```cpp
#include <iostream>
using namespace std;

int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {

        if (arr[j] < pivot)
        {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {

        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main()
{

    int n;
    cin >> n;

    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    cout << "Given array is \n";
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    cout << "\nSorted array is \n";
    printArray(arr, n);
    return 0;
}
```

Complexity of Quick Sort

Recurrence Relation

Quick Sort partitions the array and recursively sorts left & right parts.

If the pivot splits the array into two parts:

$T(n) = T(k) + T(n - k - 1) + O(n)$

where **k** is the number of elements on the left of pivot.

Partitioning takes $O(n)$.

Best / Average Case

If the pivot splits perfectly in the middle:

$T(n) = 2T(n/2) + O(n)$

,$T(n) = O(n \log n)$

Worst Case

If the pivot is always the smallest or largest element:

One part has 0 elements.

Other part has $(n - 1)$ elements.

So, $T(n) = T(n - 1) + O(n)$

Expand: $T(n) = T(n - 1) + n$
$= T(n - 2) + (n - 1) + n =$
...
$= 1 + 2 + 3 + ... + n$
$= O(n^2)$

Space Complexity: - Average: $O(\log n)$ recursion stack - Worst: $O(n)$ recursion stack

```cpp
#include <iostream>
using namespace std;

void merge(int arr[], int left, int mid, int right)
{
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, arr_size - 1);

    cout << "\nSorted array is \n";

    for (int i = 0; i < arr_size; i++)
        cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

Complexity of Merge Sort

Recurrence Relation

Merge Sort divides the array into 2 halves and merges them.

So, the time to sort **n** elements is:

T(n) = 2 * T(n/2) + O(n)

2 * T(n/2): Sorting the two halves. -

O(n): Merging the two halves.

Step 2: Solve the Recurrence

Expand:

Expand:

T(n) = 2T(n/2) + n

= 2[2T(n/4) + n/2] + n

= 4T(n/4) + 2n

= 8T(n/8) + 3n

= ...

Step 3: Generalize

After k levels:

T(n) = 2^k * T(n/2^k) + k * n When n/2^k = 1:

=> 2^k = n

=> k = log2(n)

So,

$$T(n) = n * T(1) + n \log_2(n)$$

$$= O(n \log n)$$

Space Complexity: O(n) (due to temporary arrays)