Bilal Ahmed Khalil **44926**

Bs-Cy (5-1)

Assignment: 3

Instructor: Muhammad Usman Sharif

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.

Analyse the results to compare the performance of the different sorting algorithms under various conditions.

Solution:

Python Code:

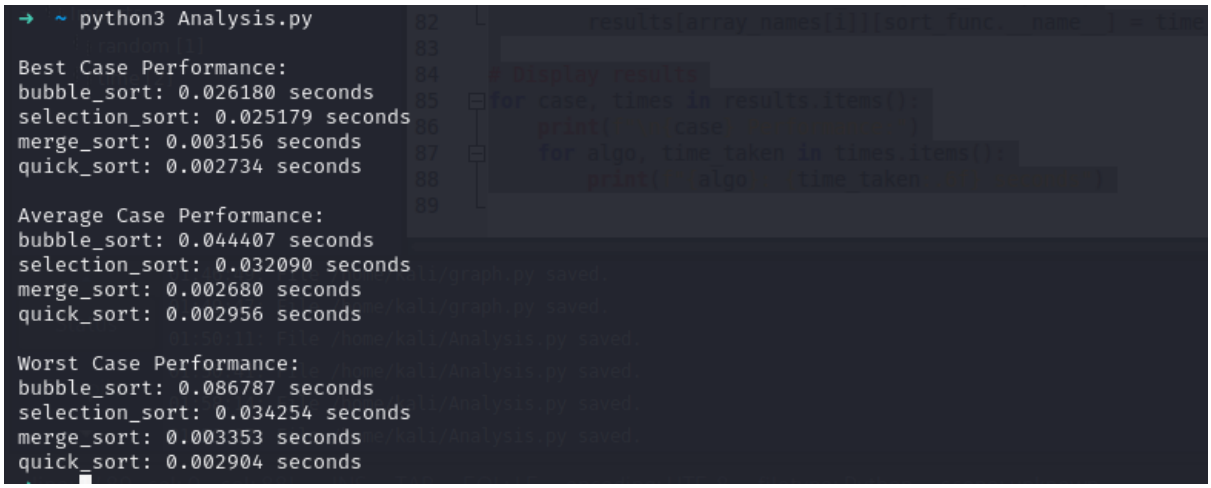
```
import random
import time


# Sorting algorithms
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]


def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
```

```
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

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

        while j < len(R):
            arr[k] = R[j]
```

```
        j += 1
        k += 1

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)

# Create arrays
size = 1000
array1 = list(range(size))  # Best case (already sorted)
array2 = random.sample(range(size), size)  # Average case (random order)
array3 = list(range(size, 0, -1))  # Worst case (reverse sorted)

# Function to measure sorting time
def measure_time(sort_func, arr):
    start_time = time.time()
    sort_func(arr.copy())  # Copy to avoid modifying the original array
    return time.time() - start_time

# Measure and print results
results = {}
algorithms = [bubble_sort, selection_sort, merge_sort, quick_sort]
arrays = [array1, array2, array3]
array_names = ['Best Case', 'Average Case', 'Worst Case']
```

```
for i, arr in enumerate(arrays):
    results[array_names[i]] = {}
    for sort_func in algorithms:
        time_taken = measure_time(sort_func, arr)
        results[array_names[i]][sort_func.__name__] = time_taken

# Display results
for case, times in results.items():
    print(f"\n{case} Performance:")
    for algo, time_taken in times.items():
        print(f"{algo}: {time_taken:.6f} seconds")
```

Output of above code:



Under different conditions i checked performance of each individual sort here is code with graph of each sort:

Bubble Sort:

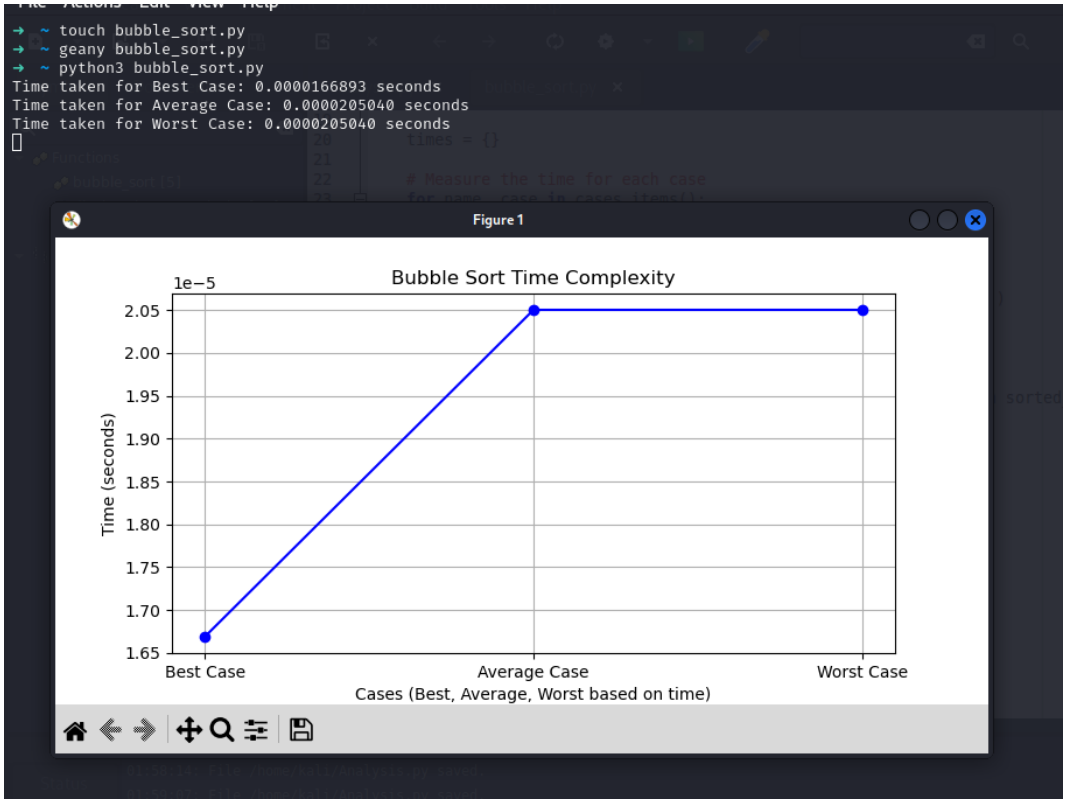
Code:

```
import time
import matplotlib.pyplot as plt

# Bubble Sort Implementation
```

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

# Time complexity display
def display_time_complexity():
    cases = {
        "Best Case": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],        # Already sorted
        "Average Case": [3, 5, 1, 4, 2, 6, 9, 8, 10, 7],    # Random order
        "Worst Case": [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]       # Reverse sorted
    }

    times = {}

    # Measure the time for each case
    for name, case in cases.items():
        start_time = time.time()
        bubble_sort(case.copy())
        end_time = time.time()
        elapsed_time = end_time - start_time
        print(f"Time taken for {name}: {elapsed_time:.10f} seconds")
        times[name] = elapsed_time

    # Sort cases based on time taken
    sorted_cases = sorted(times.items(), key=lambda x: x[1])
    return [case[0] for case in sorted_cases], [case[1] for case in sorted_cases]
```

```
# Plotting the time complexity
def plot_bubble_sort():
    cases, times = display_time_complexity()

