Assignment - 03

TASK 01:

```
*Merge(a, b):
  - Combined the elements of a and b into a single list
  - Sorted the combined list by using built-in sort()
  - Return the sorted list

*mergesort(arr):
  - If length of arr is 1:
    - Return arr (base case)
  - Finding the middle index (mid) of arr by using //
  - Splitting array into two subarrays: left (arr[:mid]) and right (arr[mid:])
  - Recursively sort left and right sub-arrays using merge sort
  - Merge the sorted left and right sub-arrays using merge function
  - Return the merged sorted array
```

TASK 02:

```
*Max_finder(arr):
  if length(arr) <= 1: In case the list is empty or single element
    return arr
*finding the midpoint index by using //
   mid = length(arr) // 2
*Splitting the array into left and right
  left_max = Max_finder(arr[:mid])
  right_max = Max_finder(arr[mid:])

*checking which is bigger by comparing
  if left_max > right_max:
    return left_max
  else:
    return right_max
```

TASK 03:

```
* function merge_Sort(arr, temp_arr, left, right):
  - If left index is less than right index:
    - Find the middle index (mid) by using //
    - Recursively call merge_Sort on left half (left, mid) and right half (mid+1,right)
      - Count inversions in each half
    - Merge the sorted left and right halves using merge function
      - Count inversions during merging
    - Return total inversion count
  - Otherwise :
    - Return 0 (In case of no inversions)

*function merge(arr, temp_arr, left, mid, right):
  - Initialize variables for indices and inversion count
  - a temporary array to hold merged elements
  - While both halves have elements:
    - If left element is less than or equal to right element:
      - Add left element to temp array and increment indices
    - Otherwise:
      - Add right element to temp array, increment right index, and
        - Add number of remaining elements in left half to inversion count
  - Move remaining elements from both halves to temp array
  - Copy temp array back to original array
  - Return inversion count
```

TASK 04:

```
*function divide_and_conquer(arr):
  - if single element or empty array
    - Return the array (maximum element is the only element)
  - Recursive case:
    - Find middle index (`mid`) of the array by using //
    - Divide array into left (`Left`) and right (`Right`) sub-arrays
    - Recursively find maximum element in each sub-array
    - Compare the maximum elements from both sub-arrays
    - Return the larger maximum element

*function max_pair_finder(arr, n):
  - Initialize `max_pair` to 0
  - Iterate through the array (except last element)
    - Get current element (`x`)
    - Find maximum element in remaining sub-array using `divide_and_conquer`
    - Calculate potential maximum pair value (`x + (merge_value**2)`)
    - Update `max_pair` with the maximum value between current `max_pair` and the calculated value
  - Return `max_pair`
```

TASK 05:

```
*function Partition(arr, low, high):
  - Chose the last element (`arr[high]`) as the pivot
  - Set a variable i to low - 1 (index before the first element)
  - Iterate through the array from low to second-last element (high - 1)
    - If the current element (arr[j]) is less than or equal to the pivot:
      - Increased i by 1
      - Swapped the elements at i and j
  - Swapped the pivot element (`arr[high]`) with the element at i + 1
  - Return the index i + 1 (partitioning point)

*function Quicksort(arr, low, high):
  - If low is less than high:
    - Find the partition index (part) using the Partition function
    - Recursively sort the left sub-array (`arr[low, part-1]`)
    - Recursively sort the right sub-array (`arr[part+1, high]`)
    - Array is sorted no need to return
```

TASK 06:

```
*function Partition(arr, low, high):
  - Chose the last element (arr[high]) as the pivot
  - Partition the array around the pivot
  - Return the partition index


*function Kth_smallest(arr, low, high, k):
  - if single element or empty
    - Return the element at the given index
  - Find the partition index using Partition
  - Calculate the number of elements (i) in the left sub-array (including the pivot)
  - If i is equal to k:
    - Return the pivot element
  - If i is greater than k:
    - Recursively search for the k th smallest element in the left sub-array
  - Otherwise:
    - Recursively search for the k - i th smallest element in the right sub-array
```