Major Assignment - 01

STUDENT INFORMATION

Name:	SK SHAKEEL AKHTAR
Regd No.:	2341001063
Section:	2341-2C3
Branch:	CSE (AI & ML)

Topics Covered

- **Searching Algorithms**:
  - Linear Search
  - Binary Search
- **Sorting Algorithms**:
  - Bubble Sort
  - Insertion Sort
  - Selection Sort
  - Merge Sort
  - Quick Sort

The single Python script:

```
# Searching and Sorting Algorithms


# Linear Search
def linear_search(arr, target):
    """
    Perform linear search on a list.
    Time Complexity: O(n)
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Return index of the target element
    return -1  # Return -1 if target is not found
```

```python
# Binary Search (Iterative)
def binary_search(arr, target):
    """
    Perform binary search on a sorted list.
    Time Complexity: O(log n)
    """
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1


# Bubble Sort
def bubble_sort(arr):
    """
    Perform Bubble Sort.
    Time Complexity: O(n^2)
    """
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]


# Insertion Sort
```

```
def insertion_sort(arr):
    """
    Perform Insertion Sort.
    Time Complexity: O(n^2)
    """
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

# Selection Sort
def selection_sort(arr):
    """
    Perform Selection Sort.
    Time Complexity: O(n^2)
    """
    for i in range(len(arr)):
        min_idx = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

# Merge Sort
def merge_sort(arr):
    """
    Perform Merge Sort.
    Time Complexity: O(n log n)
```

```
    """
    if len(arr) > 1:
        mid = len(arr) // 2
        left = arr[:mid]
        right = arr[mid:]

        merge_sort(left)
        merge_sort(right)

        i = j = k = 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1

        while i < len(left):
            arr[k] = left[i]
            i += 1
            k += 1

        while j < len(right):
            arr[k] = right[j]
            j += 1
            k += 1

# Quick Sort
def quick_sort(arr):
```

```
    """
    Perform Quick Sort.
    Time Complexity: O(n log n) on average, O(n^2) in worst case.
    """
    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 quick_sort(left) + middle + quick_sort(right)

# Test cases
if __name__ == "__main__":
    # Unordered test list
    test_list = [64, 34, 25, 12, 22, 11, 90]

    # Searching algorithms
    print("Linear Search:", linear_search(test_list, 22))  # Example target: 22
    test_list_sorted = sorted(test_list)
    print("Binary Search:", binary_search(test_list_sorted, 22))

    # Sorting algorithms
    bubble_sorted = test_list.copy()
    bubble_sort(bubble_sorted)
    print("Bubble Sort:", bubble_sorted)

    insertion_sorted = test_list.copy()
    insertion_sort(insertion_sorted)
    print("Insertion Sort:", insertion_sorted)
```

```
selection_sorted = test_list.copy()
selection_sort(selection_sorted)
print("Selection Sort:", selection_sorted)

merge_sorted = test_list.copy()
merge_sort(merge_sorted)
print("Merge Sort:", merge_sorted)

quick_sorted = quick_sort(test_list.copy())
print("Quick Sort:", quick_sorted)
```

<u>Outputs</u>

1. Searching Algorithms

- **Linear Search**
  Target: 22

Linear Search: 4

Explanation: The target 22 is at index 4 in the input list.

- **Binary Search**
  Target: 22 (using a sorted list)

Binary Search: 2

Explanation: The sorted list is [11, 12, 22, 25, 34, 64, 90], and the target 22 is at index 2.

