1. Implementation

- Implement the Quicksort algorithm using Python. Your implementation should be clear, efficient, and correctly follow the steps for selecting a pivot, partitioning the array, and recursively sorting the subarrays.

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr  # Base case: a single-element or empty array is already sorted

    pivot = arr[len(arr) // 2]  # Choosing the middle element as the pivot
    left = [x for x in arr if x < pivot]  # Elements less than pivot
    middle = [x for x in arr if x == pivot]  # Elements equal to pivot
    right = [x for x in arr if x > pivot]  # Elements greater than pivot

    return quicksort(left) + middle + quicksort(right)  # Recursively sorting and combining

# Example usage
data = [3, 6, 8, 10, 1, 2, 1]
sorted_data = quicksort(data)
print(sorted_data)
```

2. Performance Analysis

- Provide a detailed analysis of the time complexity of Quicksort in the best, average, and worst cases.

- **Best Case: O(nlogn)**
  - The best case occurs when the pivot divides the array into two nearly equal halves at each step.
  - Since each level of recursion splits the array in half, there are approximately nlogn levels.
  - At each level, we perform O(n) work to partition the array.
  - Therefore, the total time complexity is: O(n)+O(n)+⋯+O(n)=O(nlogn)
- **Average Case: O(nlogn)**
  - On average, the pivot selection divides the array into reasonably balanced partitions.
  - The recurrence relation is similar to the best case: T(n)=2T(n/2)+O(n)
  - Solving this recurrence using the Master Theorem gives: O(nlogn)
- **Worst Case: O(n^2)**
  - The worst case occurs when the pivot is always the smallest or largest element, leading to highly unbalanced partitions (one subarray has n−1 elements, the other has 0).
  - The recurrence relation in this case becomes: T(n)=T(n−1)+O(n)
  - Expanding this, we get: O(n)+O(n−1)+O(n−2)+⋯+O(1)=O(n^2)
  - This can happen in an already sorted or reverse-sorted array if the first or last element is chosen as the pivot.

- **Explain why the average-case time complexity is \(O(n \log n)\) and the worst-case time complexity is \(O(n^2)\).**
    - **average-case time complexity is \(O(n \log n)\)**
        - On average, the pivot divides the array into two roughly equal halves (i.e., one partition gets about n/2 elements).
        - This means that each recursive call reduces the problem size by half.
        - The recurrence relation for this case is:

$$T(n)=2T(n/2)+O(n)$$

$$=2(2T(n/4)+O(n/2))+O(n)$$
$$=4T(n/4)+2O(n/2)+O(n)$$
$$=8T(n/8)+4O(n/4)+2O(n/2)+O(n)$$

Continue expanding until the subproblems reach size O(1) (base case):

- The number of levels in the recursion tree is O(logn) (since n gets halved each time).
- Each level performs O(n) work for partitioning.

Since there are O(logn) levels, and each level does O(n) work, the total time complexity is:
O(nlogn)

In conclusion,The pivot on average creates well-balanced partitions.The depth of the recursion tree is O(logn). Each level requires O(n) work. Final complexity: O(nlogn).

- **the worst-case time complexity is \(O(n^2)\)**

The worst case occurs when the pivot always creates the most unbalanced partitions. This happens when:

- The pivot is always the smallest or largest element.
- One partition has n−1 elements, and the other has 0.

For example, if we always pick the first or last element as the pivot in a sorted or reverse-sorted array, we get:

$$T(n)=T(n-1)+O(n)$$

Expand the recurrence:
$$T(n)=T(n-1)+O(n)$$

$=(T(n-2)+O(n-1))+O(n)$

$=(T(n-3)+O(n-2))+O(n-1)+O(n)...$

$=T(1)+O(2)+O(3)+...+O(n)$

$=O(1+2+3+...+n)$

$=O(n^2)$

In conclusion, each recursive call reduces the problem size by only 1. The recursion tree has O(n) levels. Each level requires O(n) work. Final complexity: O(n^2).

- Discuss the space complexity and any additional overheads associated with the algorithm.

Best and Average Case: O(logn) Space

When the pivot splits the array into two nearly equal halves, recursion depth is O(logn). Each recursive call adds a function to the call stack. Since there are at most O(logn) levels of recursion, space usage is O(logn).

Worst Case: O(n) Space

If the pivot always picks the smallest or largest element, the recursion tree becomes highly unbalanced. This results in a linear recursion depth of O(n), requiring O(n) stack space. Example: Sorting a sorted or reverse-sorted array with the last element as the pivot.

Additional overheads

Stack Overhead: Every recursive function call **stores local variables and return addresses**, leading to stack memory usage. In a **worst-case scenario**, this can lead to a **stack overflow** for large n.

3. Randomized Quicksort

- Implement a randomized version of Quicksort where the pivot is chosen randomly from the subarray being sorted.

