**Nasir Hussain**
**43913**
**BS-Cy (5-1)**
**Assignment: 3**
**Instructor: Muhammad Usman Sharif**
**Analysis of Algorithms**

**Implement four sorting algorithms (Bubble Sort, Selection Sort, Merge Sort, and Quick Sort) on three different arrays of equal size.**

- Array 1: at which best case scenario applies.
- Array 2: at which average case scenario applies.
- Array 3: at which worst case scenario applies.

**For each sorting algorithm and array combination, measure the time taken to sort The array. You can use a built-in function or implement a custom function to calculate The execution time.**
**Analyze the results to compare the performance of the different sorting algorithms under various conditions.**

# Solution:

## Bubble sort:

## Code:

```
import copy
import time

def bubble_sort(arr):

    n = len(arr)
    comparisons = 0
    swaps = 0
    for i in range(n):
        swapped = False
        print(f"\nPass {i+1}:")
        for j in range(0, n - i - 1):
            comparisons += 1
            print(f"  Comparing {arr[j]} and {arr[j + 1]}")
            if arr[j] > arr[j + 1]:
                print(f"    Swapping {arr[j]} and {arr[j + 1]}")
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swaps += 1
                swapped = True
            else:
                print(f"    No swap needed for {arr[j]} and {arr[j + 1]}")
        if not swapped:
            print("  No swaps occurred in this pass. The array is already
```

```
sorted.")
            break
        print(f"  Array after pass {i+1}: {arr}")
    return {"comparisons": comparisons, "swaps": swaps}

def display_results(case_name, sorted_arr, time_taken, stats):

    print(f"\n=== {case_name} Case ===")
    print(f"Sorted Array: {sorted_arr}")
    print(f"Time Taken: {time_taken:.6f} seconds")
    print(f"Total Comparisons: {stats['comparisons']}")
    print(f"Total Swaps: {stats['swaps']}")

def main():
    # Define the arrays
    best_case = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    average_case = [3, 5, 1, 4, 2, 6, 9, 8, 10, 7]
    worst_case = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

    # Make deep copies to preserve original arrays
    best_case_sorted = copy.deepcopy(best_case)
    average_case_sorted = copy.deepcopy(average_case)
    worst_case_sorted = copy.deepcopy(worst_case)

    # Bubble Sort on Best Case
    print("Starting Bubble Sort on Best Case:")
    start_time = time.time()
    best_stats = bubble_sort(best_case_sorted)
    end_time = time.time()
    best_time = end_time - start_time
    display_results("Best", best_case_sorted, best_time, best_stats)

    # Bubble Sort on Average Case
    print("\nStarting Bubble Sort on Average Case:")
    start_time = time.time()
    average_stats = bubble_sort(average_case_sorted)
    end_time = time.time()
    average_time = end_time - start_time
    display_results("Average", average_case_sorted, average_time,
average_stats)

    # Bubble Sort on Worst Case
    print("\nStarting Bubble Sort on Worst Case:")
    start_time = time.time()
    worst_stats = bubble_sort(worst_case_sorted)
    end_time = time.time()
    worst_time = end_time - start_time
    display_results("Worst", worst_case_sorted, worst_time, worst_stats)

if __name__ == "__main__":
    main()
```

## Running:

```
=== Best Case ===
Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Time Taken: 0.000000 seconds
Total Comparisons: 9
Total Swaps: 0
```

```
=== Average Case ===
Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Time Taken: 0.001000 seconds
Total Comparisons: 30
Total Swaps: 10
```

```
=== Worst Case ===
Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Time Taken: 0.003775 seconds
Total Comparisons: 45
Total Swaps: 45
```

## TIME COMPEXITY:

## Visual Representation of Time Complexity

1. **Best Case (Already Sorted):**
   - **Passes Needed:** 1
   - **Comparisons:** n-1
   - **Swaps:** 0
   - **Time Complexity:** O(n)
2. **Average Case (Random Order):**
   - **Passes Needed:** Approximately n/2
   - **Comparisons:** $(n^2)/2$
   - **Swaps:** Depends on initial order
   - **Time Complexity:** $O(n^2)$
3. **Worst Case (Reverse Sorted):**
   - **Passes Needed:** n-1
   - **Comparisons:** $(n^2 - n)/2$
   - **Swaps:** $(n^2 - n)/2$

