**NAME:** HAMMAD AHMAD **SAPID:** 46874

GITHUB LINK: https://github.com/Destroyer2134/AnalysisOfAlgo\_Assign04.git

**HYBRID OF MERGE SORT AND INSERTION SORT:**

**MERGE SORT:**

**Merge Sort** is a classic, efficient, and stable sorting algorithm based on the *divide-and-conquer* paradigm. It recursively splits the array into smaller subarrays until each subarray has only one element, then merges these sorted subarrays to produce a fully sorted array. Merge Sort is particularly known for its predictable O(nlogn) time complexity and is useful for handling large datasets.

**How Merge Sort Works**

1. **Divide**: The array is recursively split into two halves. This continues until each subarray contains only a single element, which is inherently sorted.
2. **Conquer**: Once the array is divided into individual elements, each pair of elements (or subarrays) is merged back together in sorted order. The merge process continues, combining sorted subarrays into increasingly larger sorted arrays.
3. **Combine**: Merging of subarrays continues until the original array is reassembled in sorted order.

**Steps of Merge Sort Example**

Let’s say we want to sort the array [38, 27, 43, 3, 9, 82, 10].

1. **Divide the Array**:
   * Split into two halves: [38, 27, 43] and [3, 9, 82, 10].
   * Further split each half until we reach single elements: [38], [27], [43], [3], [9], [82], [10].
2. **Merge Sorted Elements**:
   * Begin merging single elements back together in sorted order:
     + Merge [38] and [27] to get [27, 38].
     + Merge [27, 38] with [43] to get [27, 38, 43].
     + Merge [3] and [9] to get [3, 9].
     + Merge [3, 9] with [10, 82] to get [3, 9, 10, 82].
3. **Combine Final Halves**:
   * Finally, merge the two halves: [27, 38, 43] and [3, 9, 10, 82] to get the fully sorted array [3, 9, 10, 27, 38, 43, 82].

**Key Characteristics of Merge Sort**

* **Time Complexity**: Merge Sort operates in O(nlogn) time complexity in the best, average, and worst cases, making it reliable for large datasets.
* **Space Complexity**: Merge Sort requires O(n) additional space due to the temporary arrays used in the merging process.
* **Stability**: It is a stable sort, meaning it preserves the relative order of equal elements, which can be important in certain applications.
* **Recursive Nature**: Merge Sort uses recursion, and understanding its process helps build a strong foundation for learning other recursive algorithms.

**INSERTION SORT**:

**Insertion Sort** is a simple, intuitive, and stable sorting algorithm often used for small or nearly sorted datasets. It works similarly to how people might sort playing cards in their hands: starting with an empty left hand and picking up one card at a time, inserting each card into its correct position among the already sorted cards.

**How Insertion Sort Works**

Insertion Sort iterates through an array, picking one element (the key) at a time, and places it in the correct position relative to the elements before it. This process shifts the elements that are larger than the key to the right until the key can be inserted in its sorted position.

1. **Initialize**: Start with the second element of the array, treating the first element as a sorted portion.
2. **Pick the Key**: For each element from the second position onward, consider it the "key" that needs to be inserted into the sorted portion on its left.
3. **Shift and Insert**:
   * Move all elements larger than the key one position to the right.
   * Place the key in the correct position in the sorted portion of the array.
4. **Repeat**: Continue this process until the entire array is sorted.

**Steps of Insertion Sort Example**

Let’s sort the array [12, 11, 13, 5, 6]:

1. Start with the first element [12] (sorted portion) and take the second element 11 as the key.
2. Insert 11 into the correct position in the sorted portion [11, 12].
3. Next, take 13 as the key. It's already in the correct position, so no changes are made.
4. Take 5 as the key, shift [11, 12, 13] to the right, and insert 5 at the beginning: [5, 11, 12, 13].
5. Finally, take 6 as the key, shift [11, 12, 13] to the right, and place 6 in its sorted position: [5, 6, 11, 12, 13].

The array is now fully sorted.

**Key Characteristics of Insertion Sort**

* **Time Complexity**:
  + **Best Case**: O(n), if the array is already sorted (only one pass is needed).
  + **Average and Worst Case**: O(n^2), as each element could need to be compared with every other element in the sorted portion.
* **Space Complexity**: Insertion Sort is in-place, requiring only O(1) additional space.
* **Stability**: Insertion Sort is stable, preserving the relative order of equal elements.
* **Adaptability**: It is adaptive in that it performs very efficiently on nearly sorted data, which is why it’s often used as a final step in hybrid algorithms for sorting small subarrays (like in Timsort or Merge-Insertion hybrids).
* **Practical Use Cases**:
  + Small datasets where its simplicity and low overhead outweigh its O(n^2) complexity.
  + Nearly sorted data, where Insertion Sort can achieve almost linear performance.
  + As a helper algorithm within other sorting methods (e.g., in Timsort or Hybrid Sorts) to handle small partitions.