    plt.figure(figsize=(8, 4))
    plt.plot(cases, times, marker='o', color='blue', linestyle='-')
    plt.title('Bubble Sort Time Complexity')
    plt.xlabel('Cases (Best, Average, Worst based on time)')
    plt.ylabel('Time (seconds)')
    plt.grid(True)
    plt.show()
plot_bubble_sort()
```

Graph:



Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is s orted.

1. Initial Array: [5, 1, 8, 3, 9, 6, 2, 10, 4, 7]
2. Pass 1:
    - [5, **1**, 8, 3, 9, 6, 2, 10, 4, 7] → [**1**, 5, 8, 3, 9, 6, 2, 10, 4, 7]
    - [1, 5, **8**, 3, 9, 6, 2, 10, 4, 7] → [1, **5**, 3, 8, 9, 6, 2, 10, 4, 7]
    - [1, 5, 3, **8**, 9, 6, 2, 10, 4, 7] → [1, 5, **3**, 8, 9, 6, 2, 10, 4, 7]
    - And so on...

Time complexity:

As bubble sort uses two nested loops so its time complexity would be It moves largest element of selected array at the end of each iteration

Best case:

Occurs when the array is already sorted. In this case, the algorithm will only make one pass through the array, checking for swaps and finding none

In our case best case would be the below sorted array

Best Case": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

So here time complexity would be O(n)

As no swapping occur

Average Case:

In average case i consider half of elements are swiped

For a randomly ordered array like [3, 5, 1, 4, 2, 6, 9, 8, 10, 7], the algorithm will make multiple passes, resulting in comparisons and swaps. So as bubble sort is using nested loops and inner makes the swaps so it is being used so its

Time complexity would be **O(n^2)**

As half of the elements are swapped in random order

Worst Case:

In worst case i came of with reversely sorted array here each element of array will be compared and swapped. For the array [10, 9, 8, 7, 6, 5, 4, 3, 2, 1], each element will be compared and swapped in every pass.

As here we are using nested loop and each iteration of inner loop is utilized so

Its time complexity would be **O(n^2)**

Selection Sort:

Code:

```
import time
import matplotlib.pyplot as plt


# Selection Sort Implementation
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]


# Time complexity display
def display_time_complexity():
    cases = {
        "Best Case": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],        # Already sorted
        "Average Case": [3, 5, 1, 4, 2, 6, 9, 8, 10, 7],    # Random order
        "Worst Case": [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]       # Reverse sorted
    }


    times = {}


    # Measure the time for each case
    for name, case in cases.items():
```

```
        start_time = time.time()
        selection_sort(case.copy())
        end_time = time.time()
        elapsed_time = end_time - start_time
        print(f"Time taken for {name}: {elapsed_time:.10f} seconds")
        times[name] = elapsed_time

    # Sort cases based on time taken
    sorted_cases = sorted(times.items(), key=lambda x: x[1])
    return [case[0] for case in sorted_cases], [case[1] for case in sorted_cases]

