



ULAB

**UNIVERSITY OF LIBERAL ARTS
BANGLADESH**

Lab Report – 1

Course Name: Algorithms Lab

Course Code: CSE 2202

Section: 1

Submitted by

Salvir Rahman Ratul

223014074

Problem Statement: Modify the existing Quick Sort implementation to count the number of comparisons made during the sorting process. Compare the number of comparisons for different input sizes n and analyze the relationship between the input size and the number of comparisons.

Instructions:

1. Modify the quickSort function to keep track of the number of comparisons made during the sorting process.
2. Initialize a counter variable in the main function to keep track of the number of comparisons.
3. Increment the counter variable each time a comparison is made in the quickSort function.
4. After sorting each array, print the number of comparisons made.
5. Repeat the experiment for different input sizes n and analyze the relationship between the input size and the number of comparisons.

On this report, I will provide the answers of all given questions. I will provide all the needed document, graph and code for the given problems

CONTENTS:

1. Introduction

Page 4-9

2. Answer to Question 1

Page 10

3. Answer to Question 2

Page 11

4. Answer to Question 3

Page 12 - 16

5. Answer to Question 4

6. Answer to Question 5

7. Conclusion

Page 16

8. References

Introduction: The topic of the last lab class was Quick Sort. What is quick sort? The answer is simple, It's a sorting algorithm. It is an efficient and general-purpose algorithm. Quick Sort is

- ➔ developed by Tony Hoare in 1959
- ➔ published in 1961.
- ➔ a divide-and-conquer algorithm.
- ➔ slightly faster than Merge Sort and Heap Sort for randomized data, particularly on bigger distributions.
- ➔ works by selecting a 'pivot' element from the array and partitioning the other elements into two subarrays, according to whether they are less than or greater than the pivot.
- ➔ also called by Partition-Exchange Sort,
- ➔ on average, the algorithm takes $O(n \log n)$ comparisons to sort n items. In the worst case, it makes $O(n^2)$ comparisons.

The steps for in-place quicksort are:

1. If the range has fewer than two elements, return immediately as there is nothing to do. Possibly for other very short lengths a special-purpose sorting method is applied and the remainder of these steps are skipped.
2. Otherwise pick a value, called a pivot, that occurs in the range (the precise manner of choosing depends on the partition routine and can involve randomness).
3. Partition the range: reorder its elements, while all elements with values greater than the pivot come after it; elements that are equal to the pivot can go either way. Since at least one instance of the pivot is present, most partition routines ensure that the value that ends up at the point of division is equal to the pivot, and is now in its final position.

4. Recursively apply the quicksort to the sub-range up to the point of division and to the pivot at the point of division.

The algorithm of quicksort pivot:

Step 1 – Choose the highest index value that has a pivot.

Step 2 – Take two variables to points left and right of the list excluding the pivot.

Step 3 – Left points to the high.

Step 4 – Right points to the high.

Step 5 – while the value at left is less than the pivot moves right

Step 6 – while the value at the right is greater than the pivot moves left.

Step 7 – if both Step 5 and Step 6 do not match swap left and right.

Step 8 – if the left is greater than or equal to the right, the point where they met is new pivot.

The pseudocode of the Quick Sort pivot:

```
Function partitionFunc(left, right, pivot)
Leftpointer = left
Rightpointer = right - 1
While True do
While A[++leftpointer] < pivot do
//do-nothing
End while
While rightpointer > 0 && A[-- rightpointer] > pivot do
//do-nothing
End while
If leftpinter >= rightpointer
break
```

```
else
swap leftpointer, rightpointer
end if
end while
swap leftpointer, right
return leftpointer
end function
```

The Algorithm of the Quick Sort:

Step 1 – Make the right-most index value pivot.

Step 2 – partition the array using pivot value.

Step 3 – quicksort left partition recursively.

Step 4 – quicksort right partition recursively.

