Assignment-12

TASK 1 :

```python
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            # Traverse the array from 0 to 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]

    return arr

# Example usage with user input:
user_input = input("Enter a list of numbers separated by spaces: ")
my_list = list(map(int, user_input.split()))

sorted_list = bubble_sort(my_list)
print("Sorted array:", sorted_list)
```

```
Enter a list of numbers separated by spaces: 55 25 96 5 91 62 33 21 77
Sorted array: [5, 21, 25, 33, 55, 62, 77, 91, 96]
```

EXPLANATION :

1. `def bubble_sort(arr):` : This defines a function named `bubble_sort` that takes one argument, `arr`, which is the list of numbers you want to sort.
2. `n = len(arr)` : This line gets the number of elements in the input list `arr` and stores it in the variable `n`.
3. `for i in range(n):` : This is the outer loop. It iterates `n` times. Each iteration of this outer loop is considered a "pass" through the list. After each pass, the largest unsorted element "bubbles up" to its correct position at the end of the unsorted portion of the list.
4. `for j in range(0, n - i - 1):` : This is the inner loop. It iterates through the unsorted portion of the list. The range `(0, n - i - 1)` is used because after each pass `i`, the last `i` elements are already sorted and don't need to be compared again.
5. `if arr[j] > arr[j + 1]:` : This is the core of the Bubble Sort algorithm. It compares adjacent elements (`arr[j]` and `arr[j + 1]`).
6. `arr[j], arr[j + 1] = arr[j + 1], arr[j]` : If the element at index `j` is greater than the element at index `j + 1`, this line swaps their positions. This is done using Python's tuple packing and unpacking feature.
7. `return arr` : After the loops complete, the function returns the sorted list.

The code then demonstrates how to use this function:

- It prompts the user to enter a list of numbers separated by spaces.
- It reads the input, splits the string by spaces, and converts each part into an integer using `map(int, user_input.split())`. This creates the list `my_list`.
- It calls the `bubble_sort` function with `my_list` to get the sorted list.
- Finally, it prints the sorted list.

TASK 2 :

```python
import time
import copy
import random

def bubble_sort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            # Traverse the array from 0 to 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]
    return arr

def insertion_sort(arr):
    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):
        key = arr[i]
        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

def generate_partially_sorted_array(size, disorder_level):
    """Generates a partially sorted array."""
    # Create a sorted array
    arr = list(range(size))

    # Introduce disorder by swapping elements
    for _ in range(disorder_level):
        idx1, idx2 = random.sample(range(size), 2)
        arr[idx1], arr[idx2] = arr[idx2], arr[idx1]

    return arr
```

```python
# Define array size and disorder level
array_size = 1000
disorder_level = 50

# Generate a partially sorted array
partially_sorted_array = generate_partially_sorted_array(array_size, disorder_level)

# Create a copy of the array for the second algorithm
partially_sorted_array_copy = copy.deepcopy(partially_sorted_array)

# Measure execution time for Bubble Sort
start_time_bubble = time.time()
bubble_sort(partially_sorted_array)
end_time_bubble = time.time()
duration_bubble = end_time_bubble - start_time_bubble

# Measure execution time for Insertion Sort
start_time_insertion = time.time()
insertion_sort(partially_sorted_array_copy)
end_time_insertion = time.time()
duration_insertion = end_time_insertion - start_time_insertion

# Print the execution times
print(f"Execution time for Bubble Sort: {duration_bubble:.6f} seconds")
print(f"Execution time for Insertion Sort: {duration_insertion:.6f} seconds")
```

```
Execution time for Bubble Sort: 0.056394 seconds
Execution time for Insertion Sort: 0.006146 seconds
```

EXPLANATION :

Imagine you have a pile of numbered cards that are almost sorted, but a few are in the wrong place. The code compares two ways of sorting these cards:

1. **Bubble Sort (Like a clumsy cleaner):**
    - This method looks at two cards next to each other. If they are in the wrong order, it swaps them.
    - It does this again and again, going through the whole pile many times.
    - It's like a cleaner who keeps sweeping the floor, pushing the biggest dust bunnies to the end with each sweep. Even if the floor is mostly clean, they still have to sweep almost the whole floor every time.
    - Because it always compares neighbors and makes many passes, it's slow, especially if the cards aren't perfectly sorted.
2. **Insertion Sort (Like sorting playing cards in your hand):**
    - This method takes one card at a time from the unsorted pile.
    - It then finds the correct spot for that card among the cards that are already sorted and inserts it there.
    - Think about sorting playing cards: you pick up a new card and slide it into the right spot in your hand. If your hand is already mostly sorted, you only need to slide the new card a short distance.
    - Because the cards are *almost* sorted, Insertion Sort finds the right spot quickly and doesn't have to move cards very far. This makes it much faster for this type of almost-sorted list.

