Experiment No. 3
Quick Sort
Date of Performance:
Date of Submission:

Experiment No. 3

Title: Quick Sort

Aim: To implement Quick Sort and Comparative analysis for large values of ‘n’.

Objective: To introduce the methods of designing and analyzing algorithms.

Theory:

The merge sort algorithm closely follows the divide-and-conquer paradigm. Intuitively, it operates as follows:

1. Divide: Divide the n-element sequence to be sorted into two subsequences of n=2 elements each.
2. Conquer: Sort the two subsequences recursively using merge sort.
3. Combine: Merge the two sorted subsequence to produce the sorted answer.

Partition-exchange sort or quicksort algorithm was developed in 1960 by Tony Hoare. He developed the algorithm to sort the words to be translated, to make them more easily matched to an already-sorted Russian-to-English dictionary that was stored on magnetic tape.

Quick sort algorithm on average, makes O(n log n) comparisons to sort n items. In the worst case, it makes O(n2) comparisons, though this behavior is rare. Quicksort is often faster in practice than other O(n log n) algorithms. Additionally, quicksort's sequential and localized memory references work well with a cache. Quicksort is a comparison sort and, in efficient implementations, is not a stable sort. Quicksort can be implemented with an in-place partitioning algorithm, so the entire sort can be done with only O(log n) additional space used by the stack during the recursion.

Quicksort is a divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sublists.

1. Elements less than pivot element.

2. Pivot element.

3. Elements greater than pivot element.

Where pivot as middle element of large list. Let's understand through example:

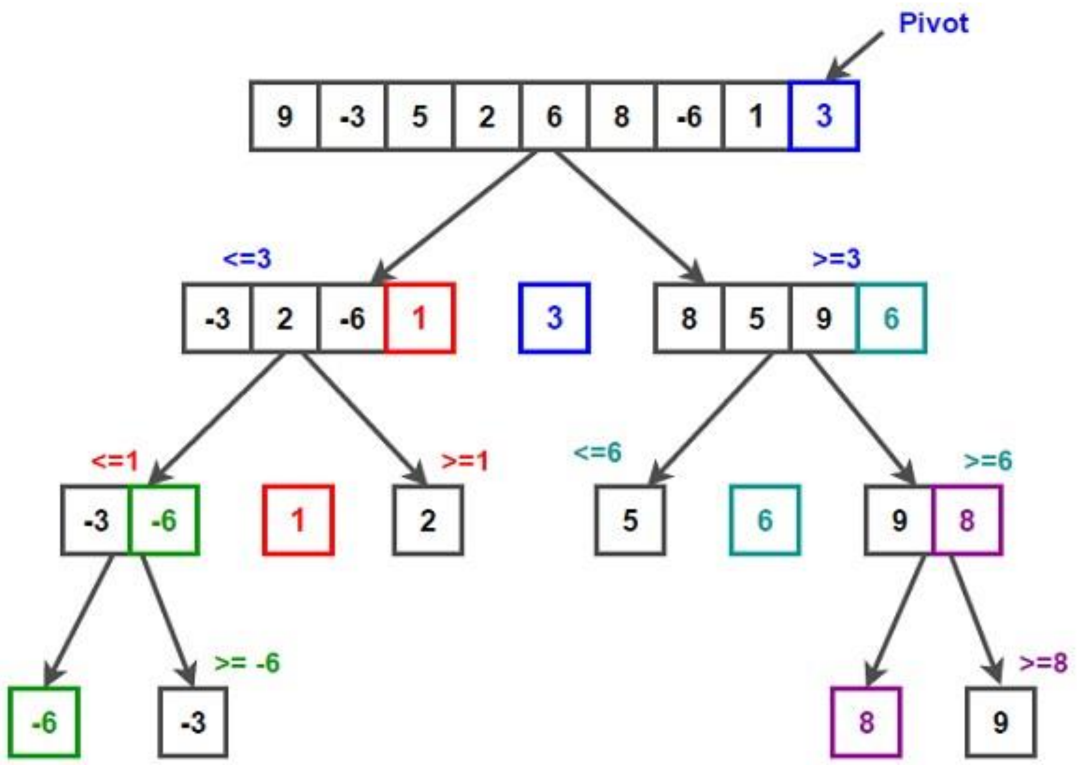
List : 3 7 8 5 2 1 9 5 4

In above list assume 4 is pivot element so rewrite list as:

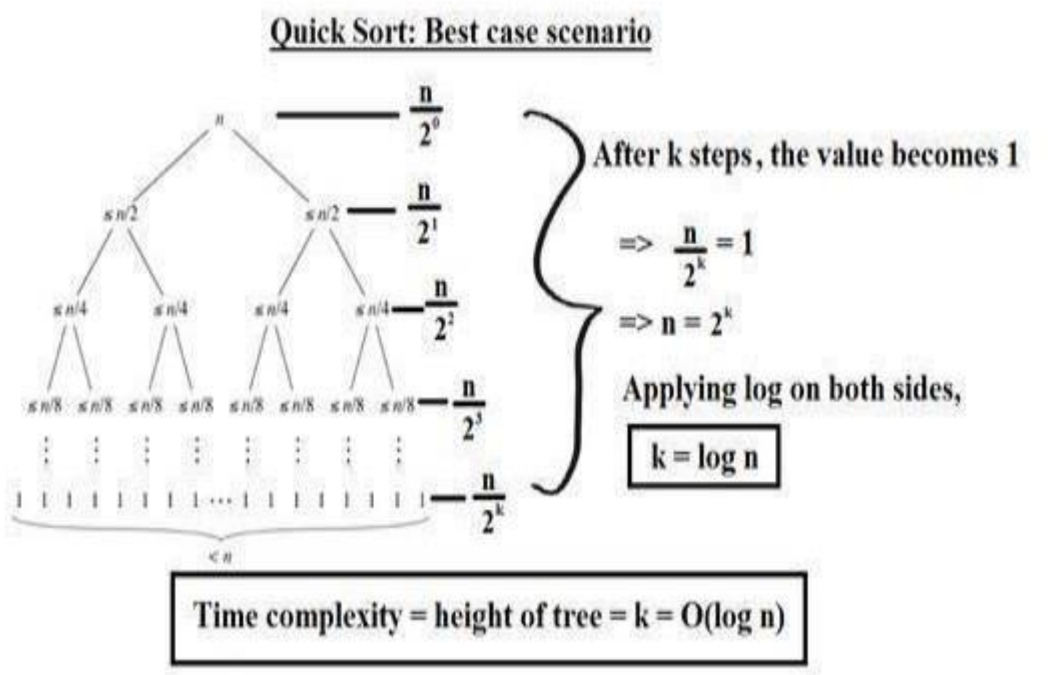
3 1 2 4 5 8 9 5 7

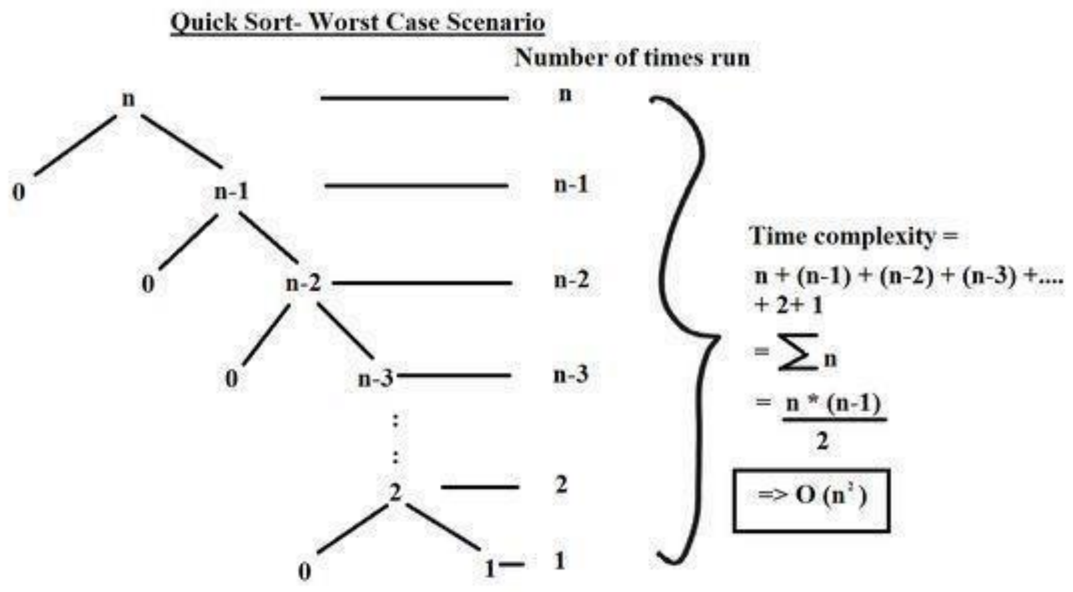
Here, I want to say that we set the pivot element (4) which has in left side elements are less than and right hand side elements are greater than. Now you think, how's arrange the less than and greater than elements? Be patient, you get answer soon.

