Day-22

Python DSA

Merge Sort

https://www.geeksforgeeks.org/problems/merge-sort/1

Solution

```
class Solution:


  # Function to merge two halves of the array arr[l..m] and arr[m+1..r].
  def merge(self, arr, l, m, r):
    """
    Merges two sorted sub-arrays: arr[l..m] and arr[m+1..r] back into arr[l..r].
    """


    # Determine the sizes of the two sub-arrays to be merged
    n1 = m - l + 1
    n2 = r - m


    # Create temporary arrays
    # L holds arr[l..m]
    L = [0] * n1
    # R holds arr[m+1..r]
    R = [0] * n2


    # Copy data to temp arrays L[] and R[]
    for i in range(n1):
```

```
    L[i] = arr[l + i]

for j in range(n2):
    R[j] = arr[m + 1 + j]

# Merge the temp arrays back into arr[l..r]

i = 0  # Initial index of first sub-array
j = 0  # Initial index of second sub-array
k = l  # Initial index of merged sub-array

# Compare elements of L and R and place the smaller one into arr
while i < n1 and j < n2:
    if L[i] <= R[j]:
        arr[k] = L[i]
        i += 1
    else:
        arr[k] = R[j]
        j += 1
    k += 1

# Copy the remaining elements of L[], if any
while i < n1:
    arr[k] = L[i]
    i += 1
    k += 1
```

```
    # Copy the remaining elements of R[], if any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

# Function to sort the array using merge sort algorithm.
def mergeSort(self, arr, l, r):
    """
    Sorts the array arr[l..r] using the merge sort algorithm.

    :param arr: The array to be sorted.
    :param l: The starting index of the sub-array (left).
    :param r: The ending index of the sub-array (right).
    """
    if l < r:
        # Find the middle point
        m = l + (r - l) // 2

        # Sort first and second halves recursively
        self.mergeSort(arr, l, m)
        self.mergeSort(arr, m + 1, r)

        # Merge the two sorted halves
        self.merge(arr, l, m, r)
```

Time Complexity- O(NlogN)- The complexity is consistent because the division and merging process is always the same, regardless of whether the array is sorted, reverse-sorted, or random.

Space Complexity – O(N)- Required for the temporary arrays (L and R) used in the `merge` function.

Quick Sort

https://www.geeksforgeeks.org/problems/quick-sort/1

```
class Solution:

def quickSort(self, arr, low, high):
    """
    Recursively sorts the sub-array arr[low...high] using Quick Sort.

    :param arr: The list/array to be sorted.
    :param low: The starting index of the sub-array.
    :param high: The ending index of the sub-array.
    """
    # Base case: If the sub-array has one or zero elements, it's already sorted.
    if low < high:

        # 1. Divide: partition the array into two halves around the pivot.
        # p_index is the final resting position of the pivot element,
        # ensuring all elements to its left are smaller and to its right are greater.
        p_index = self.partition(arr, low, high)

        # 2. Conquer (Left): Recursively sort the sub-array of smaller elements.
        # The pivot is at p_index, so we sort arr[low...p_index - 1].
        self.quickSort(arr, low , p_index - 1)

        # 3. Conquer (Right): Recursively sort the sub-array of greater elements.
        # We sort arr[p_index + 1...high].
        self.quickSort(arr, p_index + 1, high)
```

```
def partition(self, arr, low, high):
    """
    Partitions the sub-array arr[low...high] around a pivot element.
    This implementation uses the 'Lomuto/Hoare'-hybrid partitioning scheme
    where the first element (arr[low]) is chosen as the pivot.

    :param arr: The list/array containing the sub-array to partition.
    :param low: The starting index.
    :param high: The ending index.
    :return: The final index of the pivot element.
    """

    # 1. Choose the pivot. Here, the first element of the sub-array is selected.
    pivot = arr[low]

    # Initialize two pointers:
    # i starts from the beginning (low) and searches for elements > pivot.
    i = low
    # j starts from the end (high) and searches for elements <= pivot.
    j = high

    # The main partitioning loop. It stops when the pointers cross or meet.
    while i < j:

        # Increment 'i' while elements are less than or equal to the pivot (in their correct partition).
        # The check 'i <= high - 1' prevents 'i' from going out of bounds.
        while arr[i] <= pivot and i <= high - 1:
            i += 1

        # Decrement 'j' while elements are strictly greater than the pivot (in their correct partition).
        # The check 'j >= low + 1' prevents 'j' from going out of bounds towards the pivot's index (low).
```

```
    while arr[j] > pivot and j >= low + 1:
        j -= 1

    # If 'i' is still less than 'j', we found a pair of elements out of order:
    # arr[i] is too large (belongs on the right side), and arr[j] is too small (belongs on the left side).
    if i < j:
        # Swap the out-of-order elements to put them into their correct partitions.
        arr[i], arr[j] = arr[j], arr[i]

  # Final step: Place the pivot element (originally at arr[low]) into its sorted position.
  # This is the position where the two pointers 'i' and 'j' have crossed (or 'j' stopped).
  # We swap the pivot with arr[j] because 'j' points to the last element of the "less than pivot" partition.
  arr[low], arr[j] = arr[j], arr[low]

  # Return the final index of the pivot. This index is used by quickSort to define the two sub-arrays.
  return j
```

Case	Time Complexity	Notes
Worst Case	$O(N^2)$	Happens when the pivot is always the smallest or largest element. This leads to highly unbalanced partitions (e.g., an already sorted or reverse-sorted array when the first element is chosen as pivot). In this case, recursion depth becomes N, and each level takes O(N) time.
Average Case	O(N log N)	Happens when the pivot divides the array into roughly equal halves. The recursion depth is log N, and work done at each level is O(N).
Best Case	O(N log N)	Happens when the pivot always results in perfect halving of the array, leading to the most balanced partitions.

Case	Space Complexity	Notes

Worst Case	O(N)	Required for the recursion stack when the partitions are highly unbalanced (recursion depth = O(N)), which usually happens in the $O(N^2)$ time case.
Average / Best Case	O(log N)	Required for the recursion stack when the partitions are balanced, leading to a maximum recursion depth of log N.