# Plotting the time complexity
def plot_selection_sort():
    cases, times = display_time_complexity()

    plt.figure(figsize=(8, 4))
    plt.plot(cases, times, marker='o', color='orange', linestyle='-')
    plt.title('Selection Sort Time Complexity')
    plt.xlabel('Cases (Best, Average, Worst based on time)')
    plt.ylabel('Time (seconds)')
    plt.grid(True)
    plt.show()
plot_selection_sort()
```

Graph:

Selection Sort repeatedly finds the minimum element from the unsorted part and puts it at the beginning.

1. Initial Array: [5, 1, 8, 3, 9, 6, 2, 10, 4, 7]
2. Pass 1: Find the minimum value (1), swap with the first element:
    - [**5**, 1, 8, 3, 9, 6, 2, 10, 4, 7] → [**1**, 5, 8, 3, 9, 6, 2, 10, 4, 7]
3. Pass 2: Find the next minimum value (2), swap with the second element:
    - [1, **5**, 8, 3, 9, 6, 2, 10, 4, 7] → [1, **2**, 8, 3, 9, 6, 5, 10, 4, 7]

Time Complexity:
Selection Sort also uses two nested loops, it move smallest element with each iteration at the start resulting in the following time complexities:

Best Case:
Regardless of the initial order, Selection Sort will perform n(n−1)/2 comparisons to find the minimum element for each position.

For an already sorted array like [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], As it also uses nested loops

the time complexity is **O(n^2)**

as it still compares every element even if no swaps occur.

Average Case:
The average case remains O(n^2) because Selection Sort consistently performs about n(n−1)/2 comparisons and swaps regardless of the input order.

For a random array like [3, 5, 1, 4, 2, 6, 9, 8, 10, 7] it scans through the elements.

So its time complexity also would be O(n^2)

Worst Case:
The worst case is also O(n^2) occurring when the array is in reverse order. The algorithm still performs the same number of comparisons.

For the array [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] it continues to compare every element using nested loops

So its time complexity is also be **O(n2)**

They all may seems same but their runtime can be different according the hardware they run on as in my case they take different time to execute in each case although their time complexity is same i.e. **O(n^2)**

Merge Sort:

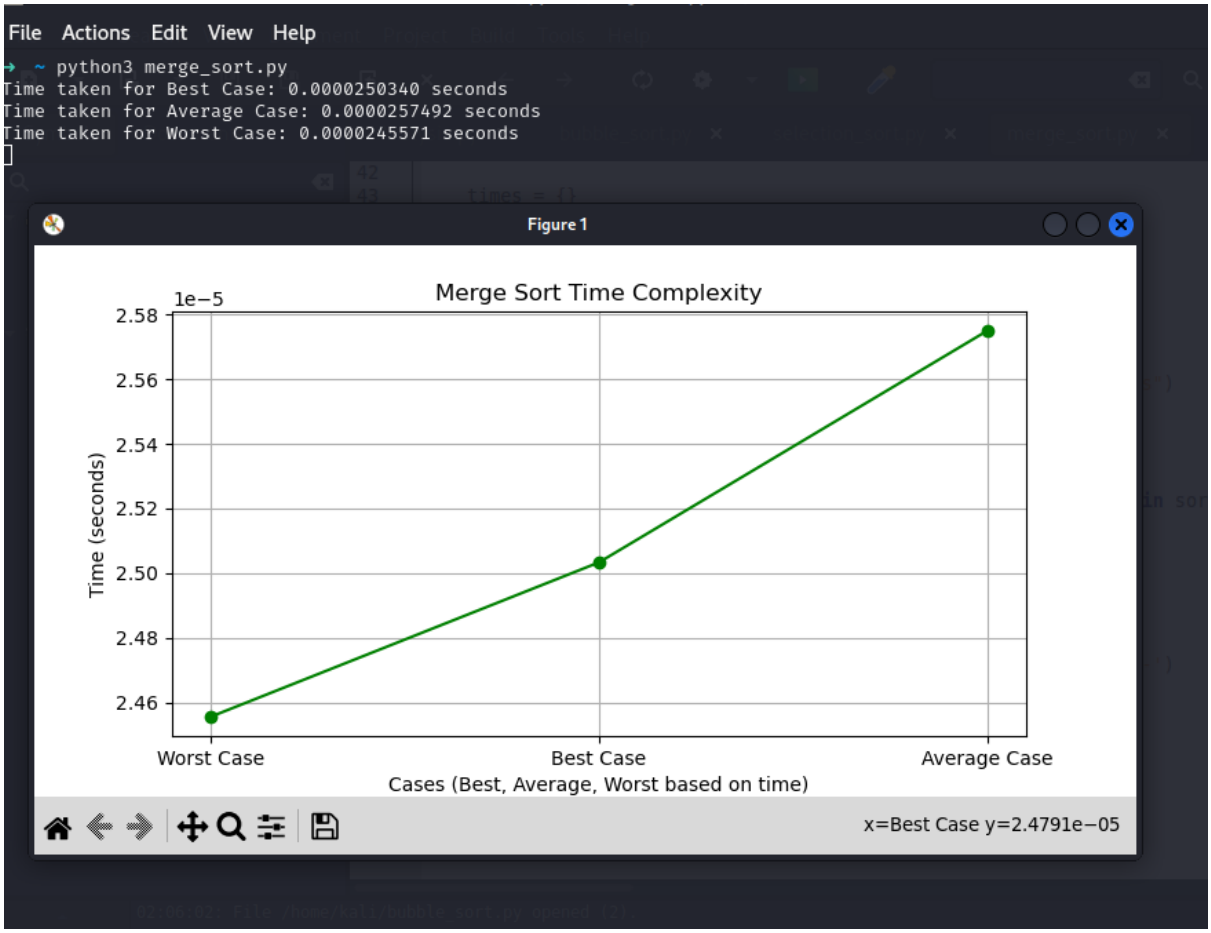
Code:

```
import time

import matplotlib.pyplot as plt


# Merge Sort Implementation

def merge_sort(arr):

  if len(arr) > 1:

    mid = len(arr) // 2

    left_half = arr[:mid]

    right_half = arr[mid:]


    merge_sort(left_half)

    merge_sort(right_half)


    i = j = k = 0


    while i < len(left_half) and j < len(right_half):
```

```
            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 < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

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

# Time complexity display
def display_time_complexity():
    cases = {
        "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]
    }

    times = {}

    # Measure the time for each case
    for name, case in cases.items():
        start_time = time.time()
```

```
        merge_sort(case.copy())
        end_time = time.time()
        elapsed_time = end_time - start_time
        print(f"Time taken for {name}: {elapsed_time:.10f} seconds")
        times[name] = elapsed_time

    # Sort cases based on time taken
    sorted_cases = sorted(times.items(), key=lambda x: x[1])
    return [case[0] for case in sorted_cases], [case[1] for case in sorted_cases]

# Plotting the time complexity
def plot_merge_sort():
    cases, times = display_time_complexity()

    plt.figure(figsize=(8, 4))
    plt.plot(cases, times, marker='o', color='green', linestyle='-')
    plt.title('Merge Sort Time Complexity')
    plt.xlabel('Cases (Best, Average, Worst based on time)')
    plt.ylabel('Time (seconds)')
    plt.grid(True)
    plt.show()

plot_merge_sort()
```

Graph:



Merge Sort divides the array into halves, sorts each half, and then merges them back tog ether.

1. Initial Array: [5, 1, 8, 3, 9, 6, 2, 10, 4, 7]
2. Divide array into halves:
    - Left: [5, 1, 8, 3, 9]
    - Right: [6, 2, 10, 4, 7]
3. Recursively sort each half and merge:
    - Left: [1, 3, 5, 8, 9]
    - Right: [2, 4, 6, 7, 10]

Time Complexity:
Merge Sort operates with a divide-and-conquer approach, leading to the following time complexities it break the array in half and then it merge them in order:

Best Case:
The best case occurs when the array is already sorted, but the algorithm still divides and merges the array. For an already sorted array like [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],

it requires **O(nlogn)** time due to the merging process.

Average Case:
The average case remains **O(nlogn)** because the algorithm consistently divides the array into halves and merges them regardless of the order. For a random array like [3, 5, 1, 4, 2, 6, 9, 8, 10, 7], the time complexity is still **O(nlogn**).

Worst Case:
The worst case is also **O(nlogn)** as the process of dividing and merging remains efficient even in the most unfavorable conditions. For a reverse sorted array like [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

it still performs **O(nlogn**) due to the consistent partitioning and merging.

Quick Sort:

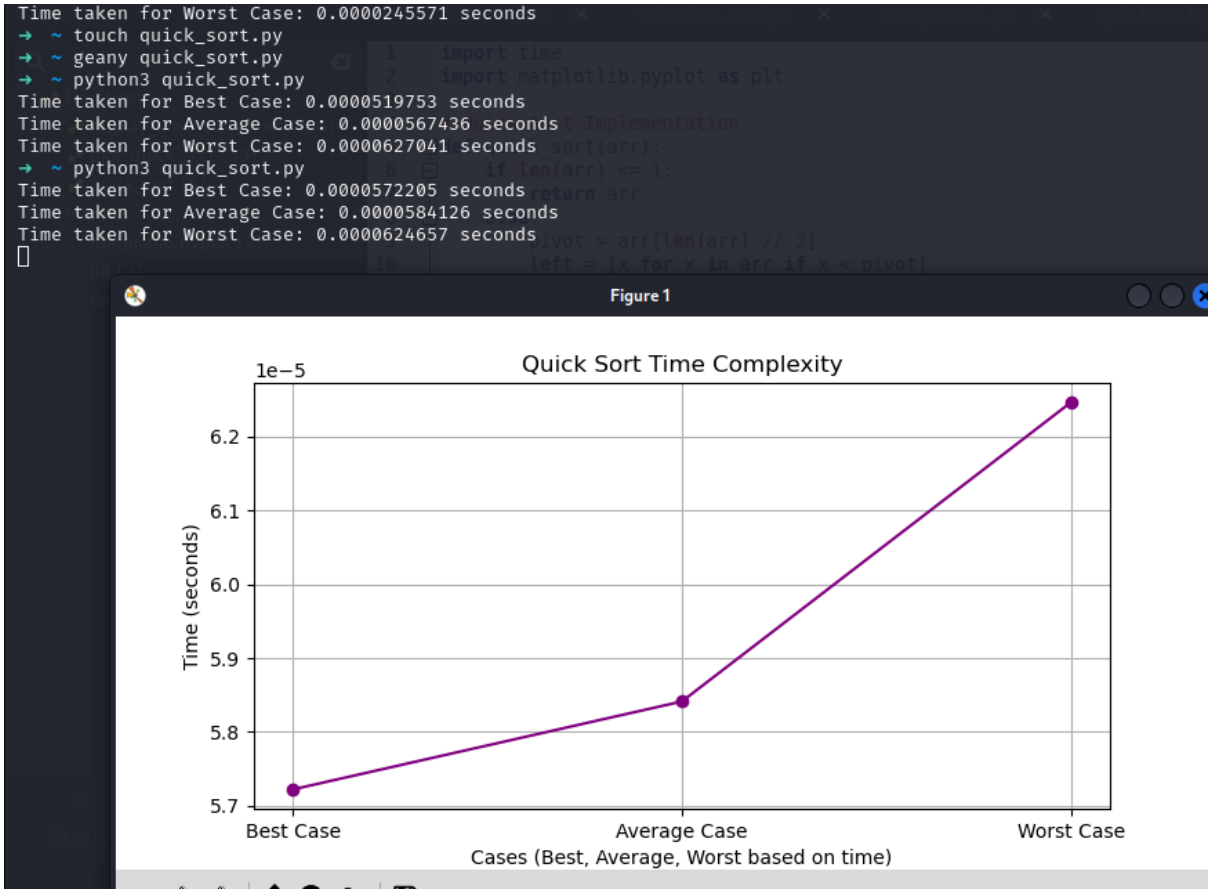
Code:

```
import time

import matplotlib.pyplot as plt


# Quick Sort Implementation

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)


# Time complexity display

def display_time_complexity():

  cases = {
```

```
        "Best Case": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],        # Already sorted
        "Average Case": [3, 5, 1, 4, 2, 6, 9, 8, 10, 7],    # Random order
        "Worst Case": [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]       # Reverse sorted
    }

    times = {}

    # Measure the time for each case
    for name, case in cases.items():
        start_time = time.time()
        quick_sort(case.copy())
        end_time = time.time()
        elapsed_time = end_time - start_time
        print(f"Time taken for {name}: {elapsed_time:.10f} seconds")
        times[name] = elapsed_time

    # Sort cases based on time taken
    sorted_cases = sorted(times.items(), key=lambda x: x[1])
    return [case[0] for case in sorted_cases], [case[1] for case in sorted_cases]

# Plotting the time complexity
def plot_quick_sort():
    cases, times = display_time_complexity()

    plt.figure(figsize=(8, 4))
    plt.plot(cases, times, marker='o', color='purple', linestyle='-')
    plt.title('Quick Sort Time Complexity')
    plt.xlabel('Cases (Best, Average, Worst based on time)')
    plt.ylabel('Time (seconds)')
```

```
plt.grid(True)
plt.show()