Example:



```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high) {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
/* This function takes last element as pivot, places
 the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
 partition (arr[], low, high)
 {
    // pivot (Element to be placed at right
    position) pivot = arr[high];

    i = (low - 1)  // Index of smaller element and indicates the
                // right position of pivot found so far
    for (j = low; j <= high- 1; j++) {
        // If current element is smaller than the
        pivot if (arr[j] < pivot)
        {
            i++;     // increment index of smaller
            element swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
 }
```


Quick Sort: Best case scenario
n
≤ n/2
≤ n/4
≤ n/8
1
< n
$\frac{n}{2^0}$
$\frac{n}{2^1}$
$\frac{n}{2^2}$
$\frac{n}{2^3}$
$\frac{n}{2^k}$
After k steps, the value becomes 1
=> $\frac{n}{2^k} = 1$
=> $n = 2^k$
Applying log on both sides,
k = log n
Time complexity = height of tree = k = O(log n)

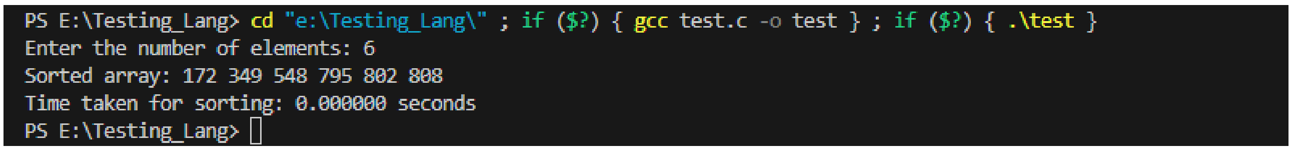


Code :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Selecting the last element as pivot
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            // Swap arr[i] and arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    // Swap arr[i+1] and arr[high] (pivot)
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return (i + 1);
}

// Function to perform Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partitioning index
        int pivot_index = partition(arr, low, high);

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

// Function to generate random array of given size
void generateRandomArray(int arr[], int n) {
    srand(time(NULL));
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 1000; // Generating random numbers between 0 and 999
    }
}

int main() {
    int n, i;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    // Generating random array
    generateRandomArray(arr, n);

    // Sorting the array using Quick Sort
    clock_t start = clock();
    quickSort(arr, 0, n - 1);
    clock_t end = clock();
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

    // Displaying sorted array
    printf("Sorted array: ");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Displaying time taken for sorting
    printf("Time taken for sorting: %f seconds\n", time_taken);

    return 0;
}
```

Output :



```
PS E:\Testing_Lang> cd "e:\Testing_Lang\" ; if ($?) { gcc test.c -o test } ; if ($?) { .\test }
Enter the number of elements: 6
Sorted array: 172 349 548 795 802 808
Time taken for sorting: 0.000000 seconds
PS E:\Testing_Lang> 
```

Conclusion :

This program prompts the user to enter the number of elements 'n'. It then generates a random array of 'n' elements, sorts the array using the Quick Sort algorithm with the Divide and Conquer technique, and displays the sorted array along with the time taken for sorting.

To perform a comparative analysis for large values of 'n', you can modify the program to execute multiple times with increasing values of 'n' and measure the time taken for each execution. Additionally, you can compare the performance of Quick Sort with other sorting algorithms to analyze their efficiency for different input sizes.