---

2. Sorting Algorithms

- **Bubble Sort**

Bubble Sort: [11, 12, 22, 25, 34, 64, 90]

- **Insertion Sort**

Insertion Sort: [11, 12, 22, 25, 34, 64, 90]

- **Selection Sort**

Selection Sort: [11, 12, 22, 25, 34, 64, 90]

- **Merge Sort**

Merge Sort: [11, 12, 22, 25, 34, 64, 90]

- **Quick Sort**

Quick Sort: [11, 12, 22, 25, 34, 64, 90]

Comparative Report Analysis :

Algorithm	Best Case	Worst Case	Average Case	Space Complexity	Stability
Linear Search	O(1)	O(n)	O(n)	O(1)	N/A
Binary Search	O(1)	O(logn)	O(logn)O	O(1)	N/A
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Stable
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)	Stable
Selection Sort	O(n^2)	O(n^2)	O(n2)	O(1)	Unstable
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	Stable
Quick Sort	O(nlogn)	O(n^2)	O(nlogn)	O(logn)	Unstable

Here are the pseudocodes for the algorithms provided in the Python script:

---

1. Linear Search

Input: List arr of size n, Target target
Output: Index of the target if found, otherwise -1

```
Procedure LinearSearch(arr, target)
  For i from 0 to n-1
    If arr[i] == target
      Return i
  End For
  Return -1
End Procedure
```

---

2. Binary Search (Iterative)

Input: Sorted list arr of size n, Target target
Output: Index of the target if found, otherwise -1

```
Procedure BinarySearch(arr, target)
```

```
  left ← 0
  right ← n - 1
  While left ≤ right
    mid ← (left + right) / 2
    If arr[mid] == target
      Return mid
    Else If arr[mid] < target
      left ← mid + 1
    Else
      right ← mid - 1
  End While
  Return -1
End Procedure
```

---

3. Bubble Sort

Input: List arr of size n
Output: Sorted list

```
Procedure BubbleSort(arr)
  For i from 0 to n-1
    For j from 0 to n-i-2
      If arr[j] > arr[j+1]
        Swap arr[j] and arr[j+1]
      End If
    End For
  End For
End Procedure
```

---

4. Insertion Sort

Input: List arr of size n
Output: Sorted list

```
Procedure InsertionSort(arr)
  For i from 1 to n-1
```

```
        key ← arr[i]
        j ← i - 1
        While j ≥ 0 AND arr[j] > key
            arr[j+1] ← arr[j]
            j ← j - 1
        End While
        arr[j+1] ← key
    End For
End Procedure
```

---

5. <u>Selection Sort</u>

Input: List arr of size n
Output: Sorted list

```
Procedure SelectionSort(arr)
    For i from 0 to n-1
        min_idx ← i
        For j from i+1 to n-1
            If arr[j] < arr[min_idx]
                min_idx ← j
            End If
        End For
        Swap arr[i] and arr[min_idx]
    End For
End Procedure
```

---

6. <u>Merge Sort</u>

Input: List arr of size n
Output: Sorted list

```
Procedure MergeSort(arr)
    If n > 1
        mid ← n / 2
        left ← arr[0...mid-1]
```

```
        right ← arr[mid...n-1]

        MergeSort(left)
        MergeSort(right)

        i ← 0, j ← 0, k ← 0
        While i < size(left) AND j < size(right)
            If left[i] < right[j]
                arr[k] ← left[i]
                i ← i + 1
            Else
                arr[k] ← right[j]
                j ← j + 1
            End If
            k ← k + 1
        End While

        While i < size(left)
            arr[k] ← left[i]
            i ← i + 1
            k ← k + 1
        End While

        While j < size(right)
            arr[k] ← right[j]
            j ← j + 1
            k ← k + 1
        End While
    End If
End Procedure
```

7. <u>Quick Sort</u>

Input: List arr of size n
Output: Sorted list

Procedure QuickSort(arr)

  If size(arr) ≤ 1

    Return arr

  End If

  pivot ← arr[n/2]

