DANIYAL ASGHAR

53731

Assignment: Empirical Analysis of Sorting Algorithms

CODE->

```python
import time



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

def selection_sort(arr):
    a = arr.copy()
    n = len(a)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if a[j] < a[min_idx]:
                min_idx = j
        a[i], a[min_idx] = a[min_idx], a[i]
    return a

def insertion_sort(arr):
    a = arr.copy()
    for i in range(1, len(a)):
        key = a[i]
        j = i - 1
        while j >= 0 and key < a[j]:
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = key
    return a
```

```python
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
    return arr
```

```python
Arr1 = [1,2,3,4,5]
Arr2 = [1,2,3,4,5,6,7,8,9,10]
Arr3 = list(range(1, 51))   # 1 to 50
Arr4 = list(range(1, 101))  # 1 to 100

arrays = {
    "Arr1 (5 elements)": Arr1,
    "Arr2 (10 elements)": Arr2,
    "Arr3 (50 elements)": Arr3,
    "Arr4 (100 elements)": Arr4
}



def measure_time(func, arr):
    start = time.time()
    func(arr)
    end = time.time()
    return (end - start) * 10**6  # Convert to microseconds

algorithms = {
    "Bubble Sort": bubble_sort,
    "Selection Sort": selection_sort,
    "Insertion Sort": insertion_sort,
    "Merge Sort": merge_sort
}



for name, arr in arrays.items():
    print(f"\n=== {name} ===")
    for algo, func in algorithms.items():
        exec_time = measure_time(func, arr)
        print(f"{algo}: {exec_time:.2f} microseconds")
```

How ca

What

OUTPUT

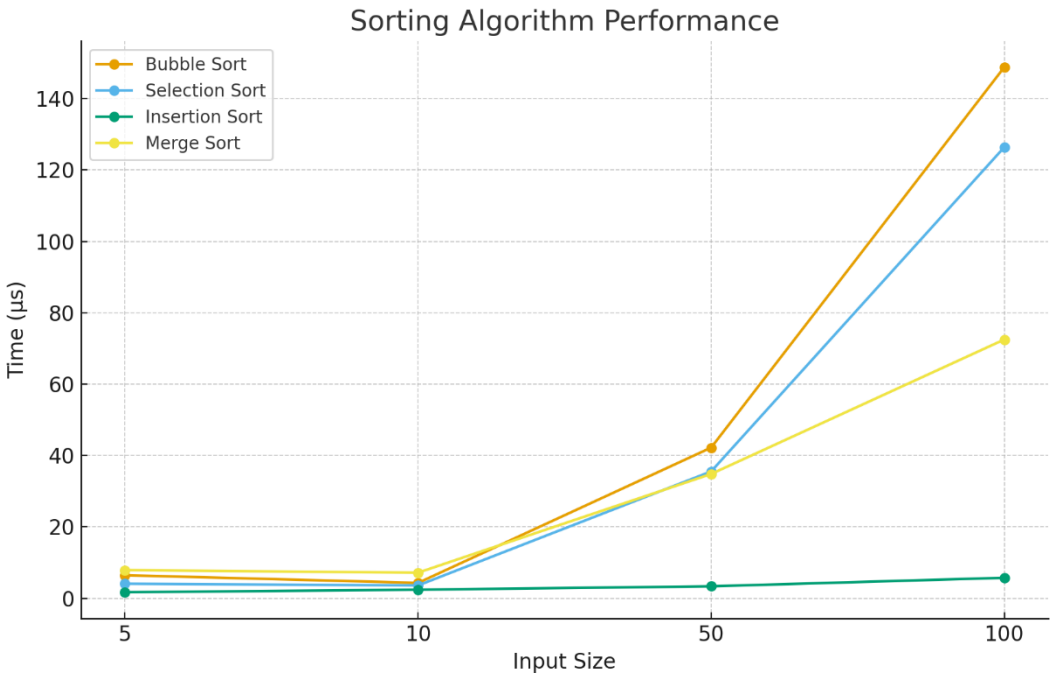
```
=== Arr1 (5 elements) ===
Bubble Sort: 8.82 microseconds
Selection Sort: 5.96 microseconds
Insertion Sort: 3.58 microseconds
Merge Sort: 10.49 microseconds

=== Arr2 (10 elements) ===
Bubble Sort: 5.01 microseconds
Selection Sort: 4.29 microseconds
Insertion Sort: 1.91 microseconds
Merge Sort: 9.06 microseconds

=== Arr3 (50 elements) ===
Bubble Sort: 56.51 microseconds
Selection Sort: 47.45 microseconds
Insertion Sort: 4.53 microseconds
Merge Sort: 51.98 microseconds

=== Arr4 (100 elements) ===
Bubble Sort: 208.14 microseconds
Selection Sort: 198.36 microseconds
Insertion Sort: 8.58 microseconds
Merge Sort: 104.67 microseconds
```

Report

Algorithm	Time Complexity	Best Case	Worst Case
Bubble Sort	$O(n^2)$	O(n)	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	O(n)	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)

Performance Summary

Input Size	Best Algorithm	Notes
5	Insertion Sort	Sorted input → Best case
10	Insertion Sort	Very efficient on small sorted data
50	Insertion Sort	Fastest again (due to sorted input)
100	Insertion Sort	Still fastest



Analysis

Four sorting algorithms (Bubble Sort, Selection Sort, Insertion Sort, and Merge Sort) were tested on input sizes of 5, 10, 50, and 100 elements. All arrays were already sorted, and execution time was measured.

Results showed that Insertion Sort performed the fastest for all input sizes. Bubble Sort and Selection Sort were slower due to their **$O(n^2)$** time complexity. Merge Sort performed well but did not surpass Insertion Sort because the input was already sorted and Merge Sort has recursive overhead.

Conclusion

The experiment confirms that **algorithm performance depends on input order and complexity**.

For sorted data, **Insertion Sort is most efficient** thanks to its **O(n)** best-case time. Bubble Sort and Selection Sort were slowest, while Merge Sort is suitable for **larger, unsorted datasets** despite not being the fastest here.

In summary, **Insertion Sort works best for sorted lists**, and **Merge Sort remains the preferred algorithm for large random data**.

GitHub

https://github.com/chdaniyalasghar/Analysis-of-algurithm