The pseudocode of the Quick Sort:

```
Procedure quicksort(left, right)
If right-left <= 0
return
else
pivot = A[right]
partition = partitionFunc(left, right, pivot)
quicksort(left, partition-1)
end if
end procedure
```

Code of the Quick Sort:

```
// C code to implement quicksort

#include <stdio.h>

// Function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

// Partition the array using the last element as the pivot
int partition(int arr[], int low, int high)
{
    // Choosing the pivot
    int pivot = arr[high];

    // Index of smaller element and indicates
    // the right position of pivot found so far
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than the pivot
        if (arr[j] < pivot) {
            // Increment index of smaller element
            i++;
        }
    }
}
```

```
        swap(&arr[i], &arr[j]);
    }

}

swap(&arr[i + 1], &arr[high]);
return (i + 1);
}

// The main function that implements QuickSort
// arr[] --> Array to be sorted,
// low --> Starting index,
// high --> Ending index
void quickSort(int arr[], int low, int high)
{
    if (low < high) {

        // pi is partitioning index, arr[p]
        // is now at right place
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Driver code
int main()
{
```



```
Int arr[] = {10, 7, 8, 9, 1. 5};  
Int N = sizeof(arr) / sizeof(arr[0]);  
//function call  
quicksort(arr, 0, N - 1);  
printf("Sorted array: \n");  
for(int I = 0; i<N; i++)  
printf("%d ", arr[i]);  
return 0;  
)
```

Answer to Question 1: To modify the 'quicksort' function to keep track of the number of comparisons made during the sorting process, we can introduce a counter variable and increment it whenever a comparison is made. Here is a new version of the quicksort function with the comparison count added.

```
void quickSort(int arr[], int low, int high, int* counter) {  
    if (low < high) {  
        // Get the pivot index through partition  
        int pi = partition(arr, low, high, counter);  
  
        // Recursively sort the elements before and after  
        partition  
        quickSort(arr, low, pi - 1, counter);  
        quickSort(arr, pi + 1, high, counter);  
    }  
}
```

Answer to Question 2:

```
int main() {  
    int arr[] = {10, 7, 8, 9, 1, 5};  
    int n = sizeof(arr) / sizeof(arr[0]);  
  
    int Counter = 0;  
    quickSort(arr, 0, n - 1, &Counter);  
  
    printf("Sorted array: ");  
    for (int i = 0; i < n; i++) {  
        printf("%d ", arr[i]);  
    }  
  
    printf("\nNumber of comparisons: %d\n", Counter);  
  
    return 0;  
}
```

Answers to Questions 3, 4, 5:

I took 5 arrays to test the program. These arrays are different in size.

Input arrays:

First array: 10, 7, 8, 9, 1, 5 (size = 6)

second array: 1, 17, 8, 90, 21, 15, 0, 2 (size = 8)

