```
# Submitted by: Jaya Sandeep, Ketha
# Based on pseudo code in Textbook: Introduction To Algorithms (3rd Edition).
# References:  https://www.youtube.com/watch?v=jlHkDBEumP0
#              https://www.youtube.com/watch?v=QN9hnmAgmOc
#              https://www.youtube.com/watch?
v=ThWBE8Y83bw&list=PL6Zs6LgrJj3u57thS7K7yLPQb5nA23iVu&index=18
#              https://www.youtube.com/watch?
v=9qVESt5bBfQ&list=PL6Zs6LgrJj3u57thS7K7yLPQb5nA23iVu&index=26
#              https://www.youtube.com/watch?v=HKTyOUx9Wf4

import random
import time
import unittest
import matplotlib.pyplot as plt

#_____________________________________________________DETERMINISTIC QUICK
SORT_____________________________________________________________________#

#Defining partitioning function for deterministic quicksort.
def det_partition(my_array, my_lb, my_ub):

    #Choose start element from array as my pivot.
    my_pivot = my_array[my_lb]

    #Defining start  and end positions for traversing array.
    array_start = my_lb
    array_end = my_ub

    #Traversing array until array_start is less than array_end.
    while array_start < array_end:
        #Incrementing array_start by one everytime when element at array_start is
less than or equal to my_pivot value.
        while array_start <= my_ub and my_array[array_start] <= my_pivot:
            array_start = array_start + 1

        #Decrementing array_end by one everytime when element at array_end is
greater than my_pivot value.
        while my_array[array_end] > my_pivot:
            array_end = array_end - 1

        #Swap the elements at array_start with array_end only when array_start is
less than array_end for everyshift.
        if array_start < array_end:
            my_array[array_start], my_array[array_end] = my_array[array_end],
my_array[array_start]

    #Swap the elements at start with element at array_end, making elements to the
left of pivot to be less than pivot and elements to right
    #of pivot to be greater than pivot.
    my_array[my_lb], my_array[array_end] = my_array[array_end], my_array[my_lb]
    return array_end
#-------------------------------------------------------------------------------
-----------------------------------------------------------#
#Defining main deterministic quick sort function:
def det_qs(my_array):
    def _det_qs(my_array, my_lb, my_ub):
        #Implementing stack based approach to prevent maximum recursion depth
reached
        #Discussed this approach with student of this course Janaki Ram.
```

```
        stack = [(my_lb, my_ub)]
        while stack:
            my_lb, my_ub = stack.pop()
            if my_lb < my_ub:
                pivot_loc = det_partition(my_array, my_lb, my_ub)
                stack.append((my_lb, pivot_loc - 1))
                stack.append((pivot_loc + 1, my_ub))

    _det_qs(my_array, 0, len(my_array) - 1)

#_____________________________________________________________________________
_____________________________________#
#_____________________________________RANDOMIZED QUICK
SORT_____________________________________#

#Defining partitioning function for randomized quicksort.
def RQS_partition(my_array, my_lb, my_ub):
    #Choosing a random index position between my_lb and my_ub.
    pivot_index = random.randint(my_lb,my_ub)

    #Introduce randomness to given array by swapping start element with pivot
element(To avoid likelihood of worst-case scenario
    #of sorted array which gives O(n^2) time complexity).
    my_array [my_lb], my_array[pivot_index] = my_array[pivot_index],
my_array[my_lb]

    #Choose start element from array as my pivot.
    my_pivot = my_array[my_lb]

    #Defining start and end positions for traversing array.
    array_start = my_lb
    array_end = my_ub

    #Traversing array until array_start is less than array_end.
    while array_start < array_end:
        #Incrementing array_start by one when element at array_start is less than
or equal to my_pivot value.
        while array_start <= my_ub and my_array[array_start] <= my_pivot:
            array_start = array_start + 1

        #Decrementing array_end by one when element at array_end is greater than
my_pivot value.