plot_quick_sort()
```

Graph:



Quick Sort picks an element as pivot and partitions the array around the pivot.

1. Initial Array: [5, 1, 8, 3, 9, 6, 2, 10, 4, 7]
2. Choose Pivot (e.g., 6):
    - Partition array into elements less than pivot and greater than pivot:
        - Less: [5, 1, 3, 2, 4]
        - Equal: [6]
        - Greater: [8, 9, 10, 7]
3. Recursively sort sub-arrays and combine:
    - Less: [1, 2, 3, 4, 5]
    - Greater: [7, 8, 9, 10]

- Combined: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Time Complexity:
Quick Sort has a more variable performance based on pivot selection, resulting in the following time complexities:

Best Case:
The best case occurs when the pivot consistently divides the array into two equal halves. For an ideally sorted array like [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

it achieves **O(nlogn)** time due to efficient partitioning.

Average Case:
The average case is also O(nlogn) as Quick Sort typically results in reasonably balanced partitions across various inputs. For a random array like [3, 5, 1, 4, 2, 6, 9, 8, 10, 7], it effectively processes elements, leading to **O(nlogn).**

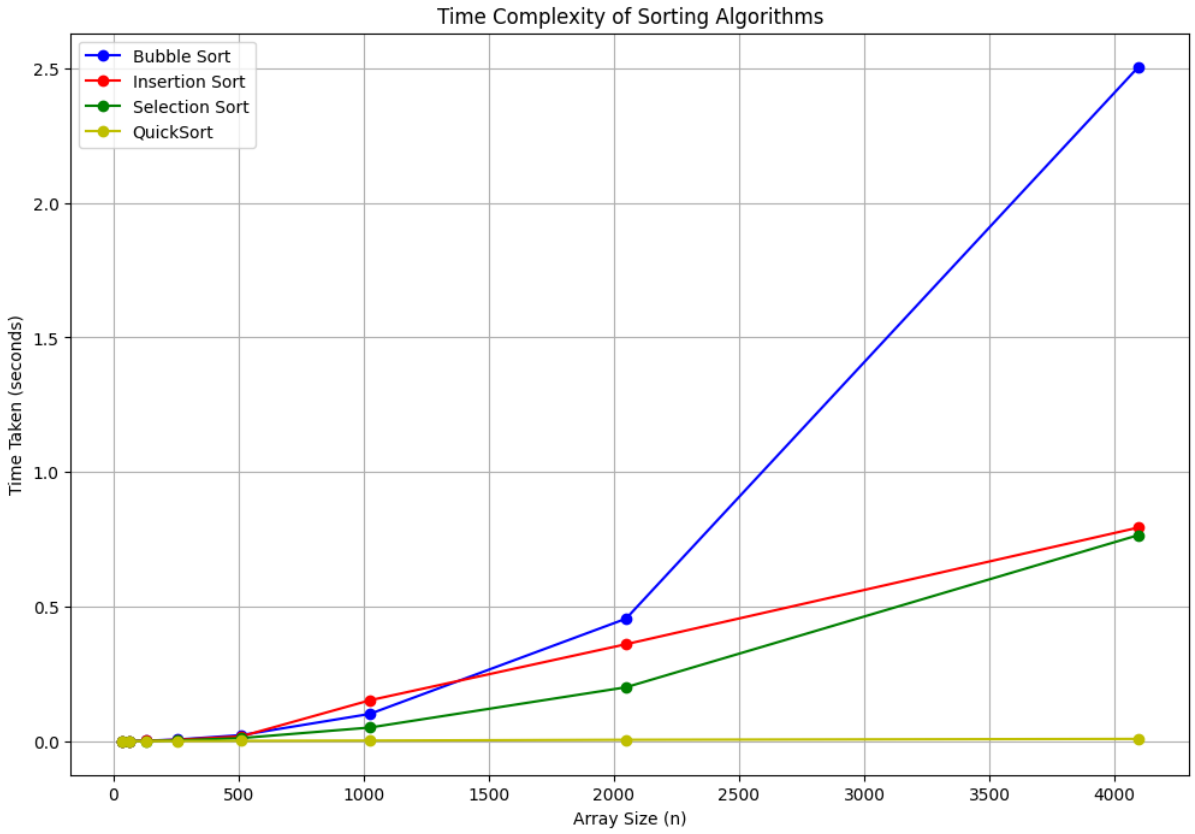
Worst Case:
The worst case occurs when the pivot selection leads to highly unbalanced partitions, particularly when the smallest or largest element is consistently chosen. For a reverse sorted array like [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Quick Sort will behave inefficiently, resulting in **O(n^2)** due to unbalanced splits.

Comparison:

Sorting Algorithm	Best Case	Average Case	Worst Case
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$

The overall time complexity chart according to time take by each sort is:



As i am using this graph formation on Linux virtual machine the graph formation can be slightly different but it provides deep valuable analysis with run time graphs.