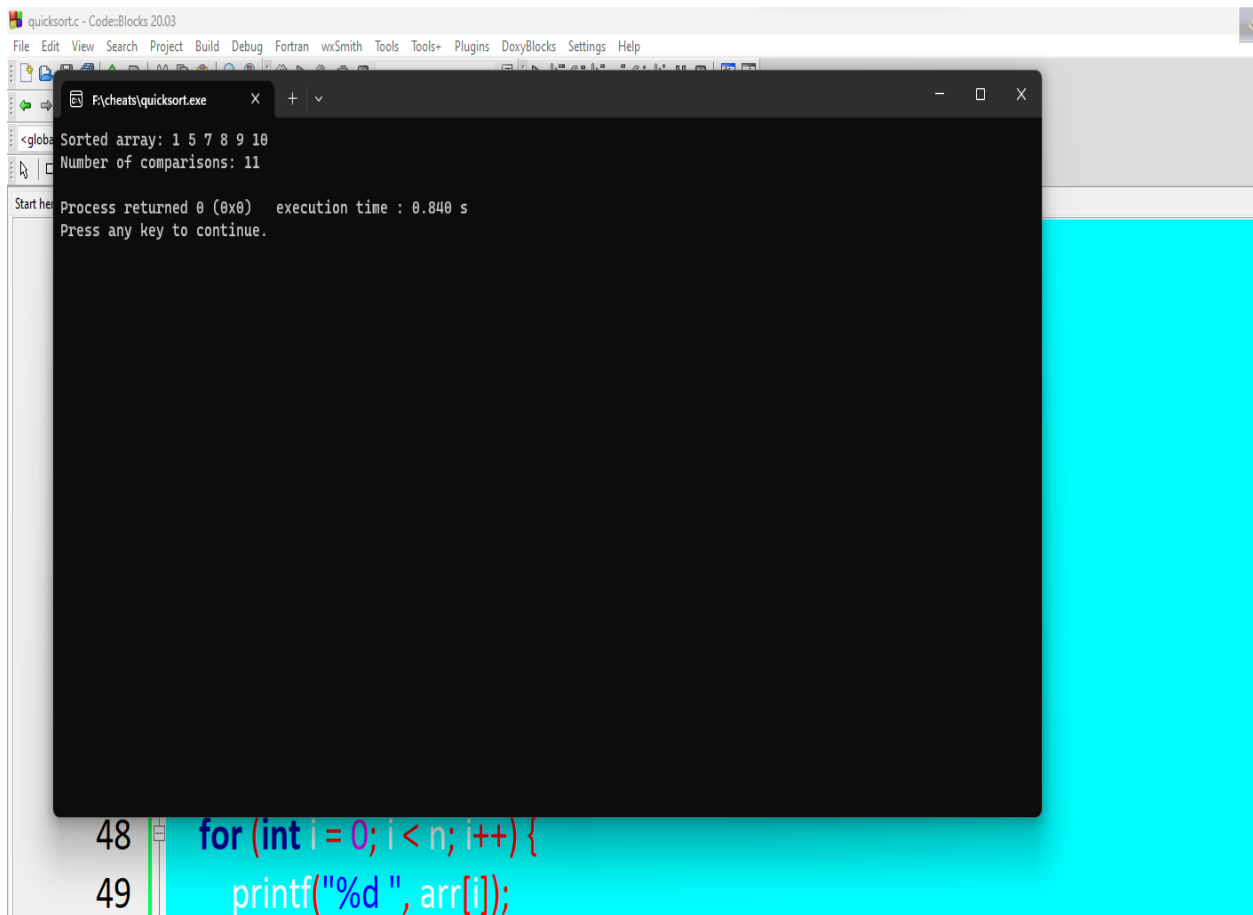
Third array: 4, 18, 94, 13, 50 (size = 5)

Fourth array: 10, 7, 8 (size = 3)

Fifth array: 10, 7, 8, 9, 1, 5, 17, 5, 4, 0 (size = 10)

Results:

For the first Case:

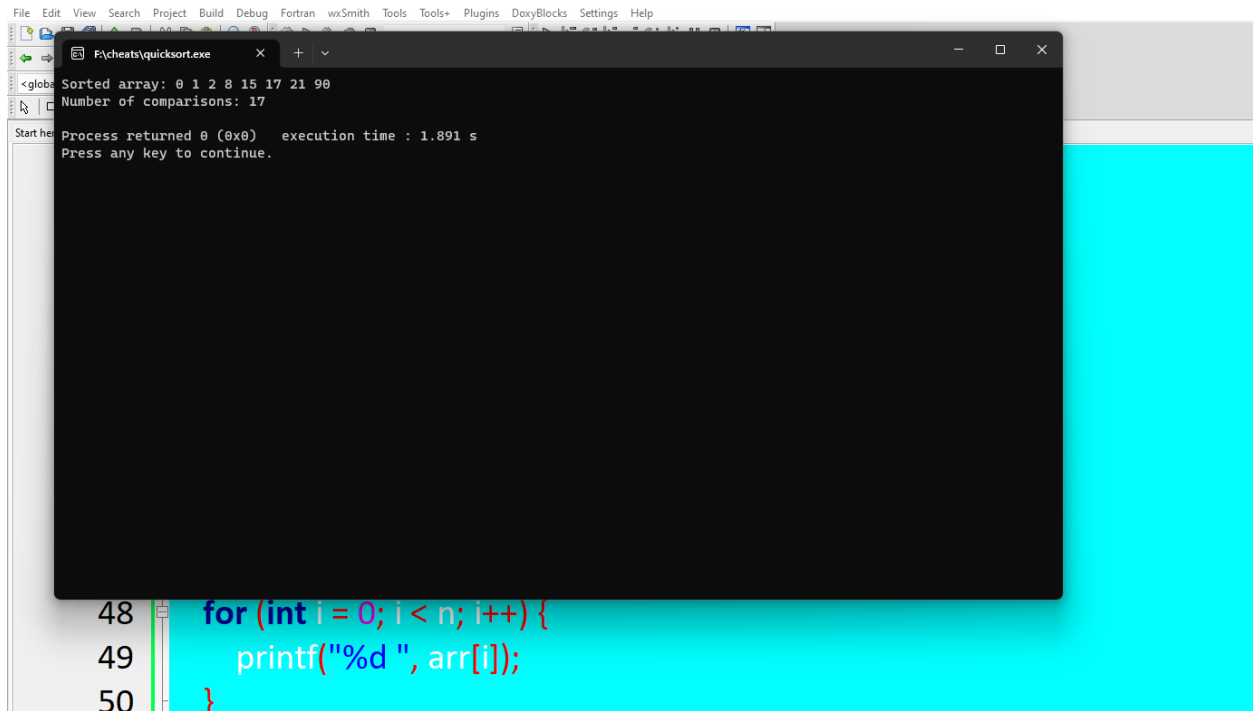


```
quicksort.c - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help

Sorted array: 1 5 7 8 9 10
Number of comparisons: 11
Process returned 0 (0x0)   execution time : 0.840 s
Press any key to continue.

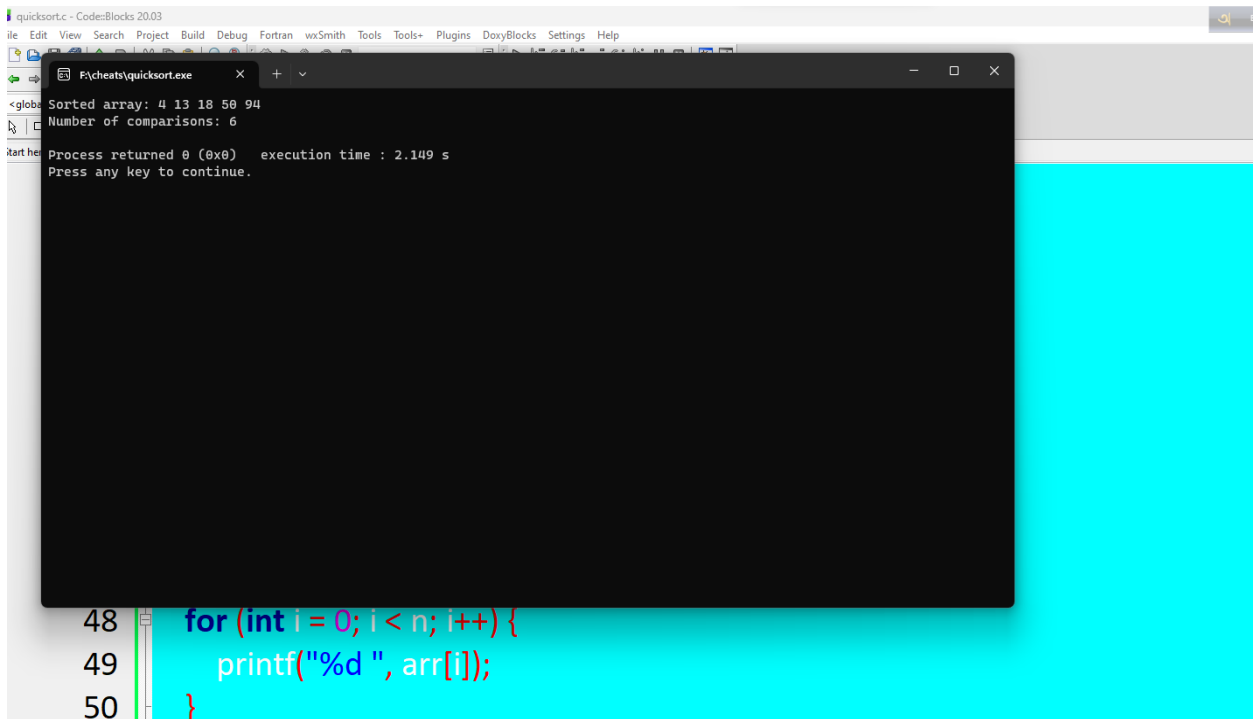
48 for (int i = 0; i < n; i++) {
49     printf("%d ", arr[i]);
```

