5. Hoare Quick Sort in C++

This section explains how the "divide and conquer" strategy of Quick Sort is implemented in C++. The logic is split into two primary functions: `quickSort()` which handles the recursive division, and `partition()` which rearranges the array using the classic **Hoare partition scheme**.

The C++ Code Implementation

Here is the complete code for a Quick Sort implementation using the Hoare partition scheme.

```
#include <iostream>
#include <vector>
#include <utility> // For std::swap

// Utility function to print a vector
void printVector(const std::vector<int>& arr) {
    for (int num : arr) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
}

// This function implements the Hoare partition scheme. It partitions the
// array and returns the split point.
int partition(std::vector<int>& arr, int low, int high) {
    // Choose the first element as the pivot (key to this Hoare implementation)
    int pivot = arr[low];
    int i = low - 1;
    int j = high + 1;

    while (true) {
        // Find an element on the left side that should be on the right
        do {
            i++;
        } while (arr[i] < pivot);

        // Find an element on the right side that should be on the left
        do {
```

```
            j--;
        } while (arr[j] > pivot);

        // If the pointers have crossed, the partition is done
        if (i >= j) {
            return j;
        }

        // Swap the elements to place them on the correct side
        std::swap(arr[i], arr[j]);
    }
}

// The main recursive function that implements Quick Sort
void quickSort(std::vector<int>& arr, int low, int high) {
    // Base case: if the array has 1 or 0 elements, it's already sorted
    if (low < high) {
        // pi is the split point from the Hoare partition
        int pi = partition(arr, low, high);

        // Recursively sort the two partitions
        quickSort(arr, low, pi);
        quickSort(arr, pi + 1, high);
    }
}

// Main driver function
int main() {
    std::vector<int> data = {6, 5, 3, 1, 8, 7, 2, 4};
    std::cout << "Original array: ";
    printVector(data);

    quickSort(data, 0, data.size() - 1);

    std::cout << "Sorted array:   ";
    printVector(data);

    return 0;
}
```

Output

```
Original array: 6 5 3 1 8 7 2 4
Sorted array:   1 2 3 4 5 6 7 8
```

Code Breakdown

The `quickSort()` Function: The Recursive Driver

This function orchestrates the **"Divide"** phase of the algorithm.

1. **Base Case**: The recursion continues as long as `low < high`. If `low >= high`, it means the sub-array has one or zero elements.
2. **Partition**: It calls the `partition()` function, which rearranges the sub-array and returns a **split point index** `pi`.
3. **Recurse**: It calls itself on the two partitions. Because the Hoare scheme's split point `pi` is not guaranteed to be the pivot's final position, the recursive calls are `quickSort(arr, low, pi)` and `quickSort(arr, pi + 1, high)` to ensure all elements are included in the sorting process.

The `partition()` Function (Hoare Scheme)

This function implements the classic **Hoare partition scheme**. It is generally more efficient than Lomuto because it performs fewer swaps on average.

1. **Pivot Selection**: It selects the **first element** (`arr[low]`) as the pivot.
2. **Two Pointers**: It uses two pointers, `i` and `j`, which start at opposite ends of the array and move toward each other.
3. **Rearranging Loop**: The `i` pointer moves right to find the first element greater than or equal to the pivot, while the `j` pointer moves left to find the first element less than or equal to the pivot. If the pointers haven't crossed, the two elements are swapped.
4. **Return Index**: The loop terminates when the pointers cross. The function returns the final position of the `j` pointer, which serves as the split point for the two partitions. Unlike Lomuto, the pivot is **not** guaranteed to be at this returned index.

Revision #1
Created 2025-09-16 15:54:56 UTC by AX
Updated 2025-09-16 15:56:25 UTC by AX