        while array_start <= my_ub and my_array[array_end] > my_pivot:
            array_end = array_end - 1

        #Swap the elements at array_start with array_end only when array_start is
less than array_end for everyshift.
        if array_start < array_end:
            my_array[array_start], my_array[array_end] = my_array[array_end],
my_array[array_start]

    #Swap the elements at start with element at array_end, making elements to the
left of pivot to be less than pivot and elements to right of
    # pivot to be greater than pivot.
    my_array[my_lb], my_array[array_end] = my_array[array_end], my_array[my_lb]
    return array_end
#-----------------------------------------------------------------------------
-----------------------------------------------------------------#
#Defining main Random Quick Sort function.
```

```
def random_qs(my_array):
    def _random_qs(my_array, my_lb, my_ub):
        stack = [(my_lb, my_ub)]
        #Implementing stack based approach to prevent maximum recursion depth
reached
        #Discussed this approach with student of this course Janaki Ram.
        while stack:
            my_lb, my_ub = stack.pop()
            if my_lb < my_ub:
                pivot_loc = RQS_partition(my_array, my_lb, my_ub)
                stack.append((my_lb, pivot_loc - 1))
                stack.append((pivot_loc + 1, my_ub))

    _random_qs(my_array, 0, len(my_array) - 1)

#_____________________________________________________________________________
_____________________________________________________________________________#
#_____________________________________________MERGE
SORT_____________________________________________________________________________
_______#

def merge_sort(my_array):
    #Checking if length of my_array is greater than 1. If not, then array is
already sorted.
    if len(my_array) > 1:
        mid = len(my_array) // 2  # Find the middle of the array.
        left_half = my_array[:mid]  # Divide the array into two halves.
        right_half = my_array[mid:]

        # Recursive calls to sort the two halves.
        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0 #i, j, and k are used to track current position in left_half,
right_half and my_array respectively.

        # Merge the two halves back together
        while i < len(left_half) and j < len(right_half):
            #Smaller of the two elements are selected and placed in the original
array.
            if left_half[i] < right_half[j]:
                my_array[k] = left_half[i]
                i += 1
            else:
                my_array[k] = right_half[j]
                j += 1
            k += 1

        # Check if any elements were left in the left array.
        while i < len(left_half):
            my_array[k] = left_half[i]
            i += 1
            k += 1

        # Check if any elements were left in the right array.
        while j < len(right_half):
            my_array[k] = right_half[j]
            j += 1
            k += 1
```

```
#_____________________________________________________________________________
_____________________________________________________________________________#
#_____________________________________UNIT
TESTING_____________________________________________________________________
_____________#

# Creating a Sorting_Algorithms_Test class for unit testing of algorithms.
class Sorting_Algorithms_Test(unittest.TestCase):

    def setUp(self):
        # Initializing an empty array for testing
        self.empty_array = []

        # Initializing a single-element array for testing
        self.single_element_array = [17]

        # Initializing a sorted array for testing
        self.sorted_array = list(range(1, 17001))

        # Initializing a reversed sorted array for testing
        self.reverse_sorted_array = list(range(17000, 0, -1))

        # Initializing a random array for testing
        self.random_array = self.sorted_array.copy()
        random.shuffle(self.random_array)

        #Initializing a negative array for testing
        self.negative_array = self.sorted_array.copy()
        for i in self.negative_array:
            self.negative_array[i] = self.negative_array[i] * -1
        random.shuffle(self.negative_array)

        # Initializing a random decimal array for testing
        # Specify the length of the array
        array_length = 100