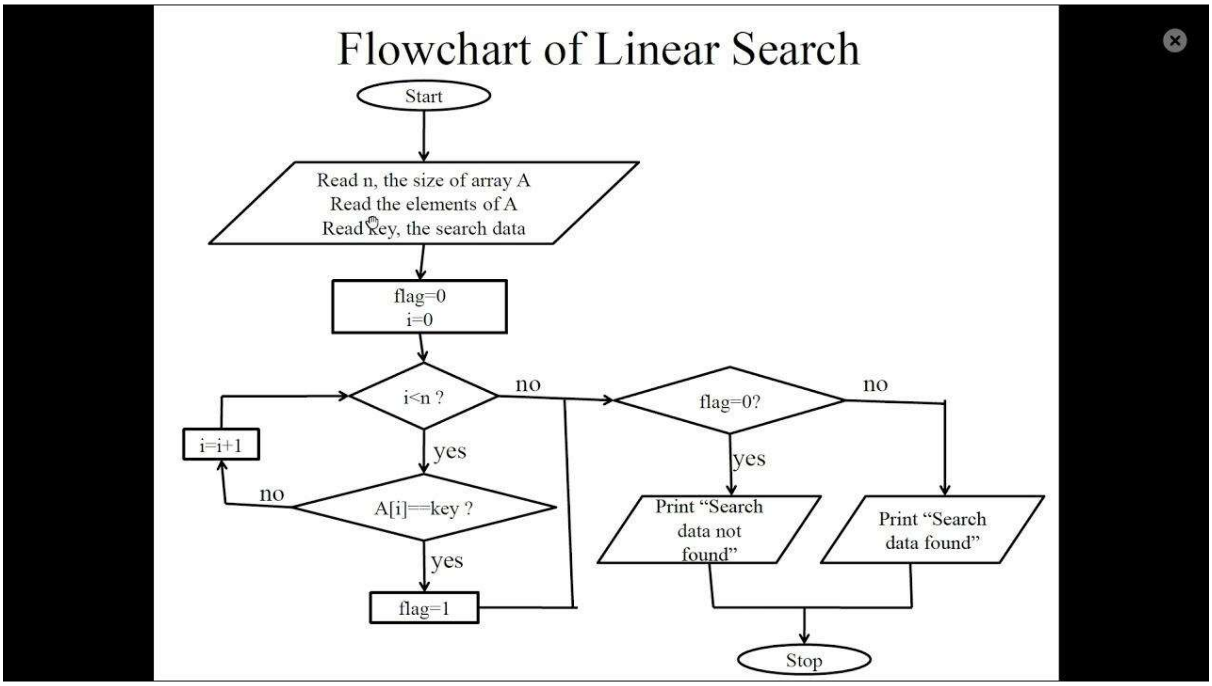
  left ← elements in arr less than pivot

