

## **EXPERIMENT 5**

**AIM :** Sort a given set of elements using quick sort algorithm and find the time complexity for different values of n.

### **THEORY :**

Quick Sort is a powerful divide-and-conquer sorting algorithm widely used in practice due to its efficiency. It begins by choosing a pivot element from the array. The array is then rearranged so that all elements smaller than the pivot are placed on its left, and all greater elements on its right — this step is called partitioning. After partitioning, the pivot is in its correct sorted position. The same process is applied recursively to the left and right subarrays until the entire list is sorted. Quick Sort performs very well on large datasets, with an average time complexity of  $O(n \log n)$ , though it can degrade to  $O(n^2)$  in the worst case if poor pivots are chosen.

## **ALGORITHM :**

1. **QuickSort(arr, low, high)**
2. If  $low < high$  then
3.   a.  $p = \text{Partition}(arr, low, high)$
4.   b.  $\text{QuickSort}(arr, low, p - 1)$
5.   c.  $\text{QuickSort}(arr, p + 1, high)$

## **Partition(arr, low, high):**

1. Set  $pivot = arr[high]$
2. Set  $i = low - 1$
3. For  $j = low$  to  $high - 1$ :

If  $arr[j] \leq pivot$ :

$i = i + 1$

swap  $arr[i]$  and  $arr[j]$

4. Swap  $arr[i + 1]$  and  $arr[high]$
5. Return  $i + 1$  (pivot index)

## SOURCE CODE:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 void swap(int *a, int *b)
6 {
7     int t = *a;
8     *a = *b;
9     *b = t;
10 }
11
12 int partition(int arr[], int low, int high)
13 {
14     int pivot = arr[high];
15     int i = low - 1;
16     for (int j = low; j < high; j++)
17     {
18         if (arr[j] <= pivot)
19         {
20             i++;
21             swap(&arr[i], &arr[j]);
22         }
23     }
24     swap(&arr[i + 1], &arr[high]);
25     return i + 1;
26 }
27
28 void quickSort(int arr[], int low, int high)
29 {
30     if (low < high)
31     {
32         int pi = partition(arr, low, high);
33         quickSort(arr, low, pi - 1);
34         quickSort(arr, pi + 1, high);
35     }
36 }
37
38 int main()
39 {
40     srand(time(NULL));
41     FILE *file = fopen("quick_sort_data.csv", "w");
42     if (file == NULL)
43     {
44         perror("Error opening file");
45         return 1;
46     }
47     fprintf(file, "Size,Time Taken(s)\n");
```

```

49  /*
50   int n;
51   printf("Enter the size of the array: ");
52   scanf("%d", &n);
53
54   int *arr = malloc(n * sizeof(int));
55   if (!arr) {
56       printf("Memory allocation failed\n");
57       return 1;
58   }
59   printf("Enter %d integers:\n", n);
60   for (int i=0; i<n; i++) {
61       scanf("%d", &arr[i]);
62   }
63   quickSort(arr, 0, n-1);
64 */
65
66   int sizes[] = {100, 1000, 10000, 50000, 80000, 100000, 150000};
67   int count = sizeof(sizes) / sizeof(sizes[0]);
68
69   for (int k = 0; k < count; k++)
70   {
71       int n = sizes[k];
72       int *arr = malloc(n * sizeof(int));
73       if (!arr)
74       {
75           printf("Memory allocation failed\n");
76           return 1;
77       }
78       for (int i = 0; i < n; i++)
79       {
80           arr[i] = rand();
81       }
82
83       clock_t start = clock();
84       quickSort(arr, 0, n - 1);
85       clock_t end = clock();
86
87       double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
88       fprintf(file, "%d,%lf\n", n, time_taken);
89
90       printf("size: %d\n", n);
91       printf("time taken: %0.6lf seconds\n", time_taken);
92
93       free(arr);
94   }
95
96   fclose(file);
97   return 0;
98 }

```

**OUTPUT:**

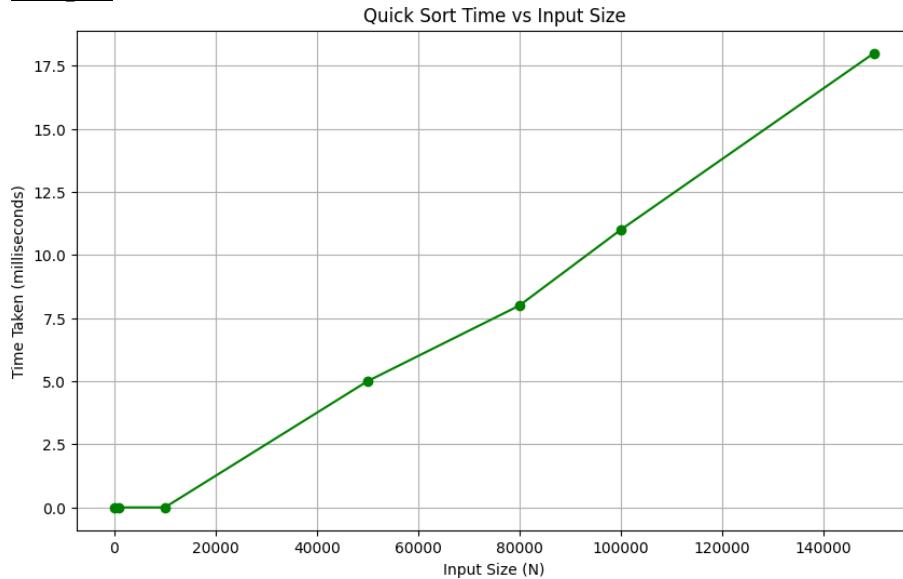
```
Enter the size of the array: 10
Enter 10 integers:
45
21
37
95
44
23
18
36
91
35
Original array:
[45, 21, 37, 95, 44, 23, 18, 36, 91, 35]
Sorted array:
[18, 21, 23, 35, 36, 37, 44, 45, 91, 95]
size: 100
time taken: 0.000000 seconds
size: 1000
time taken: 0.000000 seconds
size: 10000
time taken: 0.000000 seconds
size: 50000
time taken: 0.005000 seconds
size: 80000
time taken: 0.008000 seconds
size: 100000
time taken: 0.011000 seconds
size: 150000
time taken: 0.018000 seconds
```

**Input (n) Vs Time Taken (ms) Table:**

Array Size(n)	Time Taken (milliseconds)
100	0.000
1000	0.000
10000	0.000
50000	5.000
80000	8.000
100000	11.000

150000	18.000
--------	--------

### Graph:



### TIME COMPLEXITY :

	BEST CASE	WORST CASE	AVERAGE CASE
<b>QUICK SORT</b>	$O(n \log n)$	$O(n^2)$	$O(n \log n)$

### LEARNING OUTCOMES :