        # Generate an array of random decimal values between 0 and 1000
        self.random_decimal_array = [random.uniform(0, 1000) for _ in
range(array_length)]
        self.sorted_decimal_array = sorted(self.random_decimal_array)


    #-----------------------------------------------------------Deterministic Quick
Sort Test Cases-------------------------------------------------------------#
    def test_det_qs_empty_array(self):
        # Testing by sorting an empty array
        det_qs(self.empty_array)
        self.assertEqual(self.empty_array, [])

    def test_det_qs_single_element_array(self):
        # Testing by sorting a single-element array
        det_qs(self.single_element_array)
        self.assertEqual(self.single_element_array, [17])

    def test_det_qs_sorted_array(self):
        # Testing by sorting an already sorted array
        det_qs(self.sorted_array)
        self.assertEqual(self.sorted_array, list(range(1, 17001)))
```

```
    def test_det_qs_reverse_sorted_array(self):
        # Testing by sorting a reverse sorted array
        det_qs(self.reverse_sorted_array)
        self.assertEqual(self.reverse_sorted_array, list(range(1, 17001)))

    def test_det_qs_random_array(self):
        # Testing by sorting a random array
        det_qs(self.random_array)
        self.assertEqual(self.random_array, sorted(self.random_array))

    def test_det_qs_negative_array(self):
        # Testing by sorting a negative array
        det_qs(self.negative_array)
        self.assertEqual(self.negative_array, sorted(self.negative_array))

    def test_det_qs_decimal_array(self):
        # Testing by sorting a decimal array
        det_qs(self.random_decimal_array)
        self.assertEqual(self.random_decimal_array, self.sorted_decimal_array)

    #-------------------------------------------------------Randomized Quick
Sort Test Cases-------------------------------------------------------#

    def test_random_qs_empty_array(self):
        # Testing by sorting an empty array
        random_qs(self.empty_array)
        self.assertEqual(self.empty_array, [])

    def test_random_qs_single_element_array(self):
        # Testing by sorting a single-element array
        random_qs(self.single_element_array)
        self.assertEqual(self.single_element_array, [17])

    def test_random_qs_sorted_array(self):
        # Testing by sorting an already sorted array
        random_qs(self.sorted_array)
        self.assertEqual(self.sorted_array, list(range(1, 17001)))

    def test_random_qs_reverse_sorted_array(self):
        # Testing by sorting a reverse sorted array
        random_qs(self.reverse_sorted_array)
        self.assertEqual(self.reverse_sorted_array, list(range(1, 17001)))

    def test_random_qs_random_array(self):
        # Testing by sorting a random array
        random_qs(self.random_array)
        self.assertEqual(self.random_array, sorted(self.random_array))

    def test_random_qs_negative_array(self):
        # Testing by sorting a negative array
        random_qs(self.negative_array)
        self.assertEqual(self.negative_array, sorted(self.negative_array))

    def test_random_qs_decimal_array(self):
        # Testing by sorting a decimal array
        random_qs(self.random_decimal_array)
        self.assertEqual(self.random_decimal_array, self.sorted_decimal_array)

    #-----------------------------------------------------------------Merge Sort
```

```
Test Cases-------------------------------------------------------------------#

    def test_merge_sort_empty_array(self):
        # Testing by sorting an empty array
        merge_sort(self.empty_array)
        self.assertEqual(self.empty_array, [])

    def test_merge_sort_single_element_array(self):
        # Testing by sorting a single-element array
        merge_sort(self.single_element_array)
        self.assertEqual(self.single_element_array, [17])

    def test_merge_sort_sorted_array(self):
        # Testing by sorting an already sorted array
        merge_sort(self.sorted_array)
        self.assertEqual(self.sorted_array, list(range(1, 17001)))

    def test_merge_sort_reverse_sorted_array(self):
        # Testing by sorting a reverse sorted array
        merge_sort(self.reverse_sorted_array)
        self.assertEqual(self.reverse_sorted_array, list(range(1, 17001)))

    def test_merge_sort_random_array(self):
        # Testing by sorting a random array
        merge_sort(self.random_array)
