DESIGN AND ANALYSIS OF ALGORITHM
ASSIGNMENT # 4

GROUP MEMBERS:

Muneeb-ur-Rehman	48046
Muhammad Wasif Qamar	45815
Hassan Tasdique	45897
Umer Shahmeer Ahmad	46194

QUESTION:

Consider two algorithms, Algorithm A and Algorithm B, designed to solve the same problem. Your task is to design a hybrid algorithm that combines the best aspects of both Algorithm A and Algorithm B to achieve improved performance.

SOLUTION:

- **Sorting Problem**: Algorithms like Merge Sort (Algorithm A) and Insertion Sort (Algorithm B) can be combined. Merge Sort is efficient for large datasets, while Insertion Sort is effective for small, nearly sorted arrays.
- **Shortest Path Problem**: Dijkstra's Algorithm (Algorithm A) and A* Search (Algorithm B) can be combined for path-finding, where Dijkstra's works well in weighted graphs, while A* is faster with heuristic information.

Algorithm A: Merge Sort

- **Strengths**: Divide-and-conquer approach; efficient with a time complexity of O(nlog⁡n)O(n \log n)O(nlogn) for large datasets.
- **Weaknesses**: Not ideal for smaller arrays due to recursion overhead and extra memory usage.

Algorithm B: Insertion Sort

- **Strengths**: Simple and efficient for small, nearly sorted arrays with a time complexity of O(n2)O(n^2)O(n2).
- **Weaknesses**: Inefficient for larger datasets; lacks divide-and-conquer capability.

The hybrid design will incorporate the best features of both algorithms by using Merge Sort's structure for large datasets but switching to Insertion Sort for smaller sub-arrays during the merge process. Here's how it could work:

1. **Divide Step**: Continue to split the array as in Merge Sort.
2. **Switch to Insertion Sort**: Once a sub-array size falls below a threshold (e.g., 10–20 elements), switch to Insertion Sort instead of continuing with recursive calls.
3. **Merge Step**: After sorting small sub-arrays with Insertion Sort, merge them back using the merge process in Merge Sort.

Theoretical Analysis

- **Time Complexity**: The hybrid algorithm retains the O(nlog⁡n)O(n \log n)O(nlogn) complexity of Merge Sort for larger datasets, but with smaller sub-arrays, it performs efficiently by reducing recursive calls.

- **Space Complexity**: Similar to Merge Sort; however, slightly reduced due to fewer recursive calls when switching to Insertion Sort for small sub-arrays.

Experimental Analysis

To validate, we can run the hybrid algorithm on different input sizes and compare with standard Merge Sort and Insertion Sort. Key metrics include:

- **Execution Time**: Measure time for each algorithm over multiple array sizes.
- **Memory Usage**: Track memory usage for recursion and merging steps.

After implementing, test the hybrid algorithm against both Merge Sort and Insertion Sort over various array sizes to compare:

- **Small arrays**: Hybrid should be slightly faster than Merge Sort due to reduced recursive calls.
- **Large arrays**: Performance similar to Merge Sort but potentially better in practical runtime due to early Insertion Sort handling for small subarrays.

CODE:

```
#include <iostream>
#include <vector>
using namespace std;

void insertionSort(vector<int>& arr, int left, int right) {
    for (int i = left + 1; i <= right; ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= left && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
void merge(vector<int>& arr, int left, int mid, int right) {
    int leftSize = mid - left + 1;
    int rightSize = right - mid;
    vector<int> leftSub(leftSize), rightSub(rightSize);
    for (int i = 0; i < leftSize; ++i)
        leftSub[i] = arr[left + i];
    for (int j = 0; j < rightSize; ++j)
        rightSub[j] = arr[mid + 1 + j];
    int i = 0, j = 0, k = left;
    while (i < leftSize && j < rightSize) {
        if (leftSub[i] <= rightSub[j]) {
            arr[k] = leftSub[i];
            i++;
        } else {
            arr[k] = rightSub[j];
            j++;
        }
        k++;
    }
    while (i < leftSize) {
```

```
        arr[k] = leftSub[i];
        i++;
        k++;
    }
    while (j < rightSize) {
        arr[k] = rightSub[j];
        j++;
        k++;
    }
}
void hybridSort(vector<int>& arr, int left, int right, int threshold = 10) {
    if (left < right) {
        if (right - left + 1 <= threshold) {
            insertionSort(arr, left, right);
        } else {
            int mid = left + (right - left) / 2;
            hybridSort(arr, left, mid, threshold);
            hybridSort(arr, mid + 1, right, threshold);
            merge(arr, left, mid, right);
        }
    }
}
int main() {
    vector<int> arr = {38, 27, 43, 3, 9, 82, 10};
    hybridSort(arr, 0, arr.size() - 1);
    cout << "Sorted array: ";
    for (int num : arr) {
        cout << num << " ";
    }
    cout << endl;
    return 0;
}
```

GITHUB LINK:

https://github.com/Umer46194/hybrid_design_of_algorithms_DAA.git

SAMPLE INPUT:

{38, 27, 43, 3, 9, 82, 10}

OUTPUT:

Sorted array: 3 9 10 27 38 43 82