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:

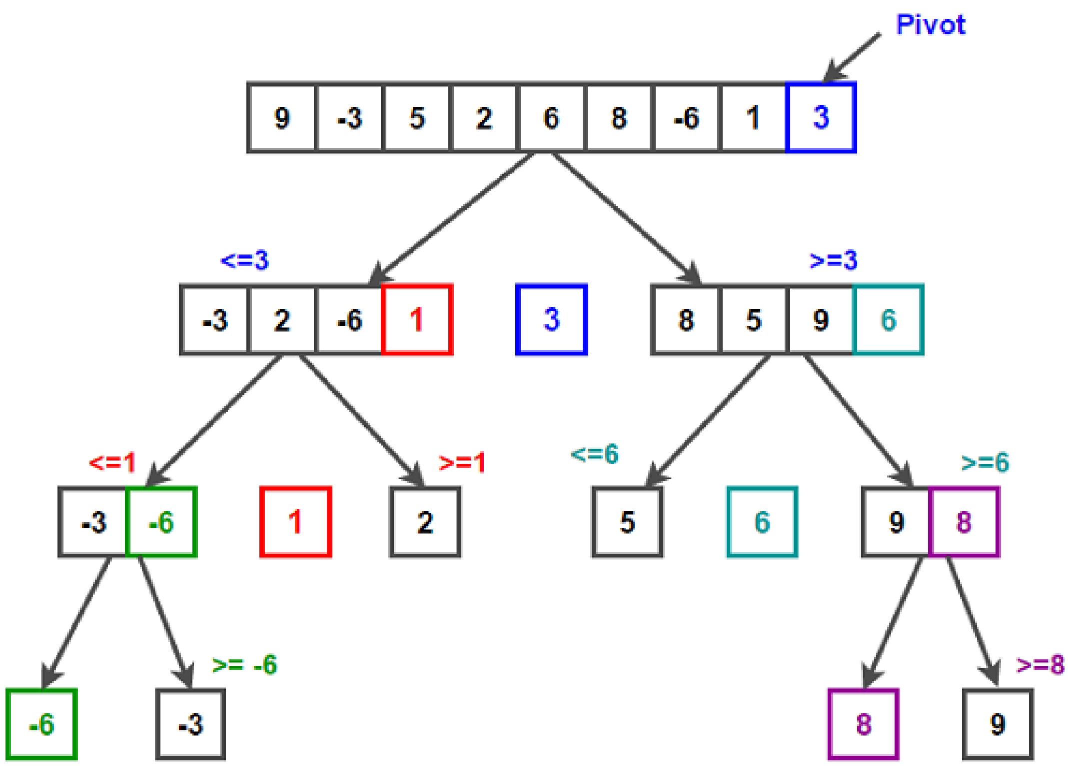
List : 3 7 8 5 2 1 9 5 4

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.

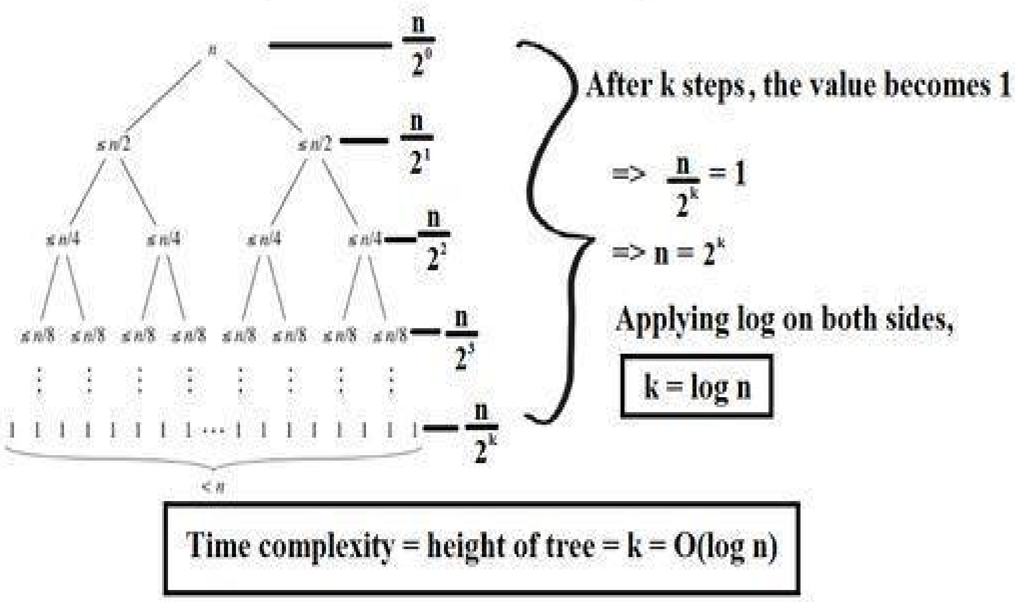
Example:



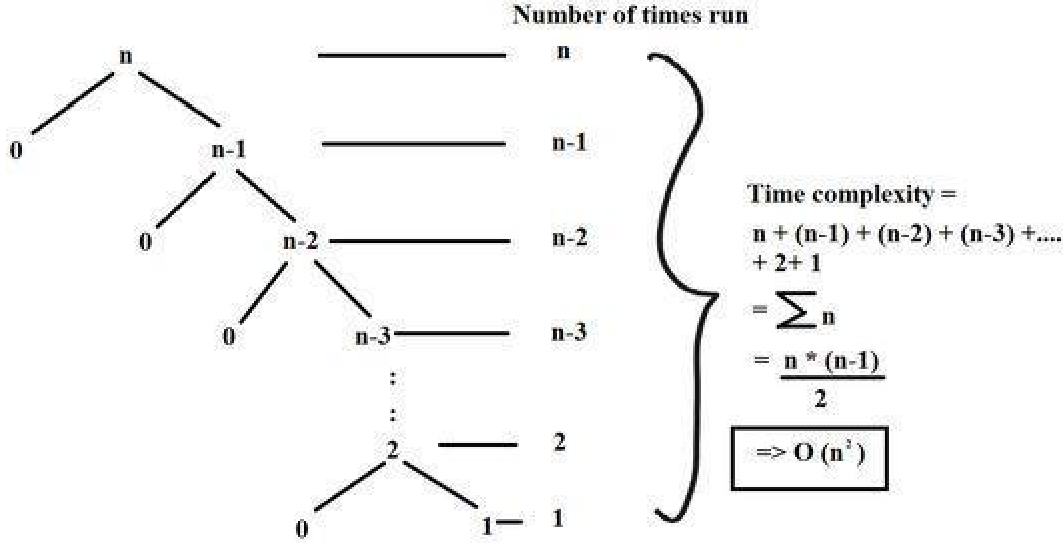
```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
   if (low < high)
   {
      /* pi is partitioning index, arr[pi] is now
         at right place */
      pi = partition(arr, low, high);
      quickSort(arr, low, pi - 1);  // Before pi
      quickSort(arr, pi + 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



Quick Sort- Worst Case Scenario

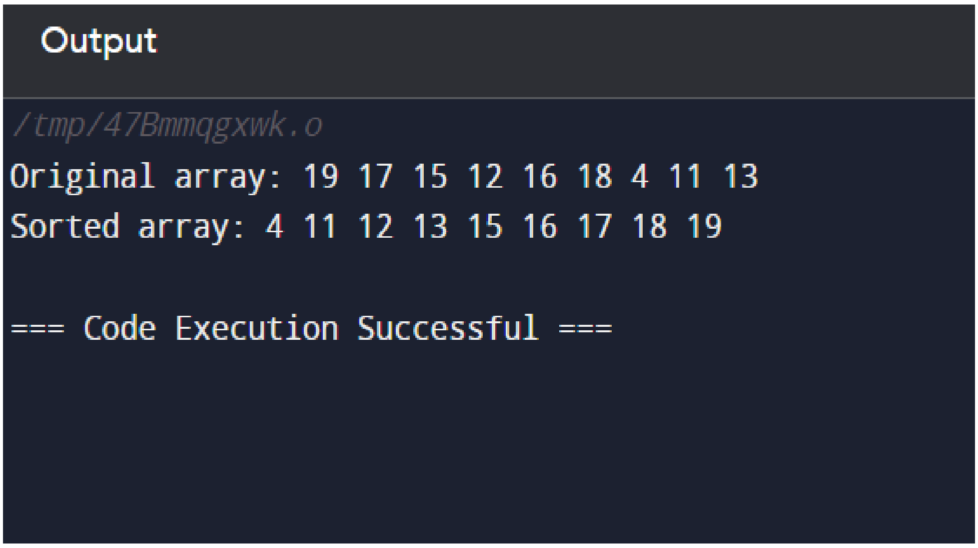


Implementation:

```
#include <stdio.h>
void swap(int* a, int* b)
{
int temp = *a;
*a = *b;
*b = temp;
}
int partition(int arr[], int low, int high)
{
int pivot = arr[low]; int i = low;
int j = high; while (i < j)
{
while (arr[i] <= pivot && i <= high - 1) { i++;
}
while (arr[j] > pivot && j >= low + 1) { j--;
}
if (i < j) {
swap(&arr[i], &arr[j]);
}
}
swap(&arr[low], &arr[j]); return j;
}
void quickSort(int arr[], int low, int high)
{
if (low < high) {
int partitionIndex = partition(arr, low, high); quickSort(arr, low, partitionIndex - 1); quickSort(arr, partitionIndex + 1, high);
}
```

```
}
int main()
{
int arr[] = { 19, 17, 15, 12, 16, 18, 4, 11, 13 };
int n = sizeof(arr) / sizeof(arr[0]); printf("Original array: ");
for (int i = 0; i < n; i++) {
printf("%d ", arr[i]);
}
quickSort(arr, 0, n - 1); printf("\nSorted array: "); for (int i = 0; i < n; i++) {
printf("%d ", arr[i]);
}
return 0;
}
```

Output:



Conclusion: Quick sort is a powerful algorithm for sorting large datasets efficiently, but careful implementation and consideration of pivot selection are crucial for achieving optimal performance in all scenarios.