4. Lomuto 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 popular **Lomuto partition scheme**.

The C++ Code Implementation

Here is the complete code for a Quick Sort implementation using `std::vector`.

```
#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 Lomuto partition scheme. It takes the last
// element as pivot and places it at its correct position in the sorted array.
int partition(std::vector<int>& arr, int low, int high) {
    // Choose the last element as the pivot (key to Lomuto)
    int pivot = arr[high];

    // Index of the smaller element, acting as a "wall"
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        // If the current element is smaller than the pivot
        if (arr[j] < pivot) {
            i++; // Move the wall to the right
            std::swap(arr[i], arr[j]);
        }
```

```
    }
    // Place the pivot in its correct position after the wall
    std::swap(arr[i + 1], arr[high]);
    return (i + 1);
}

// 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 partitioning index from the Lomuto partition
        int pi = partition(arr, low, high);

        // Separately sort elements before and after the partition
        quickSort(arr, low, pi - 1);
        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, which is inherently sorted.
2. **Partition**: It calls the `partition()` function. This function rearranges the sub-array and returns the index `pi` where the pivot element is now correctly placed.
3. **Recurse**: It calls itself twice for the two sub-arrays created by the partition: one for the elements to the left of the pivot (`low` to `pi - 1`) and one for the elements to the right (`pi + 1` to `high`). Because this uses the Lomuto scheme, the pivot at `pi` is guaranteed to be in its final place and can be excluded from recursion.

The `partition()` Function (Lomuto Scheme)

This function implements the well-known **Lomuto partition scheme**. It is where the key logic of rearranging elements occurs and can be seen as the **"Conquer"** step, as it correctly places one element (the pivot) at a time.

1. **Pivot Selection**: It selects the **last element** of the sub-array (`arr[high]`) as the **pivot**. This is the defining starting point for this version of Lomuto.
2. **Rearranging Loop**: It maintains an index `i` (the "wall") which represents the boundary for elements smaller than the pivot. It iterates through the sub-array with another index `j` (the "scanner"). If it finds an element `arr[j]` that is smaller than the pivot, it moves the wall (`i++`) and swaps `arr[j]` with the element at the new wall position, effectively expanding the "smaller than pivot" section.
3. **Final Pivot Placement**: After the loop finishes, all elements smaller than the pivot are at the beginning of the sub-array. The pivot (still at `arr[high]`) is then swapped with the element at `arr[i + 1]`. This places the pivot exactly where it belongs in the sorted array.
4. **Return Index**: The function returns the new index of the pivot (`i + 1`), so the `quickSort` function knows where to split the array for its next recursive calls.

Revision #2
Created 2025-09-16 13:23:21 UTC by AX
Updated 2025-09-16 15:53:34 UTC by AX