    - **Time Complexity:** $O(n^2)$

## Practical Implications

- **Small Datasets:**
    - Bubble Sort can be acceptable for small arrays due to its simplicity.
- **Large Datasets:**
    - Inefficient due to $O(n^2)$ time complexity.
    - Better sorting algorithms (like Merge Sort, Quick Sort, or Heap Sort) should be used for scalability.
- **Nearly Sorted Data:**
    - Optimized Bubble Sort performs well ($O(n)$) because it can terminate early if no swaps are needed.

## Conclusion:

- **Best Case:** $O(n)$ — Efficient for already sorted arrays due to early termination.
- **Average Case:** $O(n^2)$ — Quadratic time complexity makes it inefficient for larger, randomly ordered arrays.
- **Worst Case:** $O(n^2)$ — highly inefficient for reverse-sorted arrays with maximum swaps and comparisons.

### Summary of Time Complexities

| Component | Operation | Time Complexity |
|---|---|---|
| Imports | Importing modules | $O(1)$ |
| Function Definitions | Defining `bubble_sort`, `display_results`, `main` | $O(1)$ each |
| `bubble_sort` Function | Entire Bubble Sort Algorithm | $O(n^2)$ |
| `display_results` | Displaying results | $O(n)$ |
| `main` Function | Sorting Best, Average, Worst Cases | $O(n^2)$ |
| Execution Check | Running `main` | $O(1)$ |

# SELECTION SORT:

## Code:

```
import copy
import time


def selection_sort(arr):

    n = len(arr)
    comparisons = 0
    swaps = 0

    for i in range(n):
        min_idx = i
        print(f"\nSelection Pass {i + 1}:")

        for j in range(i + 1, n):
            comparisons += 1
            print(f"  Comparing {arr[min_idx]} and {arr[j]}")
            if arr[j] < arr[min_idx]:
                print(f"    New minimum found: {arr[j]} (was
{arr[min_idx]})")
                min_idx = j

        if min_idx != i:
            print(f"  Swapping {arr[i]} and {arr[min_idx]}")
            arr[i], arr[min_idx] = arr[min_idx], arr[i]
            swaps += 1
        else:
            print(f"  No swap needed for index {i} ({arr[i]})")

        print(f"  Array after pass {i + 1}: {arr}")

    return {"comparisons": comparisons, "swaps": swaps}


def display_results(case_name, sorted_arr, time_taken, stats):

    print(f"\n=== {case_name} Case ===")  # Step A: Header display (O(1))
    print(f"Sorted Array: {sorted_arr}")  # Step B: Display sorted array
(O(n))
    print(f"Time Taken: {time_taken:.6f} seconds")  # Step C: Display time
taken (O(1))
    print(f"Total Comparisons: {stats['comparisons']}")  # Step D: Display
comparisons (O(1))
    print(f"Total Swaps: {stats['swaps']}")  # Step E: Display swaps (O(1))


def main():
    # Define the arrays
    best_case = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    average_case = [3, 5, 1, 4, 2, 6, 9, 8, 10, 7]
    worst_case = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```python
    # Make deep copies to preserve original arrays
    best_case_sorted = copy.deepcopy(best_case)
    average_case_sorted = copy.deepcopy(average_case)
    worst_case_sorted = copy.deepcopy(worst_case)

    # Selection Sort on Best Case
    print("Starting Selection Sort on Best Case:")
    start_time = time.time()
    best_stats = selection_sort(best_case_sorted)
    end_time = time.time()
    best_time = end_time - start_time
    display_results("Best", best_case_sorted, best_time, best_stats)


    print("\nStarting Selection Sort on Average Case:")
    start_time = time.time()
    average_stats = selection_sort(average_case_sorted)
    end_time = time.time()
    average_time = end_time - start_time
    display_results("Average", average_case_sorted, average_time,
average_stats)  # Step XIII: Display results (O(n))

    # Selection Sort on Worst Case
    print("\nStarting Selection Sort on Worst Case:")
    start_time = time.time()
    worst_stats = selection_sort(worst_case_sorted)
    end_time = time.time()
    worst_time = end_time - start_time
    display_results("Worst", worst_case_sorted, worst_time, worst_stats)


if __name__ == "__main__":
    main()
```