For the second Case:



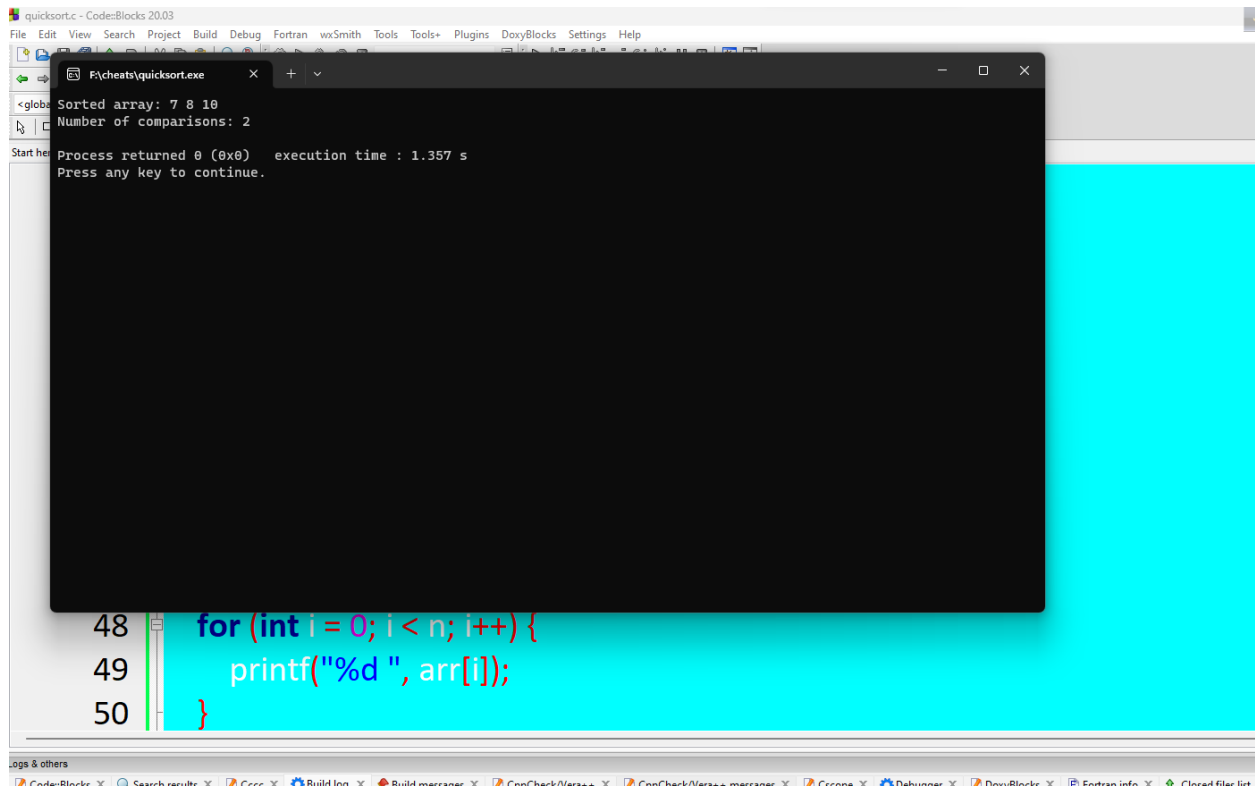
```
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
F:\cheats\quicksort.exe x + -
Sorted array: 0 1 2 8 15 17 21 90
Number of comparisons: 17
Process returned 0 (0x0) execution time : 1.891 s
Press any key to continue.
48 for (int i = 0; i < n; i++) {
49     printf("%d ", arr[i]);
50 }
```

For the third Case:



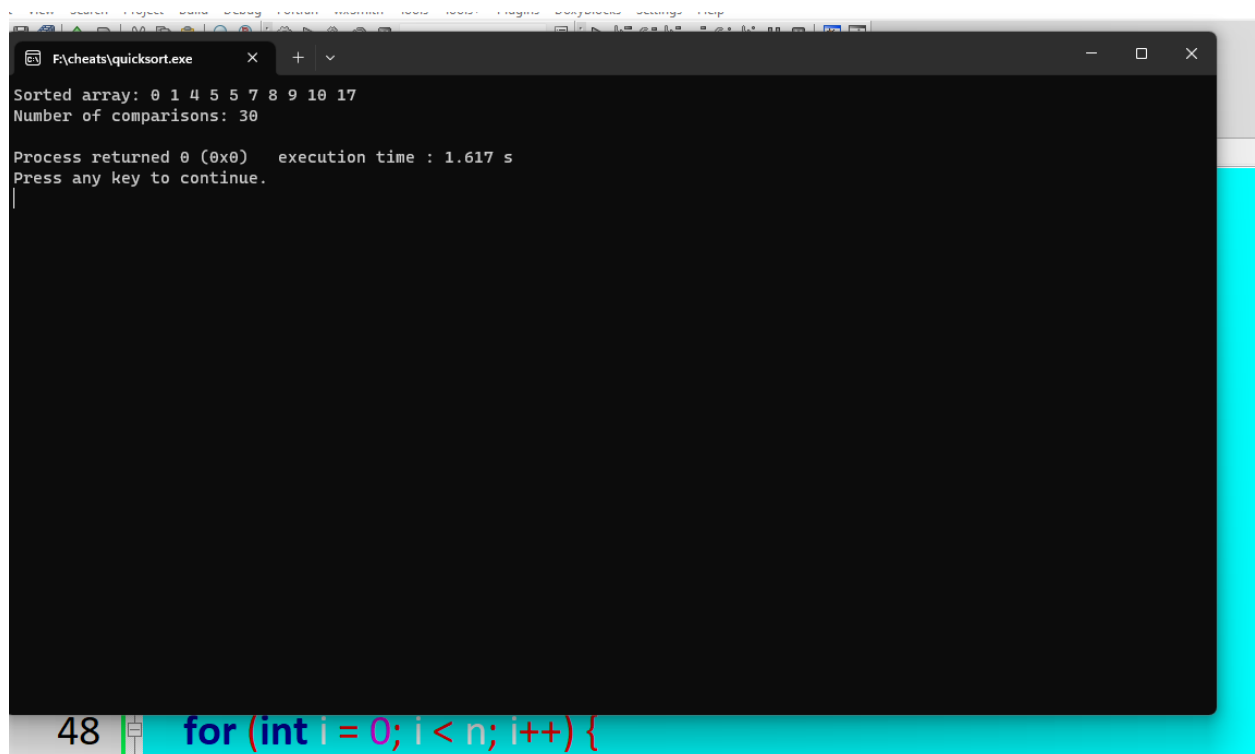
```
quicksort.c - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
F:\cheats\quicksort.exe x + -
Sorted array: 4 13 18 50 94
Number of comparisons: 6
Process returned 0 (0x0) execution time : 2.149 s
Press any key to continue.
48 for (int i = 0; i < n; i++) {
49     printf("%d ", arr[i]);
50 }
```

For the fourth Case:



```
quicksort.c - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
F:\cheats\quicksort.exe
Sorted array: 7 8 10
Number of comparisons: 2
Process returned 0 (0x0) execution time : 1.357 s
Press any key to continue.
48 for (int i = 0; i < n; i++) {
49     printf("%d ", arr[i]);
50 }
```

For the fifth Case:



```
quicksort.c - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
F:\cheats\quicksort.exe
Sorted array: 0 1 4 5 5 7 8 9 10 17
Number of comparisons: 30
Process returned 0 (0x0) execution time : 1.617 s
Press any key to continue.
48 for (int i = 0; i < n; i++) {
```