  middle ← elements in arr equal to pivot

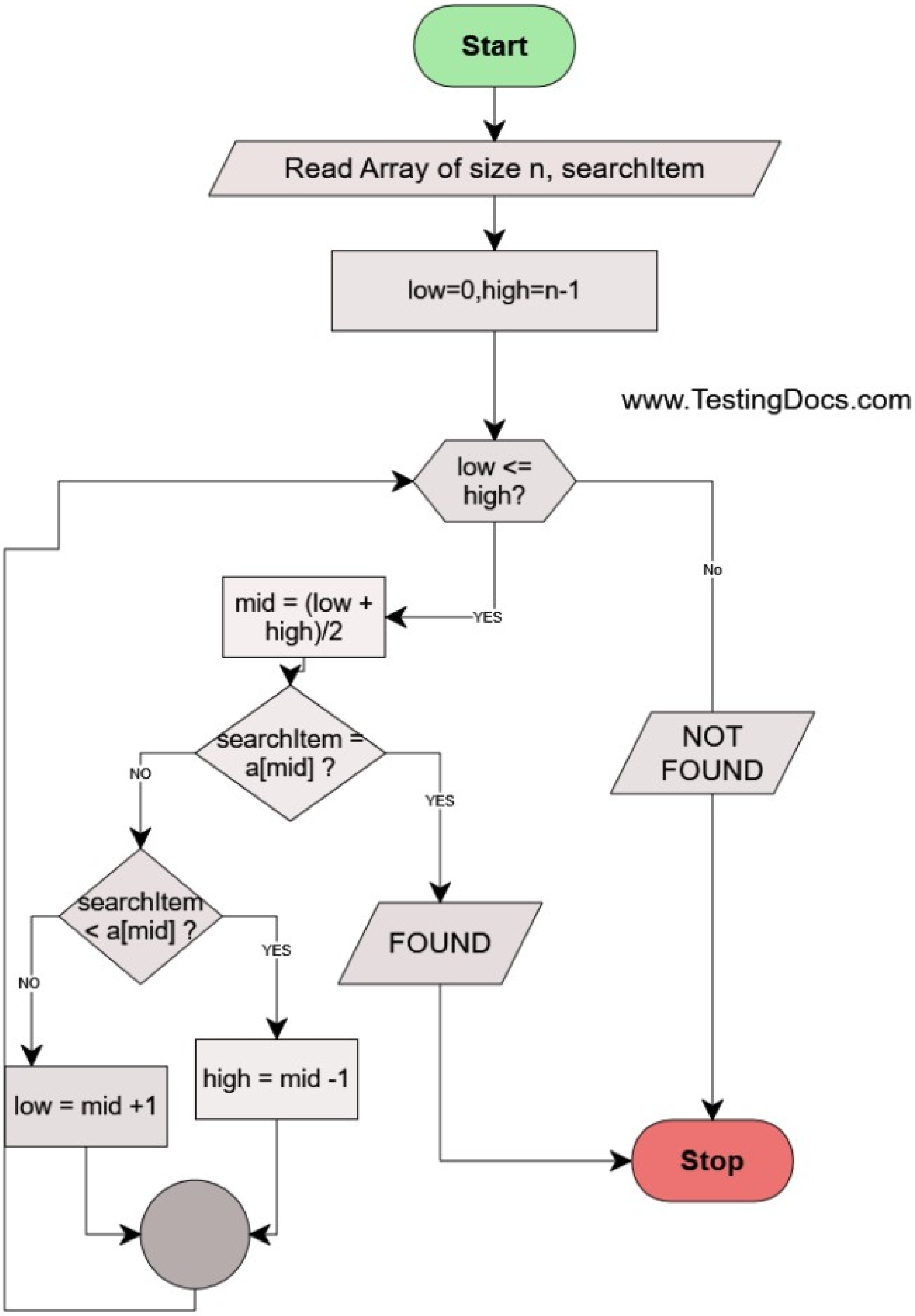
  right ← elements in arr greater than pivot

  Return QuickSort(left) + middle + QuickSort(right)

