BUBBLE AND MERGE SORT

Done By/ Ahmed Maher Almaqtari

BUBBLE SORT

- **Bubble sort** is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the entire list is sorted.

- **Graphical illustration**

- Fun Example Video: https://www.youtube.com/watch?v=lyZQPjUT5B4

BUBBLE SORT – HOW IT WORKS

1. It compares the adjacent element.
2. If the adjacent element is out of order, then swap both elements.
3. If it's not out of order, then no changes are needed.
4. For the 'n' number of elements, we need to do 'n-1' comparisons, for the 'n-1' number of elements, we need to do the 'n-2' number of comparisons. So for the first pass, we have 5 elements, therefore 4 comparisons will happen, for the next pass, we have 4 elements and 3 comparisons will happen, and so on. **So, the first pass will sort the last element, the second pass will sort the second last element, and so on.**

Bubble Sort - Pros & Cons

Pros

- **Simple Implementation**: Bubble Sort is easy to understand and implement, making it a good choice for small datasets or educational purposes.
- **In-place Sorting**: Bubble Sort operates directly on the input array, requiring no additional memory space for sorting.
- **Adaptive**: It performs well on partially sorted or nearly sorted arrays.

Cons

- **Inefficiency**: Bubble Sort has a worst-case and average-case time complexity of O(n^2), making it highly inefficient for large datasets.
- **Lack of Efficiency for Reversed Arrays**: Bubble Sort performs poorly on arrays in reverse order as it requires multiple passes to swap adjacent elements.
- **Lack of Efficiency for Random Order**: Even with random order arrays, Bubble Sort has to perform unnecessary comparisons and swaps, resulting in inefficiency.

BUBBLE SORT - EXAMPLE

main.py > ...

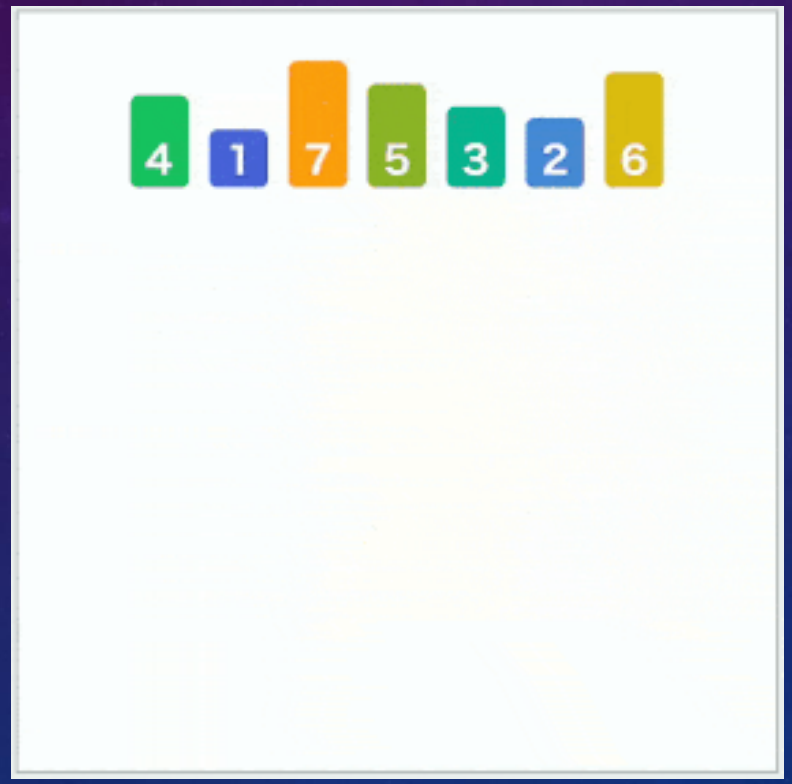
```python
def bubble_sort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n-1):

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # Swap if the element found is greater than the next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

# Example usage
my_list = [5,9,6,4,7,2,3,1,8]
bubble_sort(my_list)
print("Sorted array:", my_list)
```