Let's see the data in tabular form,

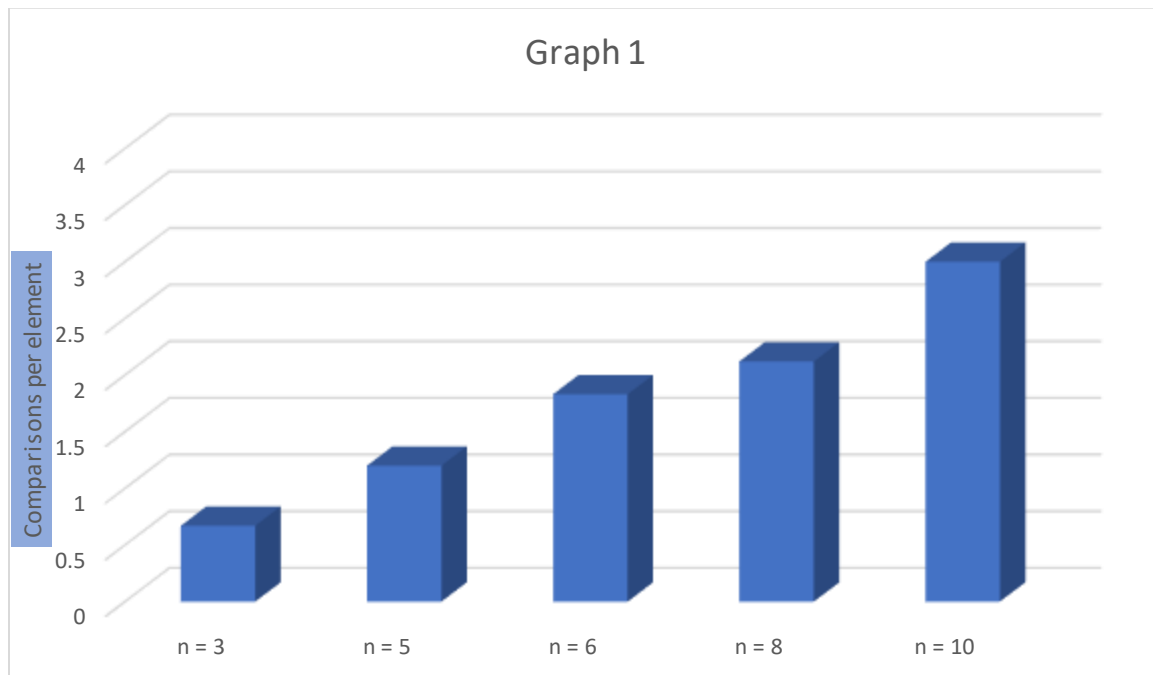
Test No	Array	Array Size	Comparison
01	10, 7, 8, 9, 1, 5	6	11
02	1, 17, 8, 90, 21, 15, 0, 2	8	17
03	4, 18, 94, 13, 50	5	6
04	10, 7, 8	3	2
05	10, 7, 8, 9, 1, 5, 17, 5, 4, 0	10	30

Analysis: For analyzing the relationship, we can calculate the average number of comparisons per element for each data point ->

- 1) For $n=6$: $11 \text{ comparisons} / 6 \text{ elements} = 1.83 \text{ comparisons per item (CPI)}$.
- 2) For $n=8$: $17 \text{ comparisons} / 8 \text{ elements} = 2.12 \text{ CPI}$.
- 3) For $n=5$: $6 \text{ comparisons} / 5 \text{ elements} = 1.20 \text{ CPI}$.

4) For $n=3$: 2 comparisons / 3 elements = 0.67 CPI.

5) For $n=10$: 30 comparisons / 10 elements = 3.00 CPI.



Conclusion: We can see that the number of comparisons per item increases as the input size (n) increases. This is consistent with the expected behavior of the quicksort algorithm where the average-case time complexity is $O(n \log n)$. As the input size increases, resulting in a higher number of comparisons per item. This shows us that the efficient sorting performance of the quicksort algorithm, especially for larger datasets.

References: 1. Introduction to Algorithms – Book by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

2. Data Structures and Algorithms Made Easy – book by Narsimha Karumanchi

3. Online resources: Wikipedia.org