

**Name: Muhammad Hasnain**

**Name: Sada Ullah Sap ID: 44315**

**Sap ID:45871**

**Name: Muhammad Hammad Sap ID:47326**

**Submitted To: Sir Usman Dep: Cyber Sec**

**Problem Selection:**

**Chosen Problem:** Sorting

**Selected Algorithm:** Merge Sort(Algorithm A) & Insertion Sort(Algorithm B)

**Reasoning:**These two algorithms are mostly used for sorting and they are composed of iteration with built-in operations that favors the variance of the data. In this method, Merge Sort is good for large data sets as well as for data that belongs to random order while Insertion sort is comparatively fast at sorting small data sets and data sets that have nearly been sorted.

# Analysis of Algorithms:

1. **Merge Sort:**
   * **Strengths:** It is splitting the data set recursively and it provides average as well as worst case time complexity of O(n log n). It is relatively stable for its use in large data set and for its high compatibility.
   * **Weaknesses:** additional memory has to be provided for the recursive stack and it is not the best in terms of performance indicating small data sets due to the\_funciotnality of splitting into sub-sets.

# Insertion Sort:

* + **Strengths:** It takes O(n) time with nearly sorted data and acts well on small list.
  + **Weaknesses:** However, for unsorted large data set or even data sets in random order, the efficient of this algorithm is not very good and is in the order of O (nZ).

# Scenarios:

* + Merge Sort is best for large, unsorted datasets.
  + Insertion Sort is optimal for small sub-arrays or nearly sorted data.

# Hybrid Algorithm Design:

Towards that end, the solution will propose the development of the so called Hybrid Sorting Algorithm, with Merge Sort as the main sorting methodology, but with the use of Insertion Sort on small sub- arrays. This switch enables us to optimize the algorithm’s ability not to make recursive calls to sort small sub-arrays.

# Design Choices:

* + **Combining Steps:**Merge Sort will be used to split the array, and Insertion Sort will handle small sub-arrays.
  + **Switching Criteria:**If the sub-array has less number of elements than a fixed number (say 10), use Insertion Sort.

# Pseudo-code:

function hybridSort(arr, threshold): if length(arr) < threshold:

return insertionSort(arr) else:

mid = length(arr) // 2

left = hybridSort(arr[:mid], threshold) right = hybridSort(arr[mid:], threshold) return merge(left, right)

# Explanation:

* + **threshold:** This parameter decides the index from where the algorithm is going to shift to the insertion sort technique. Usually, values should range from 10 to 20, however certain adjustments are needed.

# Performance Analysis:

**Theoretical Analysis**

## Time Complexity:

* + The hybrid algorithm keeps the average time complexity of Merge Sort to a value of O(n log n).
  + On peak, Insertion Sort is used for small sub-arrays to minimize the impacts of recursive calls than making the algorithm slower than normal Merge Sort in actual applications.

## Space Complexity:

* + The worst case space complexity stays at O(n) because of recursive splitting, however, loss of recursive depth in some partitions saves space.

# Experimental Analysis

1. **Implementation**: They contain the Hybrid Sort algorithm, and exploratory notes with datasets (random, nearly sorted, reverse-sorted) compared with Merge Sort and Insertion Sort.
2. **Parameter Tuning**: Adjust the values of threshold to reach the high accuracy which will define the best cut-off point.
3. **Testing:** Test performance in this program (time and space) and then compare it with Merge Sort and Insertion Sort.

# Python Code for Hybrid Sort Algorithm

def insertion\_sort(arr):

for i in range(1, len(arr)): key = arr[i]

j = i - 1

while j >= 0 and key < arr[j]: arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = key return arr

def merge(left, right): result = []

i = j = 0

while i < len(left) and j < len(right): if left[i] < right[j]:

result.append(left[i]) i += 1

else:

result.append(right[j]) j += 1

result.extend(left[i:]) result.extend(right[j:]) return result

def hybrid\_sort(arr, threshold=10): if len(arr) < threshold:

return insertion\_sort(arr) mid = len(arr) // 2

left = hybrid\_sort(arr[:mid], threshold) right = hybrid\_sort(arr[mid:], threshold) return merge(left, right)

import random

arr = random.sample(range(1, 100), 20) print("Original Array:", arr)

sorted\_arr = hybrid\_sort(arr) print("Sorted Array:", sorted\_arr)

# Experimental Validation

Run the hybrid algorithm with different values of threshold on various datasets to analyze the improvements

1. **Test Data**: Formulate small, medium and large groups of different random, sorted and reverse sorted data.
2. **Benchmarking**: Measure execution time and memory usage for each threshold value.
3. **Comparison**: Write and run test to prove that Hybrid Sort is faster than Merge Sort and Insertion Sort. Especially, several runs of computation of large, unsorted datasets must demonstrate improved runtime.

# Key Considerations for Hybridization

* + **Problem Characteristics:**Sorting is a good fit for hybridization, especially when the dataset characteristics (size, order) vary.
  + **Algorithm Complementarity:** Merge Sort is suitable for large arrays, while Insertion Sort is faster for smaller or nearly sorted sub-arrays.
  + **Hybrid Strategy:** A small modification and replacement of the Merge Sort algorithm with the Insertion Sort algorithm on the small sub-trees makes it more efficient.
  + **Parameter Tuning**: It is wise to try out different values of the trigger point in an attempt to get the best operating point
  + **Experimental Validation**: Thus, there is a need to evaluate the effectiveness of the proposed hybrid algorithm using different datasets and establish performance improvements systematically.