## RUNNING:

```
=== Best Case ===
Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Time Taken: 0.000000 seconds
Total Comparisons: 45
Total Swaps: 0
```

```
=== Average Case ===
Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Time Taken: 0.003679 seconds
Total Comparisons: 45
Total Swaps: 4
```

```
=== Worst Case ===
Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Time Taken: 0.000926 seconds
Total Comparisons: 45
Total Swaps: 5
```

# TIME COMPLEXITY:

## Visual Representation of Time Complexity

1. **Best Case (Already Sorted):**
    - **Passes Needed:** n
    - **Comparisons:** (n*(n-1))/2
    - **Swaps:** 0 (if no new minimum is found, but in this implementation, a swap is only performed if a new minimum is found. Since the array is already sorted, no swaps occur.)
    - **Time Complexity:** $O(n^2)$
2. **Average Case (Random Order):**
    - **Passes Needed:** n
    - **Comparisons:** (n*(n-1))/2
    - **Swaps:** Approximately n (one swap per pass)
    - **Time Complexity:** $O(n^2)$
3. **Worst Case (Reverse Sorted):**
    - **Passes Needed:** n
    - **Comparisons:** (n*(n-1))/2
    - **Swaps:** n (one swap per pass)
    - **Time Complexity:** $O(n^2)$

## Practical Implications:

- **Small Datasets:**
    - Selection Sort can be acceptable for small arrays due to its simplicity and predictability.
- **Large Datasets:**
    - Inefficient due to $O(n^2)$ time complexity.
    - Better sorting algorithms (like Merge Sort, Quick Sort, or Heap Sort) should be used for scalability.
- **Memory Usage:**

- Selection Sort is an in-place sorting algorithm with O (1) additional memory, making it memory-efficient.

- **Stable Sorting:**
  - Standard Selection Sort is not stable. However, with modifications, it can be made stable.

# CONCLUSION:

- **Best Case: O ($n^2$)** — No improvement even if the array is already sorted.
- **Average Case: O ($n^2$)** — consistently quadratic time complexity across all scenarios.
- **Worst Case: O ($n^2$)** — Maintains quadratic time complexity regardless of input.

### Summary of Time Complexities

| Component | Operation | Time Complexity |
|---|---|---|
| Imports | Importing modules | O(1) |
| Function Definitions | Defining `selection_sort`, `display_results`, `main` | O(1) each |
| `selection_sort` | Entire Selection Sort Algorithm | O($n^2$) |
| `display_results` | Displaying results | O(n) |
| `main` Function | Sorting Best, Average, Worst Cases | O($n^2$) |
| Execution Check | Running `main` | O(1) |