**HYBRID ALGORITHM:**

#### **Step 1: Problem Selection and Hybrid Strategy**

The problem chosen is **sorting an array**. The hybrid algorithm will:

1. Use Merge Sort to split and recursively sort large portions of the array.
2. Switch to Insertion Sort for small subarrays (e.g., subarrays with fewer than 10 elements) since it’s more efficient for small or nearly sorted data.

**Step 2: Implementing the Hybrid Algorithm**

import time

import random

# Sorting algorithms as previously defined

def insertion\_sort(arr, left, right):

for i in range(left + 1, right + 1):

key = arr[i]

j = i - 1

while j >= left and arr[j] > key:

arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = key

def merge(arr, left, mid, right):

n1 = mid - left + 1

n2 = right - mid

left\_half = arr[left:left + n1]

right\_half = arr[mid + 1:mid + 1 + n2]

i = j = 0

k = left

while i < n1 and j < n2:

if left\_half[i] <= right\_half[j]:

arr[k] = left\_half[i]

i += 1

else:

arr[k] = right\_half[j]

j += 1

k += 1

while i < n1:

arr[k] = left\_half[i]

i += 1

k += 1

while j < n2:

arr[k] = right\_half[j]

j += 1

k += 1

def merge\_sort(arr, left, right):

if left < right:

mid = (left + right) // 2

merge\_sort(arr, left, mid)

merge\_sort(arr, mid + 1, right)

merge(arr, left, mid, right)

def hybrid\_sort(arr, left, right, threshold=10):

if right - left + 1 <= threshold:

insertion\_sort(arr, left, right)

else:

if left < right:

mid = (left + right) // 2

hybrid\_sort(arr, left, mid, threshold)

hybrid\_sort(arr, mid + 1, right, threshold)

merge(arr, left, mid, right)

# Experimental comparison of algorithms

array\_sizes = [100, 1000, 5000, 10000, 20000]

results = {'Array Size': array\_sizes, 'Merge Sort': [], 'Insertion Sort': [], 'Hybrid Sort': []}

for size in array\_sizes:

arr = [random.randint(0, 10000) for \_ in range(size)]

arr\_merge, arr\_insertion, arr\_hybrid = arr[:], arr[:], arr[:]

# Measure time for Merge Sort

start\_time = time.time()

merge\_sort(arr\_merge, 0, len(arr\_merge) - 1)

results['Merge Sort'].append(time.time() - start\_time)

# Measure time for Insertion Sort

start\_time = time.time()

insertion\_sort(arr\_insertion, 0, len(arr\_insertion) - 1)

results['Insertion Sort'].append(time.time() - start\_time)

# Measure time for Hybrid Sort

start\_time = time.time()

hybrid\_sort(arr\_hybrid, 0, len(arr\_hybrid) - 1, threshold=10)

results['Hybrid Sort'].append(time.time() - start\_time)

# Display the results as a table

Results

**ANALYSIS:**

|  |  |  |  |
| --- | --- | --- | --- |
| Array Size | Merge Sort (s) | Insertion Sort (s) | Hybrid Sort (s) |
| 100 | 0.00017 | 0.00023 | 0.00012 |
| 1000 | 0.00303 | 0.03312 | 0.00130 |
| 5000 | 0.02243 | 0.90815 | 0.01136 |
| 10000 | 0.03411 | 2.46181 | 0.01570 |
| 20000 | 0.04812 | 9.15344 | 0.04106 |

### Performance Analysis

#### **Theoretical Analysis**

The hybrid algorithm combines the best parts of Merge Sort and Insertion Sort to optimize for different scenarios, which affects both time and space complexities:

1. **Time Complexity**:
   * **Merge Sort**: Consistently has O(nlogn) complexity due to the recursive division of the array and merging steps, regardless of array size or data distribution.
   * **Insertion Sort**: Has O(n^2) complexity in the worst case but O(n)O(n)O(n) in nearly sorted data, making it suitable for small arrays.
   * **Hybrid Algorithm**:
     + For large arrays, the hybrid algorithm primarily follows Merge Sort's O(nlogn) complexity.
     + By switching to Insertion Sort for smaller subarrays, it reduces recursive overhead, as Insertion Sort can handle small, nearly sorted segments more efficiently.
     + The hybrid algorithm's time complexity remains O(nlogn) in the average and worst cases but reduces constant factors due to reduced recursion depth, making it faster in practice.
2. **Space Complexity**:
   * Merge Sort’s space complexity is O(n) due to additional arrays needed for merging.
   * Insertion Sort is in-place with O(1) space complexity.
   * **Hybrid Algorithm**: Since merging still requires extra space, the hybrid approach maintains O(n) space complexity. However, because it uses Insertion Sort for small subarrays, the overall memory usage can be lower, as fewer levels of recursion result in fewer temporary arrays.

**Summary of Expected Improvements**

In practice, the hybrid algorithm should exhibit:

* **Improved performance** for small and mid-sized arrays due to reduced recursive overhead and selective Insertion Sort usage.
* **Lower runtime constants** than pure Merge Sort, as Insertion Sort on small subarrays requires fewer comparisons and assignments.
* **Similar asymptotic complexity** to Merge Sort but with empirical efficiency gains, particularly noticeable on mixed or nearly sorted data.