initializations of data and updating them are ignored. When running an algorithm on a computer and measuring the time, the time still will not exactly be the same even if the array size, elements, and algorithm does not change. This is because there are other factors such as CPU usage, computer specs, and a lot of little factors affecting the performance of the runtime.

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Time/Number of Comparisons | | | | | | |
| Input Size | Selection | Insertion | Bubble | Quick | Merge | Heap |
| 100 | 5.00E-05 | 5.00E-06 | 1.00E-05 | 5.50E-07 | 2.00E-06 | 0 |
| 1000 | 4.96E-05 | 2.43E-05 | 4.97E-05 | 3.40E-08 | 1.45E-06 | 3.33E-07 |
| 10000 | 1.76E-05 | 6.73E-06 | 3.18E-05 | 4.74E-09 | 1.18E-07 | 5.60E-07 |
| 100000 | 6.60E-04 | 4.63E-04 | 1.67E-03 | 2.13E-07 | 1.05E-06 | 7.86E-07 |
| 150000 | 2.27E-03 | 3.34E-03 | 3.58E-03 | 3.44E-07 | 1.15E-06 | 6.62E-07 |



According to the graph, the number of comparisons has predicted the execution time accurately as the non-efficient sorting algorithms including selection sort, insertion sort, and bubble sort have higher running time compared to the efficient sorting algorithms (Quick sort, Merge sort, Heap sort).

Some external factors like the performance of PC, usage of RAM, might affect the running time for different algorithms. But the graph clearly showed the differences between each sorting algorithm. Selection, insertion and bubble sorting algorithms work like the brute force approach in which they have to go through the inputs again and again to sort the inputs. Thus, the number of comparisons for these sorting algorithms would be relatively higher and caused a higher running time for the same input size compared to quick, merge, and heap sorting algorithms. Quick, merge, heap sorting algorithms are much efficient because they divide the problem into smaller subproblems after each pass which will lower the number of comparisons and reduce their execution times. Therefore, the running time for the efficient sorting algorithms don’t grow exponentially when the input size increases.

Part 2

Pseudocode for brute force and efficient algorithm

Bruteforce (vector, target)

n <- vector size

for (i<-0 to n)

for (j<-i+1 to n)

sum = vector[i] + vector[i]

if(sum == target)

return true

return false

Efficient (vector<int> vec, target)

create hash table that takes integer type key and satellite data

n <- vector size

for (i <- 0 to n)

difference = target - vector[i]

if (difference is in hash table)

return true

else

store the value of vector[i] into hash table

return false

Running Time analysis

1. Brute Force Solution
   * The brute force approach to this problem is relatively simple. Loop through each element of index i and another nested for loop with index j that start from i+1, which will compared every element in the vector to find if there is sum of any pairs that matched with the target. If there is one pair of elements that matched with the target, return true. However, if the for loop i has looped through the vector and do not find any elements that sum to the target, return false. The time complexity of this brute force solution is O(n^2). For each element, we try to find its complement by looping through the rest of the vector which will takes O(n) time. Therefore, the inner for loop will have to loop through the vector again and again which gives us the time complexity of O(n^2).
2. Efficient solution
   * The approach for efficient solution is using hash table with direct addressing. First, a hash table is being created that takes integer for key values and satellite data. First, a for loop will be used to loop through the vector, then we will find the difference of target and the element of vector at index i and store it in variable diff.

After that, we will use the built-in function for hash table, Count to check if the value of difference is in the Hash Table we created. If the value of diff exists in the Hash Table, return true. Else, the value of vector at index i will be store into Hash table with the index as its satellite data. If we finished the for loop, which indicates that we have check through the whole vector but there is no pair that matched the target, then we will return false. The time complexity for this algorithm is O(n) because it only traverse the vector that containing n elements once, then each look up at the Hash Table to check the existence of diff only takes O(1) time. Therefore, the time complexity for this algorithm is only O(n).