# MERGE SORT:

# CODE:

```
import copy
import time


def merge_sort(arr):
    comparisons = 0
    merges = 0

    def _merge_sort(arr, left, right):
        nonlocal comparisons, merges
        if left < right:
            mid = (left + right) // 2
            _merge_sort(arr, left, mid)
            _merge_sort(arr, mid + 1, right)
            comparisons, merges = merge(arr, left, mid, right, comparisons,
merges)
```

```python
    def merge(arr, left, mid, right, comparisons, merges):
        L = arr[left:mid + 1]
        R = arr[mid + 1:right + 1]

        i = j = 0
        k = left

        print(f"Merging subarrays {L} and {R}")

        while i < len(L) and j < len(R):
            comparisons += 1
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
            merges += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
            merges += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
            merges += 1

        print(f"Array after merging: {arr[left:right + 1]}")
        return comparisons, merges

    _merge_sort(arr, 0, len(arr) - 1)
    return {"comparisons": comparisons, "merges": merges}


def display_results(case_name, sorted_arr, time_taken, stats):
    print(f"\n=== {case_name} Case ===")
    print(f"Sorted Array: {sorted_arr}")
    print(f"Time Taken: {time_taken:.6f} seconds")
    print(f"Total Comparisons: {stats['comparisons']}")
    print(f"Total Merges: {stats['merges']}")


def main():
    best_case = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    average_case = [3, 5, 1, 4, 2, 6, 9, 8, 10, 7]
    worst_case = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

    best_case_sorted = copy.deepcopy(best_case)
    average_case_sorted = copy.deepcopy(average_case)
    worst_case_sorted = copy.deepcopy(worst_case)
```

```
    print("Starting Merge Sort on Best Case:")
    start_time = time.time()
    best_stats = merge_sort(best_case_sorted)
    end_time = time.time()
    best_time = end_time - start_time
    display_results("Best", best_case_sorted, best_time, best_stats)

    print("\nStarting Merge Sort on Average Case:")
    start_time = time.time()
    average_stats = merge_sort(average_case_sorted)
    end_time = time.time()
    average_time = end_time - start_time
    display_results("Average", average_case_sorted, average_time,
average_stats)

    print("\nStarting Merge Sort on Worst Case:")
    start_time = time.time()
    worst_stats = merge_sort(worst_case_sorted)
    end_time = time.time()
    worst_time = end_time - start_time
    display_results("Worst", worst_case_sorted, worst_time, worst_stats)


if __name__ == "__main__":
    main()
```

## Running:

```
=== Best Case ===
Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Time Taken: 0.000000 seconds
Total Comparisons: 19
Total Merges: 34
```

```
=== Average Case ===
Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Time Taken: 0.000000 seconds
Total Comparisons: 20
Total Merges: 34
```

```
=== Worst Case ===
Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Time Taken: 0.000000 seconds
Total Comparisons: 15
Total Merges: 34
```

# TIME COMPLEXITY:

## Detailed Explanation of Merge Sort Time Complexity:

Merge Sort is a divide-and-conquer algorithm that divides the input array into two halves, recursively sorts them, and then merges the sorted halves. Here's a more granular breakdown of its time complexity:

1. **Dividing the Array:**

   - **Operation:** The array is repeatedly divided into halves until each subarray contains a single element.
   - **Number of Divisions:** $\log_2 (n)$ divisions are needed to break the array down to single elements.
   - **Time Complexity:** O(log n)

2. **Merging the Arrays:**

   - **Operation:** Two sorted subarrays are merged into a single sorted array.
   - **Number of Merge Operations:** Each level of division requires merging n elements in total.
   - **Time Complexity per Merge Level:** O(n)
   - **Total Time for Merging Across All Levels:** O(n log n)

3. **Comparisons During Merge:**

   - **Operation:** Each element is compared once during the merge process.
   - **Time Complexity:** O(n) per merge level
   - **Total Comparisons:** O(n log n)

4. **Overall Time Complexity:**

   - **Combining Dividing and Merging:** O(n log n) + O(n log n) = O(n log n)
   - **Explanation:** The algorithm consistently performs in O (n log n) time across all cases (best, average, worst) because the number of divisions and the merging process are unaffected by the initial order of elements.

5. **Space Complexity:**

   - **Operation:** Merge Sort requires additional space for the temporary arrays during the merge process.
   - **Space Complexity:** O(n)
   - **Explanation:** Although the time complexity is efficient, Merge Sort uses extra memory proportional to the input size.

6. **Stability:**

    - **Operation:** Merge Sort is a stable sorting algorithm.
    - **Explanation:** It preserves the relative order of equal elements, which is beneficial in scenarios where the order carries meaning.