OUTPUT

```
t1/main.py"
Sorted array: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

MERGE SORT

- **Merge sort** is a divide-and-conquer algorithm that divides the input array into two halves, recursively sorts each half, and then merges the sorted halves to produce a sorted output
- **Graphical illustration**



- Fun Example Video: https://www.youtube.com/watch?v=KF2j-9iSf4Q

MERGE SORT - HOW IT WORKS

1. Divide the problem into two sub-parts.
2. Solve them Recursively.
3. Merge the two sub-part using the merge algorithm.

MERGE SORT - PROS & CONS

PROS

- **Efficient Sorting**: Merge Sort has a time complexity of O(n log n) in all cases, making it suitable for large datasets.
- **Stability:** Merge Sort is a stable sorting algorithm, meaning it maintains the relative order of equal elements.
- **Out-of-place Sorting:** Merge Sort creates temporary arrays during the merging process, which can be advantageous when memory space is not a concern.

CONS

- **Extra Space Requirement:** Merge Sort requires additional memory space to store the temporary arrays during merging, which can be a drawback for memory-constrained environments.
- **Recursive Approach:** Merge Sort uses recursion, which may result in additional function call overhead and stack space usage.
- **Not Adaptive:** Merge Sort's performance remains consistent regardless of the initial order of the elements, making it inefficient for partially sorted arrays.

MERGE SORT - EXAMPLE

main.py > merge

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    # Divide the array into two halves
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # Recursively sort the two halves
    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    # Merge the sorted halves
    merged = merge(left_half, right_half)

    return merged
```

```python
def merge(left, right):
    merged = []
    i = j = 0

    # Compare elements from the left and right halves and merge them
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1

    # Append any remaining elements from the left and right halves
    while i < len(left):
        merged.append(left[i])
        i += 1
    while j < len(right):
        merged.append(right[j])
        j += 1

    return merged

# Example usage
my_list = [4,1,7,5,3,2,6]
sorted_list = merge_sort(my_list)
print("Sorted array:", sorted_list)
```

OUTPUT

```
Sorted array: [1, 2, 3, 4, 5, 6, 7]
```

Bubble Sort vs Merge Sort - Time Complexity

- **Bubble Sort**:
  - Bubble Sort has a worst-case and average-case time complexity of O(n^2), where n is the number of elements in the array.
- **Merge Sort**:
  - Merge Sort has a time complexity of O(n log n) in all cases. It consistently performs at this efficient time complexity.

BUBBLE SORT VS MERGE SORT – SPACE COMPLEXITY

- **Bubble Sort:**
  - Bubble Sort operates in-place, meaning it requires no additional memory beyond the input array.
- **Merge Sort:**
  - Merge Sort requires additional memory space for creating temporary arrays during the merging process. Its space complexity is O(n).

Bubble Sort vs Merge Sort - Adaptability

- **Bubble Sort:**
  - Bubble Sort performs well on partially sorted or nearly sorted arrays, as it can detect when the array is already sorted and terminate early.
- **Merge Sort:**
  - Merge Sort does not have an adaptive behavior. Its performance remains consistent regardless of the initial order of the elements.

Bubble Sort vs Merge Sort - Efficiency

- **Bubble Sort:**
  - Bubble Sort is inefficient for large datasets due to its O(n^2) time complexity. It requires multiple passes and unnecessary comparisons and swaps.
- **Merge Sort:**
  - Merge Sort is an efficient sorting algorithm with a consistent O(n log n) time complexity. It divides the problem into smaller subproblems and merges them in a sorted manner.

Bubble Sort vs Merge Sort – Implementation Complexity

- **Bubble Sort**:
  - Bubble Sort has a simple implementation, making it easier to understand and implement.
- **Merge Sort:**
  - Merge Sort has a more complex implementation, involving recursion and merging of sorted subarrays.

THE END...