The Code Does This:

- It sets up two lists of numbers that are almost sorted (like our cards).
- It uses a stopwatch (`time.time()`) to time how long Bubble Sort takes to sort one list.
- It uses another stopwatch to time how long Insertion Sort takes to sort the *exact same* list (the copy).
- Finally, it prints the times. You'll see that the time for Insertion Sort is much smaller, showing it's faster for this "partially sorted" situation.

In short, for lists that are already mostly in order, Insertion Sort is smarter and faster because it can place elements correctly with less effort than Bubble Sort, which always does a lot of unnecessary comparisons and swaps.

TASK 3 :

```python
def linear_search(arr, target):
    """
    Performs a linear search to find the target value in the given array.

    Args:
        arr: The list or array to search within.
        target: The value to search for.

    Returns:
        The index of the target in the array if found, otherwise -1.

    Performance Notes:
        - Time Complexity: O(n) in the worst and average cases, where 'n' is the number of elements in the array.
                           O(1) in the best case (target is the first element).
        - Space Complexity: O(1) as it uses a constant amount of extra space.
        - Suitable for small to medium-sized arrays or when the array is unsorted.
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Return the index if the target is found
    return -1  # Return -1 if the target is not found

# Example usage:
my_list = [10, 5, 8, 12, 1, 7]
target_value = 12
index = linear_search(my_list, target_value)
if index != -1:
    print(f"Target {target_value} found at index {index}")
else:
    print(f"Target {target_value} not found in the list")
```

```
Target 12 found at index 3
```

EXPLANATION :

Here's how it works:

1. `def linear_search(arr, target):` : This line defines the function `linear_search` which takes the list (`arr`) and the value you're looking for (`target`) as input.
2. `""" ... """` : This is a docstring. It's like a helpful note within the code that explains what the function does, what arguments it takes (`Args`), and what it gives back (`Returns`). It also includes "Performance Notes" about how fast and memory-efficient the algorithm is.
3. `for i in range(len(arr)):` : This is the main part of the search. It starts a loop that goes through each element in the list one by one. The variable `i` represents the current position (index) in the list.
4. `if arr[i] == target:` : Inside the loop, this line checks if the number at the current position (`arr[i]`) is equal to the `target` value you are looking for.
5. `return i` : If the number at the current position *is* the target, the function immediately stops and returns the position (`i`) where it found the target.
6. `return -1` : If the loop finishes without finding the target value in the list, the function returns `-1`. This is a common way to indicate that the target was not found.

TASK 4 :

```python
def quick_sort(arr):
    """
    Sorts a list using the Quick Sort algorithm with recursion.

    Args:
        arr: The list to be sorted.

    Returns:
        The sorted list.

    Performance Notes:
        - Average Time Complexity: O(n log n)
        - Worst-Case Time Complexity: O(n^2) (can occur with poor pivot selection)
        - Space Complexity: O(log n) on average (due to recursion stack)
                            O(n) in worst case
    """
    # Base case: If the list is empty or contains only one element, it's already sorted.
    if len(arr) <= 1:
        return arr

    # Recursive step:
    # 1. Choose a pivot element (e.g., the last element).
    pivot = arr[-1]

    # 2. Partition the list into three sub-lists:
    #    - Elements less than the pivot
    #    - Elements equal to the pivot
    #    - Elements greater than the pivot
    less = []
    equal = []
    greater = []

    # TODO: Implement the partitioning logic here.
    # Iterate through the array and append elements to the appropriate lists.
    for x in arr:
        if x < pivot:
            less.append(x)
        elif x == pivot:
            equal.append(x)
        else:
            greater.append(x)
```

```python
    # 3. Recursively sort the 'less' and 'greater' sub-lists.
    # 4. Combine the sorted 'less' list, the 'equal' list, and the sorted 'greater' list.
    # TODO: Implement the recursive calls and combining logic here.
    return quick_sort(less) + equal + quick_sort(greater)

# Example usage:
my_list = [10, 7, 8, 9, 1, 5]
sorted_list = quick_sort(my_list)
print("Sorted array:", sorted_list)
```

```
Sorted array: [1, 5, 7, 8, 9, 10]
```