## Visual Representation of Time Complexity

1. **Best Case (Already Sorted):**

    - **Passes Needed:** $\log_2(n)$
    - **Comparisons:** O(n log n)
    - **Merges:** O(n log n)
    - **Time Complexity:** O(n log n)

2. **Average Case (Random Order):**

    - **Passes Needed:** $\log_2(n)$
    - **Comparisons:** O(n log n)
    - **Merges:** O(n log n)
    - **Time Complexity:** O(n log n)

3. **Worst Case (Reverse Sorted):**

    - **Passes Needed:** $\log_2(n)$
    - **Comparisons:** O(n log n)
    - **Merges:** O(n log n)
    - **Time Complexity:** O(n log n)

## Practical Implications

- **Large Datasets:**
    - **Advantage:** Efficient sorting with O (n log n) time complexity.
    - **Disadvantage:** Requires additional memory for temporary arrays.
- **Small Datasets:**
    - **Advantage:** Still efficient, but simpler algorithms like Insertion Sort might perform better due to lower constant factors.
- **Memory Constraints:**
    - **Consideration:** Since Merge Sort requires additional space, it may not be suitable for systems with limited memory.
- **Stability Requirement:**
    - **Advantage:** Being a stable sort, Merge Sort is ideal when the stability of elements matters.

# Conclusion:

- **Merge Sort Characteristics:**
- **Time Complexity:** O (n log n) across all cases.
- **Space Complexity:** O (n) due to additional temporary arrays.
- **Stability:** Merge Sort is stable by default.
- **Efficiency:** Highly efficient for large datasets compared to simple algorithms like Bubble Sort or Selection Sort.

- **Implementation Highlights:**

- **Recursive Division:** The array is recursively divided into halves until single-element subarrays are reached.
- **Merging Process:** Sorted subarrays are merged back together in a manner that maintains order.
- **Counting Operations:** The implementation counts the number of comparisons and merges to provide insight into the algorithm's performance.

- **Performance Metrics:**

- **Comparisons:** Reflects the number of element comparisons made during sorting.
- **Merges:** Indicates the number of times elements are merged into the main array.
- **Time Taken:** Measures the actual time consumed by the sorting process.

- **Best Practices:**

- **Deep Copies:** Making deep copies of the original arrays ensures that each sorting test starts with the unsorted data.
- **Time Measurement:** Using the `time` module to measure the duration of sorting provides empirical evidence of the algorithm's efficiency.
- **Detailed Logging:** Printing the state of the array after each merge helps in understanding how the algorithm progresses.

### Summary of Time Complexities

| Component | Operation | Time Complexity |
|---|---|---|
| Imports | Importing modules | O(1) |
| Function Definitions | Defining `merge_sort`, `display_results`, `main` | O(1) each |
| `merge_sort` | Entire Merge Sort Algorithm | O(n log n) |
| `display_results` | Displaying results | O(n) |
| `main` Function | Sorting Best, Average, Worst Cases | O(n log n) |
| Execution Check | Running `main` | O(1) |

