2. Merge Sort in C++

This section explains how the "divide and conquer" strategy of Merge Sort is implemented in C++. The logic is split into two primary functions: `mergeSort()` which handles the recursive division, and `merge()` which handles the conquering and sorting.

The C++ Code Implementation

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

```
#include <iostream>
#include <vector>

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

// Merges two sorted subarrays into a single sorted subarray.
// First subarray is arr[left..mid]
// Second subarray is arr[mid+1..right]
void merge(std::vector<int>& arr, int left, int mid, int right) {
    // Calculate sizes of the two temporary subarrays
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temporary vectors to hold the data
    std::vector<int> L(n1), R(n2);

    // Copy data from the main array to the temporary vectors
    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];
```

```
    // Merge the temporary vectors back into the original array arr[left..right]
    int i = 0; // Initial index for left subarray
    int j = 0; // Initial index for right subarray
    int k = left; // Initial index for the merged subarray

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

    // Copy any remaining elements from the left subarray
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy any remaining elements from the right subarray
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// The main recursive function that implements the "divide" phase
void mergeSort(std::vector<int>& arr, int left, int right) {
    // Base case: if the array has 1 or 0 elements, it's already sorted
    if (left >= right) {
        return;
    }
```

```
    // Find the middle point to divide the array
    int mid = left + (right - left) / 2;

    // Recursively call mergeSort for the two halves
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);

    // Merge the two now-sorted halves
    merge(arr, left, mid, right);
}

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

    mergeSort(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

Code Breakdown

The `mergeSort()` Function: The Divider

This function is the engine that drives the **"Divide"** phase.

1. **Base Case**: The recursion stops when `left >= right`, which means the sub-array has one or zero elements and is considered sorted.
2. **Divide**: It calculates the `mid` point of the current array segment. The formula `left + (right - left) / 2` is used to prevent potential overflow on very large arrays.
3. **Recurse**: It calls itself for the left half (`left` to `mid`) and then for the right half (`mid + 1` to `right`).
4. **Conquer**: After the recursive calls return (guaranteeing the two halves are sorted), it calls `merge()` to combine them into a single, sorted segment.

The `merge()` Function: The Conqueror

This function is the core of the algorithm where the actual sorting happens. It takes two adjacent, sorted sub-arrays and merges them.

1. **Setup**: It creates two temporary vectors, `L` and `R`, and copies the data from the two halves of the main array into them.
2. **Merge Loop**: It uses three index pointers: `i` for vector `L`, `j` for vector `R`, and `k` for the original array `arr`. It compares the elements at `L[i]` and `R[j]` and places the smaller of the two into `arr[k]`. It then increments the pointer of the vector from which the element was taken, as well as the pointer `k`.
3. **Copy Leftovers**: After the main loop, one of the temporary vectors might still contain elements. The final two `while` loops copy these remaining elements back into the main array, ensuring all data is merged correctly.

Revision #1
Created 2025-09-16 03:55:40 UTC by AX
Updated 2025-09-16 04:02:46 UTC by AX