EXPLANATION :

1. `def quick_sort(arr):` : This defines the function named `quick_sort` that takes the list to be sorted (`arr`) as input.
2. `""" ... """` : This is the docstring, explaining the function's purpose, arguments, return value, and important performance notes about Quick Sort (average and worst-case time complexity, and space complexity related to recursion).
3. `if len(arr) <= 1:` : This is the **base case** of the recursion. If the list has 0 or 1 elements, it's already sorted, so the function just returns the list as is. This stops the recursion from going on forever.
4. `pivot = arr[-1]` : This line chooses a **pivot** element. In this specific implementation, the last element of the list is chosen as the pivot. The goal is to arrange the list so that all elements smaller than the pivot are before it, and all elements larger are after it.
5. `less = []`, `equal = []`, `greater = []` : These lines create three empty lists that will be used to hold elements less than the pivot, equal to the pivot, and greater than the pivot, respectively.
6. `for x in arr:` : This loop iterates through each element (`x`) in the input list (`arr`).
7. `if x < pivot: ... elif x == pivot: ... else: ...` : This is the **partitioning logic**. For each element `x`, it compares it to the `pivot` and appends it to the appropriate list (`less`, `equal`, or `greater`).
8. `return quick_sort(less) + equal + quick_sort(greater)` : This is the **recursive step** and the **combining step**.
    - `quick_sort(less)` : It recursively calls `quick_sort` on the `less` list. This means Quick Sort is applied again to sort the smaller elements.
    - `quick_sort(greater)` : It recursively calls `quick_sort` on the `greater` list to sort the larger elements.
    - `+ equal +` : The `equal` list (containing elements equal to the pivot) is placed in between the sorted `less` list and the sorted `greater` list.
    - The `+` operator here concatenates (joins) the three lists together to form the final sorted list.

TASK 5 :

```python
def find_duplicates_naive(arr):
    """
    Finds duplicate elements in a list using a naive brute-force approach (O(n^2)).

    Args:
        arr: The list to search for duplicates.

    Returns:
        A list of unique duplicate elements found in the input list.
    """
    duplicates = []
    n = len(arr)
    # Iterate through each element
    for i in range(n):
        # Compare the current element with all subsequent elements
        for j in range(i + 1, n):
            # If a duplicate is found and it's not already in the duplicates list
            if arr[i] == arr[j] and arr[i] not in duplicates:
                duplicates.append(arr[i])
    return duplicates

#Example usage:
my_list = [1, 2, 3, 4, 2, 5, 6, 3, 7, 8, 8]
duplicate_items = find_duplicates_naive(my_list)
print("Duplicate elements:", duplicate_items)
```

```
Duplicate elements: [2, 3, 8]
```

Explanation :

1. `def find_duplicates_naive(arr):` : This defines the function named `find_duplicates_naive` that takes the list to search within (`arr`) as input.
2. `""" ... """` : This is the docstring, providing a description of what the function does, its arguments, return value, and importantly, its performance characteristics (noting it's an $O(n^2)$ approach).
3. `duplicates = []` : This initializes an empty list called `duplicates` . This list will store the unique duplicate elements found in the input list.
4. `n = len(arr)` : This gets the number of elements in the input list and stores it in the variable `n` .
5. `for i in range(n):` : This is the **outer loop**. It iterates through each element in the list, starting from the first element (index 0) up to the last element.
6. `for j in range(i + 1, n):` : This is the **inner loop**. For each element at index `i` in the outer loop, this inner loop iterates through the *remaining* elements in the list, starting from the element *after* the current one (index `i + 1`) up to the last element.
7. `if arr[i] == arr[j] and arr[i] not in duplicates:` : This is the core logic. It compares the element at index `i` with the element at index `j` .
    - `arr[i] == arr[j]` : Checks if the two elements are the same.
    - `arr[i] not in duplicates` : Checks if this duplicate value has *already* been added to our `duplicates` list. This ensures that each duplicate value is added only once to the result list.
8. `duplicates.append(arr[i])` : If both conditions in the `if` statement are true (the elements are the same, and this duplicate hasn't been recorded yet), the duplicate element (`arr[i]`) is added to the `duplicates` list.
9. `return duplicates` : After both loops have finished checking all possible pairs of elements, the function returns the `duplicates` list containing all the unique duplicate elements found.