        self.assertEqual(self.random_array, sorted(self.random_array))

    def test_merge_sort_negative_array(self):
        # Testing by sorting a negative array
        merge_sort(self.negative_array)
        self.assertEqual(self.negative_array, sorted(self.negative_array))

    def test_merge_sort_decimal_array(self):
        # Testing by sorting a decimal array
        merge_sort(self.random_decimal_array)
        self.assertEqual(self.random_decimal_array, self.sorted_decimal_array)

if __name__ == '__main__':
    # Creating a test suite
    test_suite =
unittest.defaultTestLoader.loadTestsFromTestCase(Sorting_Algorithms_Test)

    # Running the tests
    unittest.TextTestRunner().run(test_suite)
#_____________________________________________________________________________
_____________________________________________________________#
#_____________________________________________GRAPH
PLOTTING_____________________________________________________________________#

# Defining the input sizes
input_sizes = list(range(0, 15001, 300))

# Initializing lists to store execution times for each function and input type
deterministic_sort_sorted_times = []
deterministic_sort_random_times = []
random_qs_sort_sorted_times = []
random_qs_sort_random_times = []
merge_sort_sorted_times = []
merge_sort_random_times = []
```

```
# Loop for Benchmarking
for size in input_sizes:
    # Generating a sorted array
    sorted_array = list(range(1, size + 1))

    # Generating a random array by shuffling the sorted array
    random_array = sorted_array.copy()
    random.shuffle(random_array)

    # Measuring execution time for deterministic quick sort on sorted array
    start_time = time.time()
    det_qs(sorted_array)
    end_time = time.time()
    deterministic_sort_sorted_times.append(end_time - start_time)

    # Measuring execution time for deterministic quick sort on random array
    start_time = time.time()
    det_qs(random_array)
    end_time = time.time()
    deterministic_sort_random_times.append(end_time - start_time)

    # Measuring execution time for randomized quick sort on sorted array
    start_time = time.time()
    random_qs(sorted_array)
    end_time = time.time()
    random_qs_sort_sorted_times.append(end_time - start_time)

    # Measuring execution time for randomized quick sort on random array
    start_time = time.time()
    random_qs(random_array)
    end_time = time.time()
    random_qs_sort_random_times.append(end_time - start_time)

    # Measuring execution time for merge sort on sorted array
    start_time = time.time()
    merge_sort(sorted_array)
    end_time = time.time()
    merge_sort_sorted_times.append(end_time - start_time)

    # Measuring execution time for merge sort on random array
    start_time = time.time()
    merge_sort(random_array)
    end_time = time.time()
    merge_sort_random_times.append(end_time - start_time)

# Creating a plot
plt.figure(figsize=(10, 6))
plt.plot(input_sizes, deterministic_sort_sorted_times, label='Deterministic Sort
(Sorted)')
plt.plot(input_sizes, deterministic_sort_random_times, label='Deterministic Sort
(Random)')
plt.plot(input_sizes, random_qs_sort_sorted_times, label='Random Sort (Sorted)')
plt.plot(input_sizes, random_qs_sort_random_times, label='Random Sort (Random)')
plt.plot(input_sizes, merge_sort_sorted_times, label='Merge Sort (Sorted)')
plt.plot(input_sizes, merge_sort_random_times, label='Merge Sort (Random)')

# Adding labels and legend
plt.xlabel('Input Size')
```

```
plt.ylabel('Execution Time (seconds)')
plt.legend()

# Show the plot
plt.show()

# Defining a 2nd plot to show clear seperation between the graphs except for the
Deterministic Sort (Sorted)
plt.plot(input_sizes, deterministic_sort_random_times, label='Deterministic Sort
(Random)')
plt.plot(input_sizes, random_qs_sort_sorted_times, label='Random Sort (Sorted)')
plt.plot(input_sizes, random_qs_sort_random_times, label='Random Sort (Random)')
plt.plot(input_sizes, merge_sort_sorted_times, label='Merge Sort (Sorted)')
plt.plot(input_sizes, merge_sort_random_times, label='Merge Sort (Random)')

# Adding labels and legend
plt.xlabel('Input Size')
plt.ylabel('Execution Time (seconds)')
plt.legend()

# Show the plot
plt.show()

#_____________________________________________________________________THE
END_____________________________________________________________________________#
```