```
import random

def randomized_quick_sort(arr, low, high):
    if low < high:
        # Pick a random pivot and swap with the last element
        pivot_index = random.randint(low, high)
        arr[pivot_index], arr[high] = arr[high], arr[pivot_index]

        # Partition the array around the random pivot
        pivot = partition(arr, low, high)

        # Recursively sort the partitions
        randomized_quick_sort(arr, low, pivot - 1)
        randomized_quick_sort(arr, pivot + 1, high)

def partition(arr, low, high):
    pivot = arr[high]  # Pivot is now at the end
    i = low - 1  # Pointer for smaller elements

    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]  # Swap elements to maintain order

    arr[i + 1], arr[high] = arr[high], arr[i + 1]  # Place pivot in correct position
    return i + 1

# Example Usage
arr = [10, 7, 8, 9, 1, 5]
randomized_quick_sort(arr, 0, len(arr) - 1)
print("Sorted array:", arr)
```

- Analyze how randomization affects the performance of Quicksort and reduces the likelihood of encountering the worst-case scenario.

In a standard Quicksort implementation where the pivot is chosen as the first or last element, the worst case occurs when the input is already sorted (ascending or descending). In such cases, the pivot always partitions the array in the most unbalanced way, leading to one subarray of size $n-1$ and the other of size 0. The recursion tree becomes highly unbalanced, leading to a recurrence relation:
$T(n)=T(n-1)+O(n)$

Expanding this results in:
$O(1 + 2 + 3 + ... + n) = O(n^2)$

Thus, standard Quicksort exhibits $O(n^2)$ time complexity in the worst case.

Randomized Quicksort improves performance by randomly selecting a pivot from the array before partitioning. This changes the pivot selection process from deterministic to probabilistic, meaning, the chance of selecting the smallest or largest element as the pivot repeatedly is exponentially low. The pivot will, on average, split the array into nearly equal halves.

Since the pivot is chosen randomly, each partition will typically have a balanced structure.

- The recurrence relation becomes:
  $T(n)=2T(n/2)+O(n)$

Expanding this results in:
O(nlogn)

Thus, with high probability, the recursive depth remains O(logn), ensuring an expected time complexity of O(nlogn).

Even though the worst case is still theoretically possible, the probability of encountering it is extremely low.

- In deterministic Quicksort, choosing the worst pivot happens every time for sorted input, leading to O(n^2).
- In Randomized Quicksort, the probability of picking a worst-case pivot at every step decreases exponentially.

Probability of worst-case at every level: $1/n \times 1/(n-1) \times 1/(n-2) \ldots \approx 1/(n!)$.

Since n! grows exponentially, the probability of hitting the worst case across multiple recursive levels is almost negligible.

4. Empirical Analysis

- Empirically compare the running time of the deterministic and randomized versions of Quicksort on different input sizes and distributions (e.g., random, sorted, reverse-sorted).

```
import random
import time

# Deterministic Quicksort (Pivot: Last Element)
def deterministic_quick_sort(arr, low, high):
    if low < high:
        pivot = partition(arr, low, high)
        deterministic_quick_sort(arr, low, pivot - 1)
        deterministic_quick_sort(arr, pivot + 1, high)

def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

# Randomized Quicksort
def randomized_quick_sort(arr, low, high):
    if low < high:
        pivot_index = random.randint(low, high)
        arr[pivot_index], arr[high] = arr[high], arr[pivot_index]  # Swap random pivot with last element
        pivot = partition(arr, low, high)
        randomized_quick_sort(arr, low, pivot - 1)
        randomized_quick_sort(arr, pivot + 1, high)

# Function to run and time sorting
def time_sorting(algorithm, arr):
    start = time.time()
    algorithm(arr, 0, len(arr) - 1)
    end = time.time()
    return end - start

# Generate test data
def generate_test_data(size, order="random"):
    arr = list(range(size))
    if order == "random":
        random.shuffle(arr)
    elif order == "reverse":
        arr.reverse()
    return arr

# Run experiments
sizes = [100, 500, 900]
orders = ["random", "sorted", "reverse"]

for size in sizes:
    print(f"\n--- Input Size: {size} ---")
    for order in orders:
        arr1 = generate_test_data(size, order)
        arr2 = list(arr1)  # Copy to ensure same input for both algorithms

        time_det = time_sorting(deterministic_quick_sort, arr1)
        time_rand = time_sorting(randomized_quick_sort, arr2)

        print(f"  {order.capitalize()} input: Deterministic = {time_det:.6f} sec, Randomized = 
{time_rand:.6f} sec")
```

- Discuss the observed results and relate them to your theoretical analysis.

Theoretically,
For random input, both versions should perform similarly (O(nlogn)).
For sorted or reverse-sorted input, deterministic Quicksort will degrade to O(n^2).
Randomized Quicksort should consistently perform at O(nlogn) in all cases.

Observed Results:

```
--- Input Size: 100 ---
  Random input: Deterministic = 0.000062 sec, Randomized = 0.000096 sec
  Sorted input: Deterministic = 0.000447 sec, Randomized = 0.000085 sec
  Reverse input: Deterministic = 0.000300 sec, Randomized = 0.000095 sec

--- Input Size: 500 ---
  Random input: Deterministic = 0.000380 sec, Randomized = 0.000558 sec
  Sorted input: Deterministic = 0.009426 sec, Randomized = 0.000524 sec
  Reverse input: Deterministic = 0.006871 sec, Randomized = 0.000538 sec

--- Input Size: 900 ---
  Random input: Deterministic = 0.000810 sec, Randomized = 0.000624 sec
  Sorted input: Deterministic = 0.030926 sec, Randomized = 0.001063 sec
  Reverse input: Deterministic = 0.022856 sec, Randomized = 0.001026 sec
```

For Random Input: Both deterministic and randomized Quicksort perform similarly.
For Sorted and Reverse-Sorted Input: Deterministic Quicksort slows down significantly. Randomized Quicksort remains efficient.