COMP 314: Algorithms and Complexity
Lab work 2: Sorting

Purpose

Implementation, testing and performance measurement of sorting algorithms.

Methodology:

Some of the methodologies and the code snippet is given below. For complete code please refer to https://github.com/ranjanlamsal/COMP314Lab.git .

Input Generation:

Random arrays of integers are generated using the generate_array function, with the lengths of the arrays varying from 10 to 100,000.

```
def generate_array(n):
    A = []
    for i in range(n):
        A.append(randint(0, 100000000000))

    return A
```

Sorting Algorithms:

Two sorting algorithms, insertion sort and selection sort, are applied to the generated arrays to measure their execution times.

Merge Sort

```
#merge.py

import math

def merge(A, p, q, r):
    n1 = q - p + 1
    n2 = r - q
```

```
    L = [0] * (n1 + 1)
    R = [0] * (n2 + 1)

    for i in range(n1):
        L[i] = A[p + i]

    for j in range(n2):
        R[j] = A[q + 1 + j]

    L[n1] = math.inf
    R[n2] = math.inf

    i = 0
    j = 0

    for k in range(p, r + 1):
        if L[i] <= R[j]:
            A[k] = L[i]
            i += 1
        else:
            A[k] = R[j]
            j += 1

def merge_sort(A, p, r):
    if p < r:
        q = (p + r) // 2
        merge_sort(A, p, q)
        merge_sort(A, q + 1, r)
        merge(A, p, q, r)

def main():
    arr = [8, 0, 2, 1, 4, 5, 3, 98, 8]
    merge_sort(arr, 0, len(arr) - 1)
    print(arr)

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

Quick Sort

```
#quick_sort.py

def partition(arr, low, high):
   pivot = arr[high]  # Choose the last element as pivot
   i = low - 1

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

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

def quick_sort(arr, low, high):
   if low < high:
       pi = partition(arr, low, high)  # Partition index

       # Recursively sort elements before and after partition
       quick_sort(arr, low, pi - 1)
       quick_sort(arr, pi + 1, high)

def main():
   arr = [8, 0, 2, 1, 4, 5, 3, 98, 8]
   quick_sort(arr, 0, len(arr) - 1)
   print("Sorted array:", arr)

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

Execution Time Measurement:

The execution times of both sorting algorithms are recorded using the time module in Python.

```python
#time_complexity.py

from merge import merge_sort
from quick_sort import quick_sort
from generate_array import generate_array
from time import time


def calculate_merge_time(n):
   testArray1 = generate_array(n)

   start = time()

   resultArray1_insertion = merge_sort(testArray1, 0, len(testArray1)-1)

   end = time()

   return end - start


def calculate_quick_time(n):
   testArray1 = generate_array(n)

   start = time()

   resultArray1_selection = quick_sort(testArray1, 0, len(testArray1)-1)

   end = time()

   return end - start
```

Visualization

The execution times of insertion sort and selection sort for different input sizes are compared and graph is visualized using matplotlib

```
#graph.py
import matplotlib.pyplot as plt
from time_complexity import calculate_merge_time, calculate_quick_time

sizes = [10, 100, 1000, 10000, 100000]
merge_time= []
quick_time = []

for size in sizes:
   merge_time.append(calculate_merge_time(size))
   quick_time.append(calculate_quick_time(size))


# Graph
x = sizes
y1 = merge_time
y2 = quick_time
print(f"Merge sort time:{merge_time}")
print(f"Quick sort time:{quick_time}")

# Plotting
plt.plot(x, y1, marker='o', color='b', label='Merge sort')
plt.plot(x, y2, marker='s', color='r', label='Quick Sort')

# Adding labels and title
plt.xlabel('N')
plt.ylabel('Time')
plt.title('Merge and Quick Sort')

# Adding legend
plt.legend()

# Displaying the plot
plt.grid(True)
plt.show()
```

Observations:

Both insertion sort and selection sort exhibit a time complexity of $O(n2)$, leading to significant increases in execution time as the input size grows. Insertion sort, while efficient for small arrays, sees rapid performance degradation with larger inputs. Selection sort shows slightly better average-case performance for larger sizes, though both are impractical for very large datasets. In contrast, merge sort and quicksort, with their $O(n\log n)$ time complexity, handle larger arrays more efficiently. Merge sort provides consistent and predictable performance due to its stable time complexity, making it suitable for large datasets and linked lists. Quicksort generally performs faster on average, especially with good pivot selection, but can degrade to $O(n2)$ in rare cases of poor pivot choices.

