Quicksort Algorithm

Quicksort is a sorting algorithm based on the **divide and conquer approach** where

1. An array is divided into subarrays by selecting a **pivot element** (element selected from the array).

   While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Sorting is a way of arranging items in a systematic manner. Quicksort is the widely used sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Algorithm

Algorithm:

```
QUICKSORT (array A, start, end)
{
  if (start < end)
  {
    p = partition(A, start, end)
    QUICKSORT (A, start, p - 1)
    QUICKSORT (A, p + 1, end)
  }
}
```

Partition Algorithm:

The partition algorithm rearranges the sub-arrays in a place.

```
1.  PARTITION (array A, start, end)
2.  {
3.    pivot ? A[end]
4.    i ? start-1
5.    for j ? start to end -1 {
6.    do if (A[j] < pivot) {
7.    then i ? i + 1
8.    swap A[i] with A[j]
9.     }}
10.   swap A[i+1] with A[end]
11.   return i+1
12. }
```

Quicksort Complexity

Time Complexity	
Best	O(n*log n)
Worst	O(n^2)
Average	O(n*log n)
Space Complexity	O(log n)
Stability	No

1. Time Complexities

- **Worst Case Complexity [Big-O]**: O(n^2)
  It occurs when the pivot element picked is either the greatest or the smallest element.

  This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains `n - 1` elements. Thus, quicksort is called only on this sub-array.

However, the quicksort algorithm has better performance for scattered pivots.

- **Best Case Complexity [Big-omega]**: `O(n*log n)`
  It occurs when the pivot element is always the middle element or near to the middle element.
- **Average Case Complexity [Big-theta]**: `O(n*log n)`
  It occurs when the above conditions do not occur.

2. Space Complexity

The space complexity for quicksort is `O(log n)`.

Quicksort Applications

Quicksort algorithm is used when

- the programming language is good for recursion
- time complexity matters
- space complexity matters

Complexity Analysis of Quick Sort:

Time Complexity:

- **Best Case:** Ω (N log (N))
- **Average Case:** θ (N log (N))
- **Worst Case:** O(N^2)

Auxiliary Space: O(1), if we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make *O*(*N*).

Advantages of Quick Sort:

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.

Disadvantages of Quick Sort:

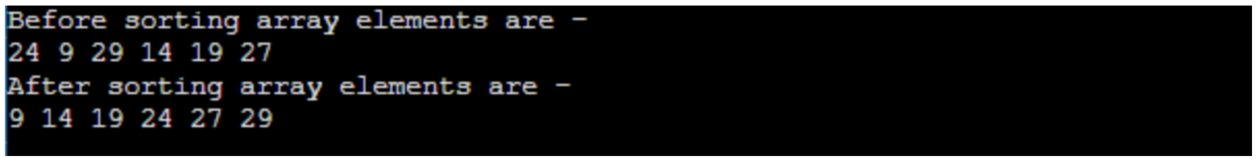
- It has a worst-case time complexity of O(N^2), which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

Program: Write a program to implement quicksort in C language.

```
#include <stdio.h>
/* function that consider last element as pivot,
place the pivot at its exact position, and place
smaller elements to left of pivot and greater
elements to right of pivot.  */
int partition (int a[], int start, int end)
{
    int pivot = a[end]; // pivot element
    int i = (start - 1);

    for (int j = start; j <= end - 1; j++)
    {
        // If current element is smaller than the pivot
        if (a[j] < pivot)
        {
            i++; // increment index of smaller element
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
    int t = a[i+1];
    a[i+1] = a[end];
    a[end] = t;
    return (i + 1);
}

/* function to implement quick sort */
void quick(int a[], int start, int end) /* a[] = array to be sorted, start = Starting index, end =
Ending index */
{
    if (start < end)
    {
        int p = partition(a, start, end); //p is the partitioning index
```

```c
        quick(a, start, p - 1);
        quick(a, p + 1, end);
    }
}

/* function to print an array */
void printArr(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
}
int main()
{
    int a[] = { 24, 9, 29, 14, 19, 27 };
    int n = sizeof(a) / sizeof(a[0]);
    printf("Before sorting array elements are - \n");
    printArr(a, n);
    quick(a, 0, n - 1);
    printf("\nAfter sorting array elements are - \n");
    printArr(a, n);

    return 0;
}
```

Output:



```
Before sorting array elements are -
24 9 29 14 19 27
After sorting array elements are -
9 14 19 24 27 29
```

Quicksort Pivot Algorithm

Based on our understanding of partitioning in quicksort, we will now try to write an algorithm for it, which is as follows.

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left

Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot