Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

Algorithm and Complexities Lab Report 02

on

‘**Quick & Merge Sort - Time Complexities**’

Submitted By:

Reewaj Khanal (61)

Submitted to:

Dr. Rajani Chulyadyo

Assistant Professor

Department of Computer Science and Engineering

School of Engineering

Kathmandu University

Dhulikhel, Kavre

Submission Date: Thursday 30 May 2024

1. Purpose

Implementation, testing and performance measurement of sorting algorithms.

2. Tasks

Implementing the following sorting algorithms [Code in Python]:

2.1. Implementing the following sorting algorithms:

2.2. Also some test cases have been added to my program.

2.2.1. Quick sort

```python
def quick_sort(arr):

   if len(arr) <= 1:

       return arr

   else:

       pivot = arr[len(arr) // 2]

       left = [x for x in arr if x < pivot]

       middle = [x for x in arr if x == pivot]

       right = [x for x in arr if x > pivot]

       return quick_sort(left) + middle + quick_sort(right)



# Test cases

def run_tests():

   test_cases = [

       [],  # Empty list

       [5],  # Single element

       [3, 6, 8, 10, 1, 2, 1],  # List with duplicates
```

```python
        [10, 7, 8, 9, 1, 5],  # Random list

        [5, 4, 3, 2, 1],  # Reversed list

        [1, 2, 3, 4, 5],  # Already sorted list

    ]



    for i, arr in enumerate(test_cases):

        print(f"Test case {i + 1}: {arr}")

        sorted_arr = quick_sort(arr)

        print(f"Sorted: {sorted_arr}\n")



# Run the tests

run_tests()
```

```
(base) reewajkhanal.rk10@RK10 LAB02 % python task01.py
Test case 1: []
Sorted: []

Test case 2: [5]
Sorted: [5]

Test case 3: [3, 6, 8, 10, 1, 2, 1]
Sorted: [1, 1, 2, 3, 6, 8, 10]

Test case 4: [10, 7, 8, 9, 1, 5]
Sorted: [1, 5, 7, 8, 9, 10]

Test case 5: [5, 4, 3, 2, 1]
Sorted: [1, 2, 3, 4, 5]

Test case 6: [1, 2, 3, 4, 5]
Sorted: [1, 2, 3, 4, 5]

(base) reewajkhanal.rk10@RK10 LAB02 %
```

Code: Inputs and Outputs with Test Cases

2.2.2. Merge sort

```python
def merge_sort(arr):

   if len(arr) <= 1:

       return arr



   mid = len(arr) // 2

   left = merge_sort(arr[:mid])

   right = merge_sort(arr[mid:])



   return merge(left, right)



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
```

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

    return result

# Test cases
def run_tests():
    test_cases = [
        [],  # Empty list
        [5],  # Single element
        [3, 6, 8, 10, 1, 2, 1],  # List with duplicates
        [10, 7, 8, 9, 1, 5],  # Random list
        [5, 4, 3, 2, 1],  # Reversed list
        [1, 2, 3, 4, 5],  # Already sorted list
    ]

    for i, arr in enumerate(test_cases):
        print(f"Test case {i + 1}: {arr}")
        sorted_arr = merge_sort(arr)
        print(f"Sorted: {sorted_arr}\n")

# Run the tests
run_tests()
```

```
(base) reewajkhanal.rk10@RK10 LAB02 % python task02.py
Test case 1: []
Sorted: []

Test case 2: [5]
Sorted: [5]

Test case 3: [3, 6, 8, 10, 1, 2, 1]
Sorted: [1, 1, 2, 3, 6, 8, 10]

Test case 4: [10, 7, 8, 9, 1, 5]
Sorted: [1, 5, 7, 8, 9, 10]

Test case 5: [5, 4, 3, 2, 1]
Sorted: [1, 2, 3, 4, 5]

Test case 6: [1, 2, 3, 4, 5]
Sorted: [1, 2, 3, 4, 5]
```

Code: Inputs and Outputs with Test Cases

2.3. Generated some random inputs for my program and applied both quick sort and merge sort algorithms to sort the generated sequence of data. Recorded the execution times of both algorithms for best and worst cases on inputs of different sizes. Plotted an input-size vs execution-time graph.

```
import time

import random

import matplotlib.pyplot as plt



# Quick Sort Implementation

def quick_sort(arr):

   if len(arr) <= 1:

       return arr

   else:
```

```python
        pivot = arr[len(arr) // 2]

        left = [x for x in arr if x < pivot]

        middle = [x for x in arr if x == pivot]

        right = [x for x in arr if x > pivot]

        return quick_sort(left) + middle + quick_sort(right)


# Merge Sort Implementation

def merge_sort(arr):

    if len(arr) <= 1:

        return arr


    mid = len(arr) // 2

    left = merge_sort(arr[:mid])

    right = merge_sort(arr[mid:])


    return merge(left, right)


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])
```

```python
            i += 1

        else:

            result.append(right[j])

            j += 1



    result.extend(left[i:])

    result.extend(right[j:])



    return result



# Function to generate random input array of given size

def generate_random_input(size):

    return [random.randint(1, 1000) for _ in range(size)]



# Function to measure execution time of a sorting algorithm

def measure_execution_time(sort_function, arr):

    start_time = time.time()

    sort_function(arr)

    end_time = time.time()

    return end_time - start_time



# Function to generate best case input for Quick Sort

def generate_quick_sort_best_case(size):

    return list(range(1, size+1))
```

```
# Function to generate worst case input for Quick Sort
def generate_quick_sort_worst_case(size):
   return list(range(size, 0, -1))


# Function to plot best and worst case execution times for Quick Sort and
Merge Sort
def plot_best_worst_case():
   quick_sort_best_case_times = []
   quick_sort_worst_case_times = []
   merge_sort_best_case_times = []
   merge_sort_worst_case_times = []


   for size in sizes:
       # Best case for Quick Sort
       quick_sort_best_case_data = generate_quick_sort_best_case(size)
       quick_sort_best_case_time = measure_execution_time(quick_sort,
quick_sort_best_case_data)
       quick_sort_best_case_times.append(quick_sort_best_case_time)


       # Worst case for Quick Sort
       quick_sort_worst_case_data = generate_quick_sort_worst_case(size)
       quick_sort_worst_case_time = measure_execution_time(quick_sort,
quick_sort_worst_case_data)
       quick_sort_worst_case_times.append(quick_sort_worst_case_time)
```

```
        # Best case for Merge Sort
        merge_sort_best_case_data = quick_sort_best_case_data.copy()  # Merge Sort behaves the same for all cases
        merge_sort_best_case_time = measure_execution_time(merge_sort, merge_sort_best_case_data)
        merge_sort_best_case_times.append(merge_sort_best_case_time)

        # Worst case for Merge Sort
        merge_sort_worst_case_data = quick_sort_worst_case_data.copy()  # Merge Sort behaves the same for all cases
        merge_sort_worst_case_time = measure_execution_time(merge_sort, merge_sort_worst_case_data)
        merge_sort_worst_case_times.append(merge_sort_worst_case_time)

    # Plot best and worst case execution times
    plt.figure(figsize=(12, 6))

    # Quick Sort
    plt.subplot(1, 2, 1)
    plt.plot(sizes, quick_sort_best_case_times, label='Best Case')
    plt.plot(sizes, quick_sort_worst_case_times, label='Worst Case')
    plt.xlabel('Input Size')
    plt.ylabel('Execution Time (s)')
    plt.title('Quick Sort: Best and Worst Case Execution Times')
    plt.legend()
```

```
    # Merge Sort
    plt.subplot(1, 2, 2)
    plt.plot(sizes, merge_sort_best_case_times, label='Best Case')
    plt.plot(sizes, merge_sort_worst_case_times, label='Worst Case')
    plt.xlabel('Input Size')
    plt.ylabel('Execution Time (s)')
    plt.title('Merge Sort: Best and Worst Case Execution Times')
    plt.legend()

    plt.tight_layout()
    plt.show()

# Measure execution times for different input sizes
sizes = [1000, 2000, 3000, 4000, 5000]
quick_sort_times = []
merge_sort_times = []

for size in sizes:
    data = generate_random_input(size)

    quick_sort_time = measure_execution_time(quick_sort, data.copy())
    quick_sort_times.append(quick_sort_time)

```

```
    merge_sort_time = measure_execution_time(merge_sort, data.copy())
    merge_sort_times.append(merge_sort_time)

# Plot input size vs. execution time
plt.plot(sizes, quick_sort_times, label='Quick Sort')
plt.plot(sizes, merge_sort_times, label='Merge Sort')
plt.xlabel('Input Size')
plt.ylabel('Execution Time (s)')
plt.title('Input Size vs. Execution Time for Quick Sort and Merge Sort')
plt.legend()
plt.show()

# Plot best and worst case execution times for Quick Sort and Merge Sort
plot_best_worst_case()
```

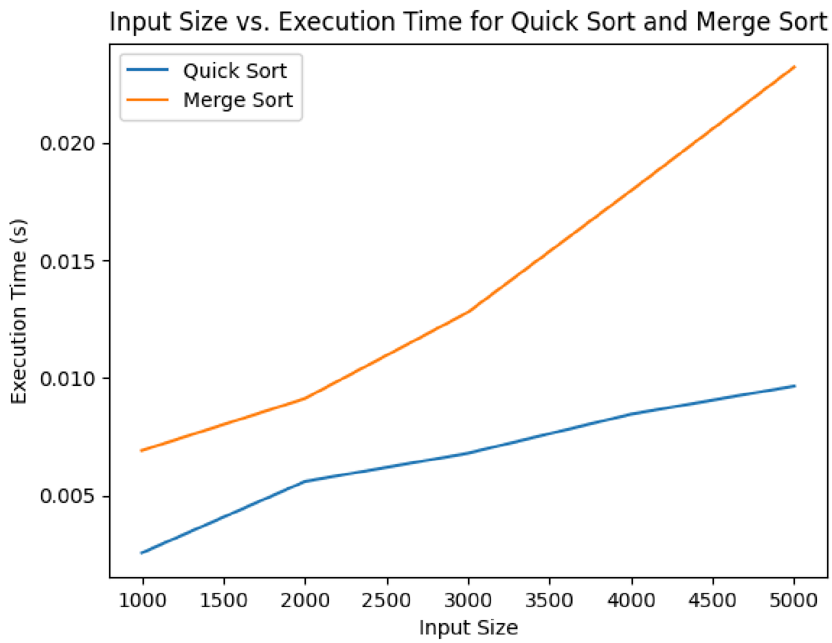

Fig: Input-size vs execution-time graph

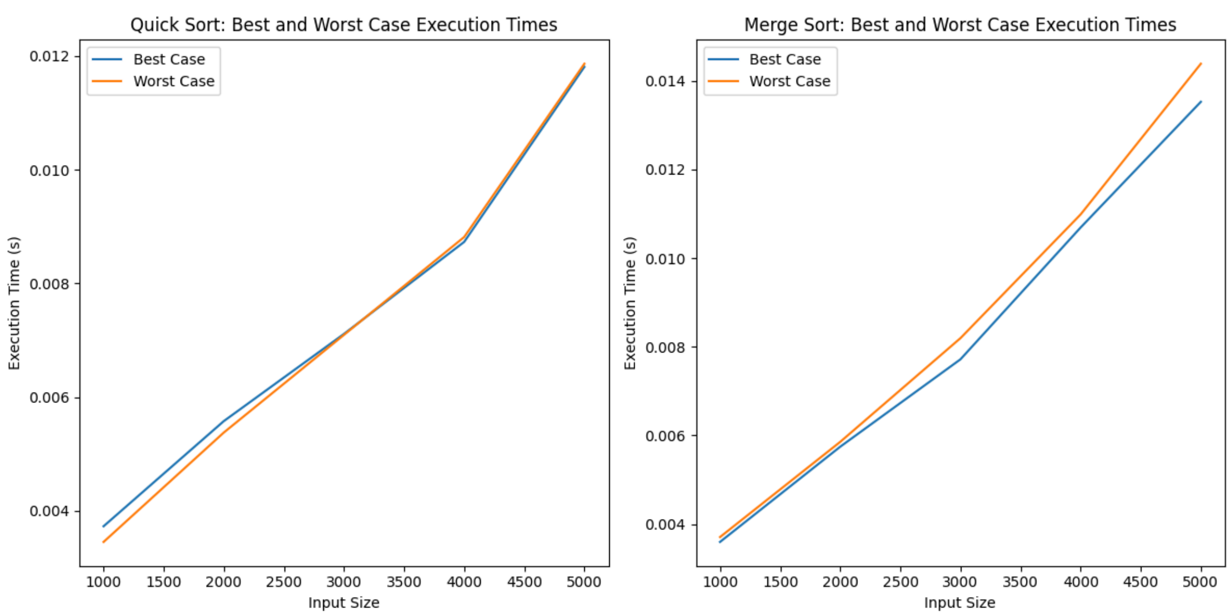

Fig: Best and Worst Case execution-time graph

2.4. Compare your results with those from Lab 1, and explain your observations.

```python
import time

import random

import matplotlib.pyplot as plt



# Insertion Sort

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



# Selection Sort

def selection_sort(arr):

   for i in range(len(arr)):

       min_idx = i

       for j in range(i + 1, len(arr)):

           if arr[j] < arr[min_idx]:

               min_idx = j

       arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

```python
# Quick Sort

def quick_sort(arr, low, high):

   if low < high:

       pi = partition(arr, low, high)

       quick_sort(arr, low, pi - 1)

       quick_sort(arr, pi + 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



# Merge Sort

def merge_sort(arr):

   if len(arr) <= 1:

       return arr

   middle = len(arr) // 2

   left = merge_sort(arr[:middle])

   right = merge_sort(arr[middle:])
```

```
    return merge(left, right)


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


# Function to generate random input

def generate_random_input(size):

    return [random.randint(800, 1200) for _ in range(size)]  # Adjusted
range to have less gaps around 1000


# Function to measure execution time of sorting algorithms

def measure_execution_time(sort_function, arr, *args):

    start_time = time.time()
```

```
    sort_function(arr, *args)

    end_time = time.time()

    return end_time - start_time



# Number of test cases

num_test_cases = 10



# Fixed array size for all test cases

array_size = 100



# Lists to store execution times for each sorting algorithm

insertion_times = []

selection_times = []

quick_times = []

merge_times = []



# Perform test cases

for _ in range(num_test_cases):

    # Generate random input for each test case

    data = generate_random_input(array_size)



    # Measure insertion sort execution time

    insertion_data = data.copy()
```

```
    insertion_time = measure_execution_time(insertion_sort,
insertion_data)

    insertion_times.append(insertion_time)



    # Measure selection sort execution time

    selection_data = data.copy()

    selection_time = measure_execution_time(selection_sort,
selection_data)

    selection_times.append(selection_time)



    # Measure quick sort execution time

    quick_data = data.copy()

    quick_time = measure_execution_time(quick_sort, quick_data, 0,
len(quick_data) - 1)

    quick_times.append(quick_time)



    # Measure merge sort execution time

    merge_data = data.copy()

    merge_time = measure_execution_time(merge_sort, merge_data)

    merge_times.append(merge_time)



# Plot input size vs. execution time for all sorting algorithms

plt.plot(range(1, num_test_cases + 1), insertion_times, label='Insertion
Sort')

plt.plot(range(1, num_test_cases + 1), selection_times, label='Selection
Sort')
```

```
plt.plot(range(1, num_test_cases + 1), quick_times, label='Quick Sort')

plt.plot(range(1, num_test_cases + 1), merge_times, label='Merge Sort')

plt.xlabel('Test Case')

plt.ylabel('Execution Time (s)')

plt.xticks(range(1, num_test_cases + 1))

plt.title('Execution Time for Sorting Algorithms (Array Size: 100)')

plt.legend()

plt.show()
```

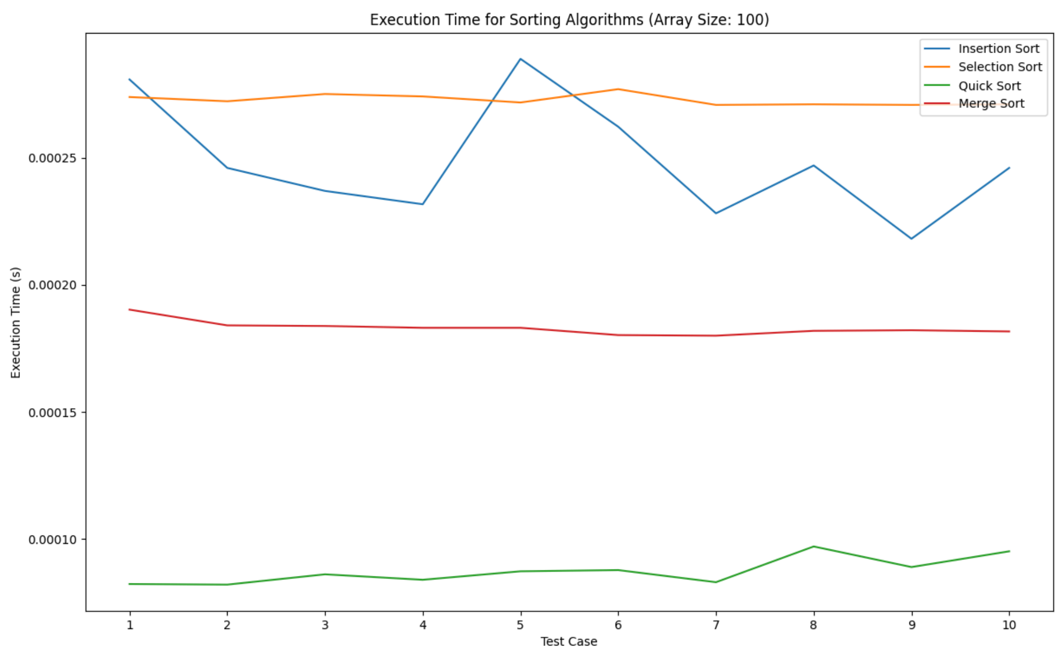

Fig: Result Comparisons of Execution Time for Sorting Algorithms from LAB01 and LAB02 (data set may vary from previous labs)

3. My observations.

The graph illustrates the time complexities of various sorting algorithms concerning array size. Insertion Sort and Selection Sort display O(n^2) worst-case time complexities, while Quick Sort also exhibits O(n^2) but generally outperforms the former due to its average-case efficiency. However, Merge Sort consistently boasts O(n log n) complexity, making it the top performer among the algorithms considered. Thus, while Quick Sort is often favored, Merge Sort emerges as the optimal choice across all scenarios.