# QUICK SORT:

## Code:

```
import copy
import time


def quick_sort(arr):
    comparisons = 0
    swaps = 0

    def _quick_sort(arr, low, high):
        nonlocal comparisons, swaps
        if low < high:
            pi, comparisons, swaps = partition(arr, low, high, comparisons,
swaps)
            _quick_sort(arr, low, pi - 1)
            _quick_sort(arr, pi + 1, high)

    def partition(arr, low, high, comparisons, swaps):
        pivot = arr[high]
        i = low - 1
        print(f"Partitioning with pivot: {pivot} in subarray {arr[low:high +
1]}")

        for j in range(low, high):
            comparisons += 1
            print(f"  Comparing {arr[j]} with pivot {pivot}")
            if arr[j] <= pivot:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]
                swaps += 1
                print(f"    Swapping {arr[j]} and {arr[i]}")

        arr[i + 1], arr[high] = arr[high], arr[i + 1]
        swaps += 1
        print(f"  Swapping pivot {arr[high]} with {arr[i + 1]}")
        print(f"  Partition index: {i + 1}")
        print(f"  Array after partitioning: {arr}")
        return i + 1, comparisons, swaps

    _quick_sort(arr, 0, len(arr) - 1)
    return {"comparisons": comparisons, "swaps": swaps}


def display_results(case_name, sorted_arr, time_taken, stats):
    print(f"\n=== {case_name} Case ===")
    print(f"Sorted Array: {sorted_arr}")
    print(f"Time Taken: {time_taken:.6f} seconds")
    print(f"Total Comparisons: {stats['comparisons']}")
    print(f"Total Swaps: {stats['swaps']}")


def main():
    best_case = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  # Best Case: Already sorted
```

```
    average_case = [3, 5, 1, 4, 2, 6, 9, 8, 10, 7]  # Average Case: Random
order
    worst_case = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]  # Worst Case: Reverse
sorted

    best_case_sorted = copy.deepcopy(best_case)
    average_case_sorted = copy.deepcopy(average_case)
    worst_case_sorted = copy.deepcopy(worst_case)

    print("Starting Quick Sort on Best Case:")
    start_time = time.time()
    best_stats = quick_sort(best_case_sorted)
    end_time = time.time()
    best_time = end_time - start_time
    display_results("Best", best_case_sorted, best_time, best_stats)

    print("\nStarting Quick Sort on Average Case:")
    start_time = time.time()
    average_stats = quick_sort(average_case_sorted)
    end_time = time.time()
    average_time = end_time - start_time
    display_results("Average", average_case_sorted, average_time,
average_stats)

    print("\nStarting Quick Sort on Worst Case:")
    start_time = time.time()
    worst_stats = quick_sort(worst_case_sorted)
    end_time = time.time()
    worst_time = end_time - start_time
    display_results("Worst", worst_case_sorted, worst_time, worst_stats)


if __name__ == "__main__":
    main()
```

## Running:

```
=== Best Case ===
Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Time Taken: 0.000000 seconds
Total Comparisons: 45
Total Swaps: 54
```

```
=== Average Case ===
Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Time Taken: 0.016070 seconds
Total Comparisons: 23
Total Swaps: 22
```

```
=== Worst Case ===
Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Time Taken: 0.001104 seconds
Total Comparisons: 45
Total Swaps: 29
```

# TIME COMPLEXITY:

## Detailed Explanation of Quick Sort Time Complexity

Quick Sort is a highly efficient sorting algorithm based on the divide-and-conquer paradigm. Here's a more granular breakdown of its time complexity:

1. **Choosing a Pivot:**
    - **Operation:** Select a pivot element from the array.
    - **Time Complexity:** O (1) if choosing the last element as pivot, as in the provided implementation.
2. **Partitioning the Array:**
    - **Operation:** Rearrange elements in the array such that elements less than or equal to the pivot are on the left, and elements greater than the pivot are on the right.
    - **Time Complexity:** O (n), where n is the number of elements in the current subarray.
    - **Explanation:** Each element is compared to the pivot once.
3. **Recursive Sorting of Subarrays:**
    - **Operation:** Recursively apply Quick Sort to the subarrays formed by partitioning.
    - **Time Complexity:** O (log n) levels of recursion on average.
    - **Explanation:** Each recursive call splits the array approximately in half, leading to a logarithmic number of levels.
4. **Overall Time Complexity:**
    - **Best Case:** O(n log n)
        - **Explanation:** Balanced partitions lead to logarithmic recursion depth with linear work at each level.
    - **Average Case:** O(n log n)
        - **Explanation:** Even with some unbalanced partitions, the overall time remains logarithmic multiplied by linear.
    - **Worst Case:** $O(n^2)$
        - **Explanation:** Highly unbalanced partitions (e.g., when the smallest or largest element is always chosen as pivot) lead to linear recursion depth with linear work at each level.
5. **Space Complexity:**
    - **Operation:** Requires O (log n) space for the recursion stack on average.
    - **Explanation:** Each recursive call adds a layer to the stack, and with balanced partitions, the depth is logarithmic.

