Merge sort algorithm

***Source:* https://github.com/rusevrosen/codehub.github.io/tree/main/Algorithms/Sorting_Algorithms**

1. What is the Merge Sort algorithm?

Merge sort is an efficient, general-purpose, comparison-based sorting algorithm. It works by dividing an array into two halves, sorting each half, and then merging the sorted halves back together.

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

2. Code example.

```
#include <iostream>

#include <vector>


using namespace std;


void merge(vector<int> &arr, int left, int mid, int right) {

    // Create temporary arrays to hold the left and right halves of the input array

    vector<int> left_arr(mid - left + 1);

    vector<int> right_arr(right - mid);


    // Copy the left and right halves of the input array into the temporary arrays

    for (int i = 0; i < left_arr.size(); i++) {

        left_arr[i] = arr[left + i];

    }

    for (int i = 0; i < right_arr.size(); i++) {

        right_arr[i] = arr[mid + 1 + i];

    }


    // Merge the temporary arrays back into the input array in sorted order

    int left_index = 0, right_index = 0;

    for (int i = left; i <= right; i++) {

        if (left_index == left_arr.size()) {

            // Left array is exhausted, copy remaining elements from right array

            arr[i] = right_arr[right_index];

            right_index++;

        } else if (right_index == right_arr.size()) {

            // Right array is exhausted, copy remaining elements from left array

            arr[i] = left_arr[left_index];

            left_index++;

        } else if (left_arr[left_index] < right_arr[right_index]) {

            // Left element is smaller, copy it to the output array

            arr[i] = left_arr[left_index];

            left_index++;

        }
        else {
            // Right element is smaller, copy it to the output array
            arr[i] = right_arr[right_index];
            right_index++;
        }
    }
}

void merge_sort(vector<int> &arr, int left, int right) {
    if (left < right) {
        // Divide the array into two halves and sort them recursively
        int mid = left + (right - left) / 2;
        merge_sort(arr, left, mid);
        merge_sort(arr, mid + 1, right);

        // Merge the sorted halves back together
        merge(arr, left, mid, right);
    }
}

int main() {
    // Sort the array [6, 5, 3, 1, 8, 7, 2, 4]
    vector<int> arr{6, 5, 3, 1, 8, 7, 2, 4};
    merge_sort(arr, 0, arr.size() - 1);

    // Print the sorted array
    for (int x : arr) {
        cout << x << " ";
    }
    cout << endl;

    return 0;
}
```

This program will output the sorted array: 1 2 3 4 5 6 7 8.

3. Code explanation.

The merge sort algorithm works by dividing the input array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

The **`merge_sort`** function uses recursion to divide the input array into smaller and smaller subarrays until each subarray consists of only a single element, which is already considered sorted. The function then begins merging the subarrays back together, starting with the subarrays of size 1, and combining them into larger and larger sorted arrays.

The merge function is responsible for actually merging the subarrays back together. It does this by creating two temporary arrays to hold the left and right halves of the input array, and then merging these temporary arrays back into the input array in sorted order.

The main function in the example sorts the array **`[6, 5, 3, 1, 8, 7, 2, 4]`** using the **`merge_sort`** function and prints the sorted array to the console.

4. Time complexity.

The time complexity of the merge sort algorithm in the example above is **`O(n * log(n)).`**

This means that the time taken by the algorithm grows at most logarithmically with the size of the input. This makes it more efficient than other sorting algorithms, such as bubble sort and insertion sort, which have a time complexity of **`O(n^2)`** and are much slower for large inputs.

The logarithmic time complexity of merge sort is due to the divide-and-conquer approach used by the algorithm. The input array is repeatedly split in half until each subarray consists of only a single element, and then the subarrays are merged back together. This results in a time complexity of O(log(n)) for the divide step, and O(n) for the merge step, giving a total time complexity of **`O(n * log(n)).`**

..

I wish you happy learning,

Your Rosen Rusev