For input_sizes = [10, 100, 1000, 10000, 100000]

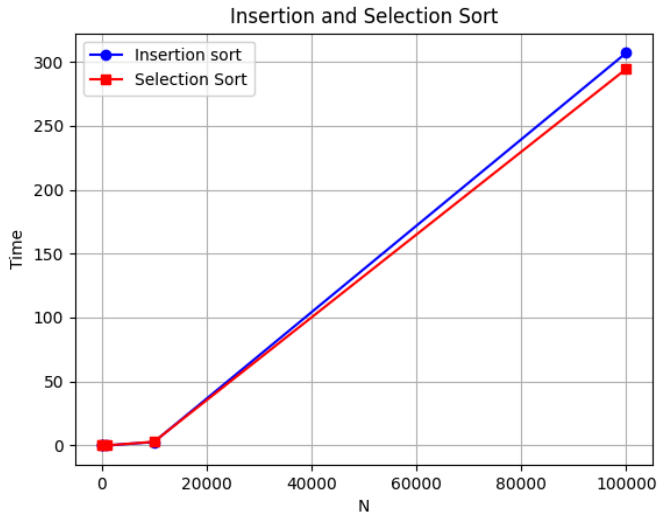
Time lapses for Insertion sort = [1.430511474609375e-05, 0.0004062652587890625, 0.02154684066772461, 2.634857654571533, 306.99769043922424]

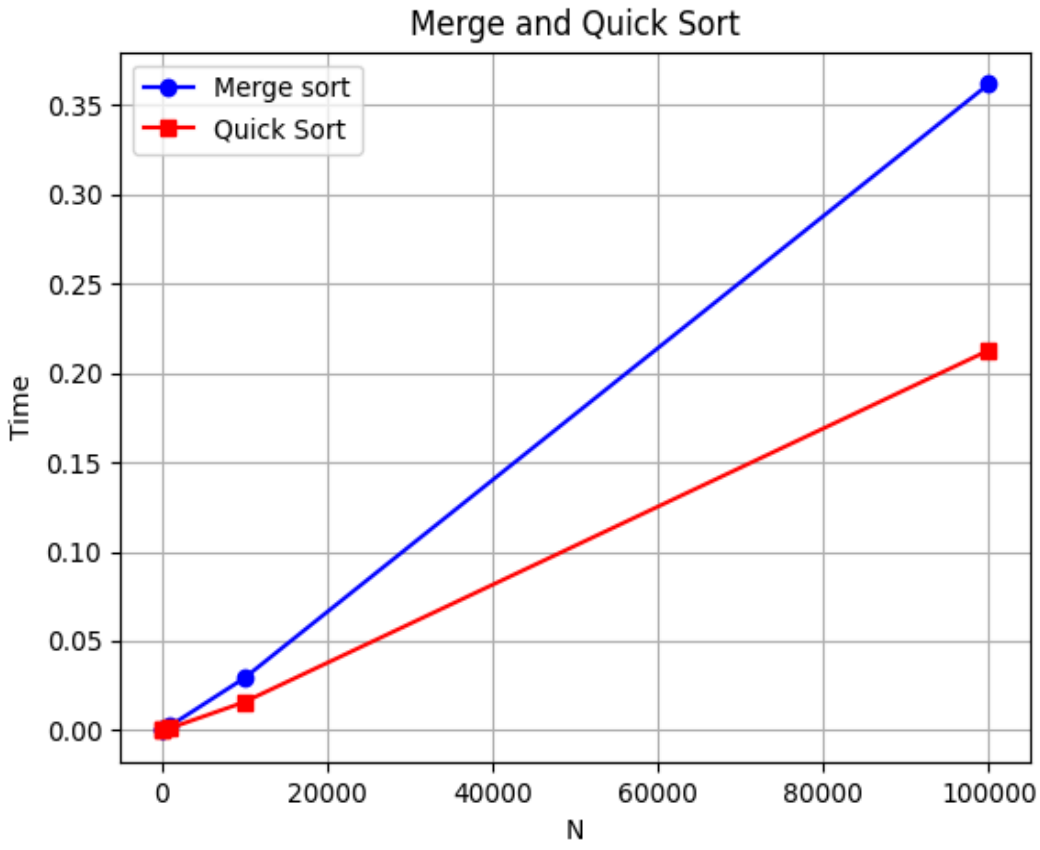
Time lapses for selection sort = [1.2636184692382812e-05, 0.0003943443298339844, 0.026239871978759766, 2.8456101417541504, 294.4455142021179]

Time lapses for merge sort = [3.218650817871094e-05, 0.0017428398132324222, 0.002317667007446289, 0.042077064514160156, 0.49659061431884766]

Time lapses for quick sort =[1.1444091796875e-05, 7.033348083496094e-05, 0.001188039779663086, 0.025665283203125, 0.32913899421691895]

For small arrays or nearly sorted data, insertion sort is advantageous, while selection sort is useful for its simplicity and low memory usage. For larger datasets, merge sort and quicksort are preferable, with quicksort often offering better average performance and merge sort ensuring predictable results.





Submitted By:

Ranjan Lamsal
Roll call: 26
Group: CE (3rd year 2nd Semester)