DAA Assignment 4

**Group:**

* **M.taha (54592)**
* **Umair Waseem (44296)**
* **Muhammad Bilal (45775)**
* **Hamad Ahmed Inayat (46391)**

**BSCS**

**GitHub Repository link:**

<https://github.com/tahajani/Assignment_4>

**Code:**

#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 i = 0; i < n2; i++)

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

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)

return;

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = low - 1;

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

if (arr[j] <= pivot) {

i++;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]);

return i + 1;

}

void hybridSort(int arr[], int low, int high, int threshold) {

if (low < high) {

if (high - low < threshold) {

mergeSort(arr, low, high);

} else {

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

hybridSort(arr, low, pivot - 1, threshold);

hybridSort(arr, pivot + 1, high, threshold);

}

}

}

void printArray(int arr[], int size) {

for (int i = 0; i < size; i++)

cout << arr[i] << " ";

cout << endl;

}

int main() {

int arr[] = {38, 27, 43, 3, 9, 82, 10};

int n = sizeof(arr) / sizeof(arr[0]);

int threshold = 10;

cout << "Original array:\n";

printArray(arr, n);

hybridSort(arr, 0, n - 1, threshold);

cout << "Sorted array:\n";

printArray(arr, n);

return 0;

}

**Explanation:**

1. **merge Function:**
   * This function merges two sorted subarrays of arr[].
   * **Parameters**:
     + arr[]: The original array.
     + left: Starting index of the first subarray.
     + mid: Ending index of the first subarray.
     + right: Ending index of the second subarray.
   * It creates temporary arrays L[] and R[] for the left and right subarrays, then merges them back into the original array.
2. **mergeSort Function:**
   * This is the recursive Merge Sort algorithm.
   * It splits the array into halves and recursively sorts each half using the merge function to combine them.
   * **Parameters**:
     + arr[]: The array to sort.
     + left and right: The starting and ending indices of the current subarray.
3. **partition Function:**
   * This is the partitioning logic of Quick Sort.
   * **Parameters**:
     + arr[]: The array to partition.
     + low and high: The starting and ending indices of the current subarray.
   * It chooses the last element as the pivot, rearranges the elements so that those less than the pivot come before it, and those greater come after it.
   * Returns the final position of the pivot.
4. **hybridSort Function:**
   * This function combines Quick Sort and Merge Sort.
   * **Parameters**:
     + arr[]: The array to sort.
     + low and high: The starting and ending indices of the current subarray.
     + threshold: The size limit below which it switches to Merge Sort.
   * **Working**:
     + It checks the size of the current partition.
     + If the size is smaller than the threshold, it uses mergeSort.
     + Otherwise, it continues with Quick Sort using the partition function.
5. **printArray Function:**
   * Prints the elements of the array.
   * **Parameters**:
     + arr[]: The array to print.
     + size: The number of elements in the array.
6. **main Function:**
   * Initializes an array and calls hybridSort.
   * **Output**:
     + Prints the original and sorted array.

**Performance Analysis:**

* **Time Complexity**:
  + **Average and Best Case**: O(nlog⁡n)O(n \log n)O(nlogn), since the algorithm leverages the efficiency of both Quick Sort and Merge Sort.
  + **Worst Case**: O(nlog⁡n)O(n \log n)O(nlogn), as the use of Merge Sort for smaller partitions helps avoid Quick Sort's worst-case scenarios.
* **Space Complexity**:
  + Slightly higher than O(n)O(n)O(n), due to the additional memory used in the merge step of Merge Sort.

**Usage:**

This hybrid sorting algorithm combines the best of both worlds:

* Uses Quick Sort's in-place and efficient sorting on large arrays.
* Switches to Merge Sort for small subarrays to ensure stability and avoid Quick Sort's worst-case scenario.