6. **Stability:**
    - **Operation:** Quick Sort is not a stable sort by default.
    - **Explanation:** Equal elements may not retain their original order after sorting.
7. **Optimizations to Prevent Worst Case:**
    - **Randomized Pivot Selection:** Choosing a random pivot reduces the probability of encountering the worst case.
    - **Median-of-Three Pivot Selection:** Choosing the median of the first, middle, and last elements as pivot can lead to more balanced partitions.

## Practical Implications

- **Large Datasets:**
    - **Advantage:** Quick Sort is generally faster in practice compared to other O(n log n) algorithms like Merge Sort, especially due to better cache performance and in-place sorting.
    - **Disadvantage:** Vulnerable to worst-case performance without optimizations.
- **Small Datasets:**
    - **Advantage:** Efficient and quick to sort.
    - **Additional Note:** Sometimes, insertion sort is used for very small subarrays to optimize performance further.
- **Memory Usage:**
    - **Advantage:** In-place sorting with O (log n) additional space for the recursion stack.
    - **Disadvantage:** Merge Sort requires O (n) additional space.
- **Stability Requirement:**
    - **Consideration:** If stability is required, Quick Sort may not be suitable unless modified to be stable.

## For Best Case:

- **Time Complexity:**

  - **quick_sort(best_case_sorted):** O(n log n) on average
  - **display_results:** O(n)

- **Overall:** O (n log n) + O (n) = O (n log n)

## Best Case Average:

- **Time Complexity:**

  - **quick_sort(average_case_sorted):** O(n log n) on average
  - **display_results:** O(n)

- **Overall:** O (n log n) + O (n) = O (n log n)

## Worst Case:

- **Time Complexity:**

  - **quick_sort(worst_case_sorted):** $O(n^2)$ in the worst case
  - **display_results:** O(n)

- **Overall:** $O(n^2) + O(n) = O(n^2)$

### Summary of Time Complexities

| Component | Operation | Time Complexity |
|---|---|---|
| Imports | Importing modules | O(1) |
| Function Definitions | Defining `quick_sort`, `display_results`, `main` | O(1) each |
| `quick_sort` | Entire Quick Sort Algorithm | O(n log n) on average, $O(n^2)$ worst case |
| `display_results` | Displaying results | O(n) |
| `main` Function | Sorting Best, Average, Worst Cases | O(n log n) on average, $O(n^2)$ worst case |
| Execution Check | Running `main` | O(1) |

## Conclusion

The provided Python script offers a comprehensive exploration of the Quick Sort algorithm across best, average, and worst-case scenarios. By incorporating detailed logging, performance metrics, and structured output, it serves as both an educational tool and a practical means of analyzing the algorithm's behavior under various conditions.

### Key Points:

- **Quick Sort Characteristics:**
  - **Time Complexity:** O (n log n) on average, $O(n^2)$ in the worst case.
  - **Space Complexity:** O (log n) due to recursion stack.
  - **Stability:** Not stable by default.
  - **Efficiency:** Highly efficient for large datasets with appropriate pivot selection.

- **Implementation Highlights:**
  - **Recursive Division:** The array is recursively divided into partitions based on the pivot.
  - **Partitioning Process:** Elements are rearranged around the pivot to ensure left <= pivot < right.
  - **Counting Operations:** The implementation tracks the number of comparisons and swaps to provide insight into the algorithm's performance.
- **Performance Metrics:**
  - **Comparisons:** Reflects the number of element comparisons made during sorting.
  - **Swaps:** Indicates the number of times elements are swapped.
  - **Time Taken:** Measures the actual time consumed by the sorting process.
- **Best Practices:**
  - **Deep Copies:** Making deep copies of the original arrays ensures that each sorting test starts with the unsorted data.
  - **Time Measurement:** Using the `time` module to measure the duration of sorting provides empirical evidence of the algorithm's efficiency.
  - **Detailed Logging:** Printing the state of the array after each partition helps in understanding how the algorithm progresses.