**ALGORITHM ANALYSIS**

Laboratory Nr 2

**Subject**: Study and empirical analysis of sorting algorithms. Analysis of quickSort, mergeSort, heapSort, (one of your choice-BubbleSort)

1. Test sorting algorithms on different input cases:

* Random data
* Sorted data (best case)
* Reverse sorted data (worst case)
* Few unique values (partially sorted)

1. Choose Metrics for Comparison

* Time complexity (execution time)

1. Empirical Analysis
2. **QuickSort:**

def quicksort(arr):

    if len(arr) <= 1:

        return arr

    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 quicksort(left) + middle + quicksort(right)

* + Performs efficiently on all input sizes, maintaining relatively low execution times.
  + Handles small and large arrays well, making it a solid choice for general sorting tasks.
  + Slight performance variations due to pivot selection but remains among the fastest.
  + **Best Case:** O(nlog n) (When pivot divides the array evenly)
  + **Average Case:** O(nlog n) (Expected case)
  + **Worst Case:** O(n2)(When pivot is the smallest or largest element, leading to unbalanced partitions)
  + **Space Complexity:** O(log n)(Recursive calls)

1. **MergeSort:**

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

    result.extend(left[i:])

    result.extend(right[j:])

    return result

* + Performs consistently well, with execution times close to QuickSort.
  + Stable sorting algorithm, meaning it preserves the order of equal elements.
  + Performs well for larger datasets but has a slight overhead due to recursion.
  + **Best Case:** O(nlog n)
  + **Average Case:** O(nlog n)
  + **Worst Case:** O(nlog n)
  + **Space Complexity:** O(n) (Extra space needed for merging)

1. **HeapSort:**

def heap\_sort(arr):

    def heapify(arr, n, i):

        largest = i

        left = 2 \* i + 1

        right = 2 \* i + 2

        if left < n and arr[left] > arr[largest]:

            largest = left

        if right < n and arr[right] > arr[largest]:

            largest = right

        if largest != i:

            arr[i], arr[largest] = arr[largest], arr[i]

            heapify(arr, n, largest)

    n = len(arr)

    for i in range(n // 2 - 1, -1, -1):

        heapify(arr, n, i)

    for i in range(n - 1, 0, -1):

        arr[i], arr[0] = arr[0], arr[i]

        heapify(arr, i, 0)

    return arr

* + Shows slightly higher execution times compared to QuickSort and MergeSort.
  + Suitable for applications where constant time complexity is preferred, but it does not outperform QuickSort in most cases.
  + **Best Case:** O(nlog n)
  + **Average Case:** O(nlog n)
  + **Worst Case:** O(nlog n)
  + **Space Complexity:** O(1) (In-place sorting)

1. **BubbleSort:**

def bubble\_sort(arr):

    n = len(arr)

    for i in range(n):

        swapped = False

        for j in range(0, n-i-1):

            if arr[j] > arr[j+1]:

                arr[j], arr[j+1] = arr[j+1], arr[j]

                swapped = True

        if not swapped:

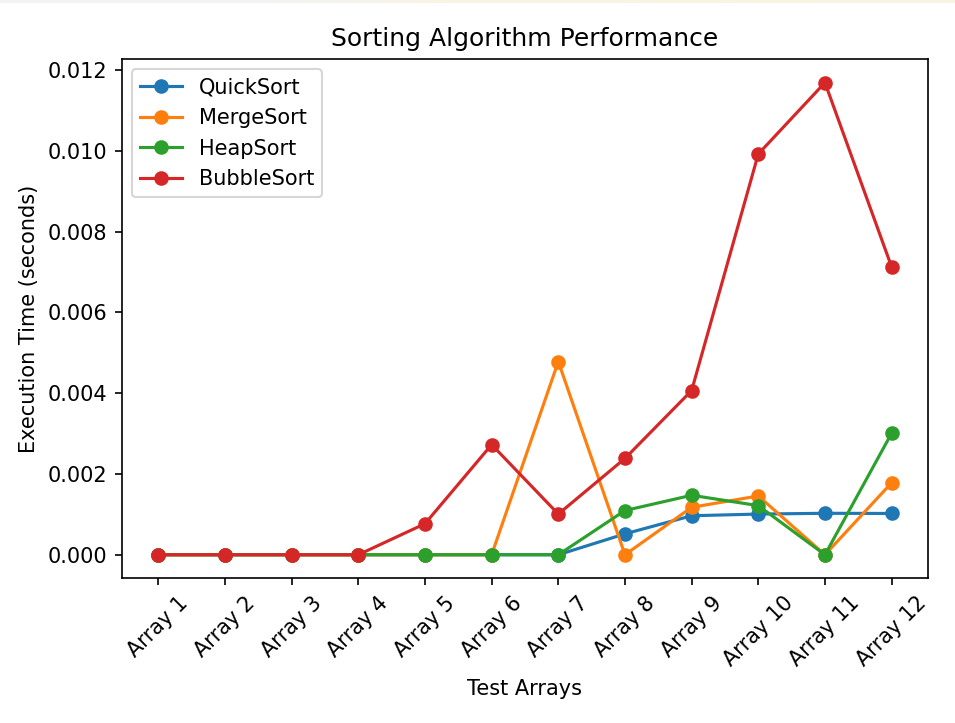
            break

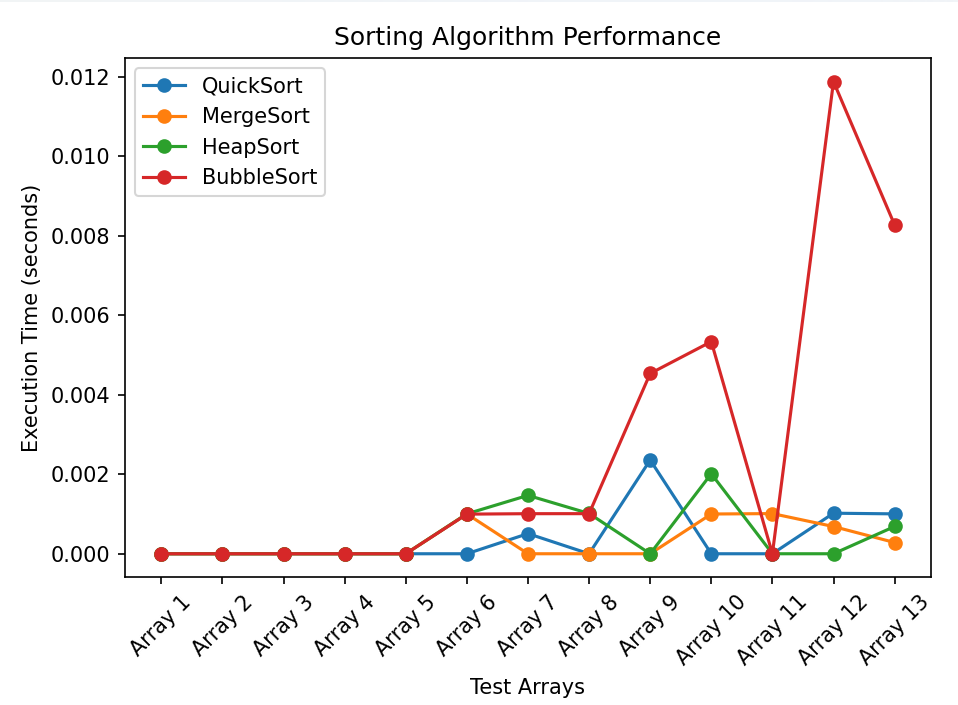
    return arr

* + Performs very poorly on large arrays, especially visible in the spike for larger datasets.
  + Works decently on already sorted or nearly sorted arrays due to the optimization of breaking early if no swaps occur.
  + Completely impractical for large input sizes due to its O(n²) complexity.
  + **Best Case:** O(n)(When the array is already sorted)
  + **Average Case:** O(n2)
  + **Worst Case:** O(n2)(Completely reversed array)
  + **Space Complexity:** O(1) (In-place sorting)

**Conclusions**

* **QuickSort** and **MergeSort** are the best choices for general sorting, with QuickSort often being slightly faster.
* **HeapSort** is a good alternative but not as efficient as QuickSort in most scenarios.
* **BubbleSort** should only be used for small or nearly sorted datasets due to its inefficiency on larger inputs.

1. 



Test Arrays:

  [5, 3, 8, 6, 2, 7, 4, 1],#1

    [10, 9, 8, 7, 6, 5, 4, 3, 2, 1],#2

    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],#3

    [random.randint(0, 100) for \_ in range(10)],#4

    [random.randint(0, 200) for \_ in range(50)],#5

    [random.randint(0, 500) for \_ in range(100)],#6

    [random.randint(0, 700) for \_ in range(150)],#7

    [random.randint(0, 1000) for \_ in range(200)],#8

    [random.randint(0, 1300) for \_ in range(250)],#9

    [random.randint(0, 1500) for \_ in range(300)],#10

    [random.randint(0, 1700) for \_ in range(350)],#11

    [random.randint(0, 100) for \_ in range(400)],#12

[random.randint(0, 10) for \_ in range(400)]#13