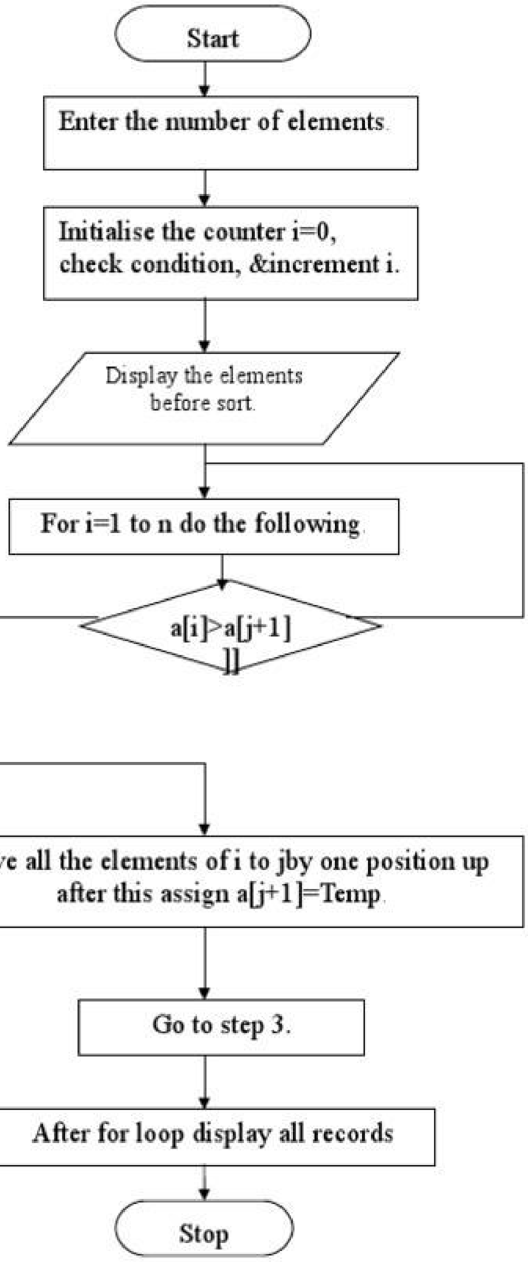
End Procedure



Binary Search

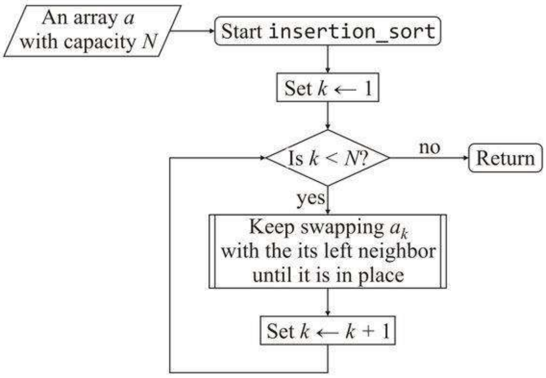


BUBBLE SORT FLOWCHART

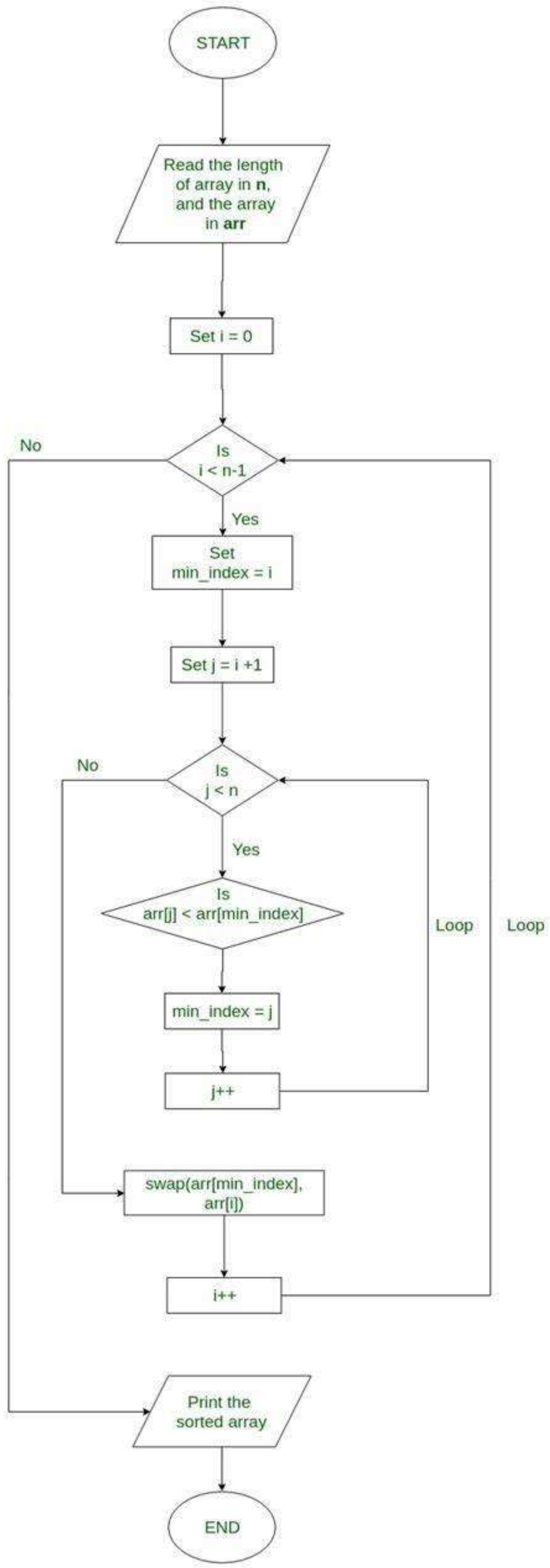


Insertion sort

- Here is a flow chart:

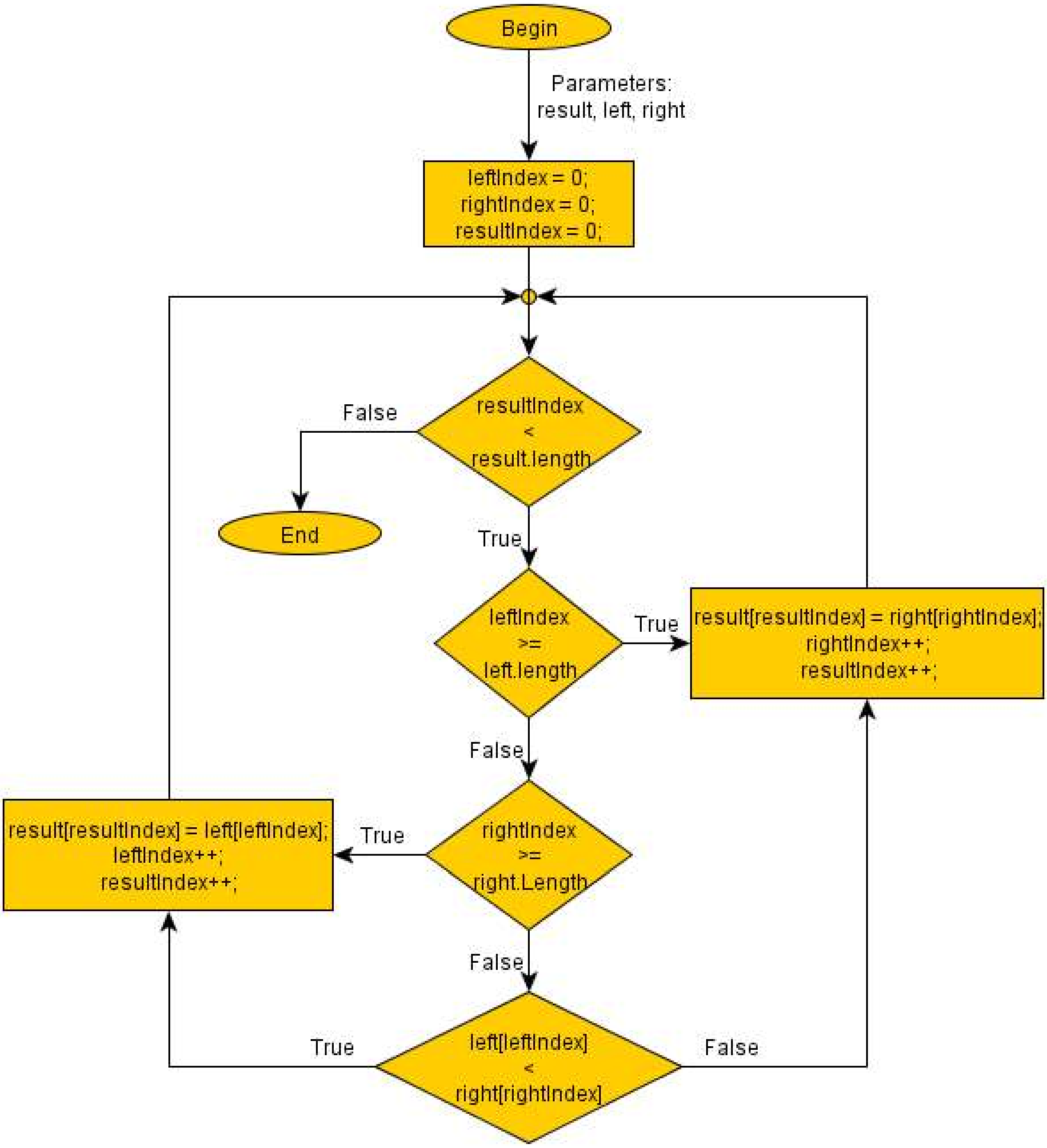


cc BY NC SA

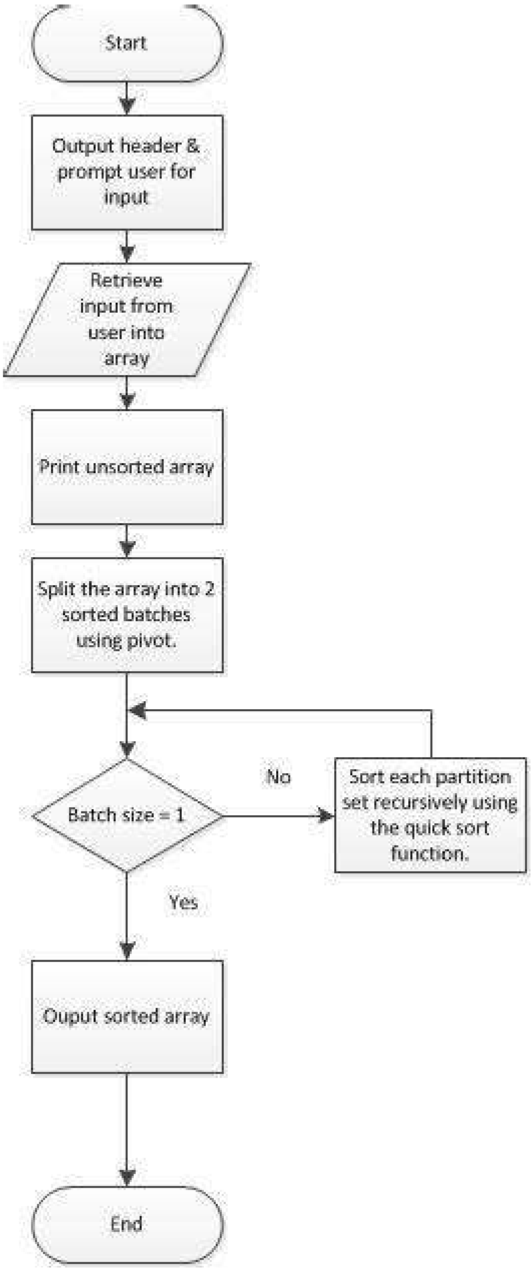


Flowchart for Selection Sort

FLOW CHART FOR MERGE SORT:



FLOW CHART FOR QUICK SORT:



Here's a **step-by-step explanation** of each algorithm:

---

1. Linear Search

1. Start at the first element of the list.
2. Compare the current element with the target value.
3. If they match, return the index of the current element.
4. If not, move to the next element.

5. Repeat steps 2–4 until the target is found or the list ends.
6. If the list ends without finding the target, return -1.

---

2. Binary Search

1. Ensure the list is sorted.
2. Set two pointers: left at the beginning and right at the end of the list.
3. Calculate the middle index: mid = (left + right) // 2.
4. Compare the middle element with the target value:
    - If equal, return mid.
    - If the target is smaller, set right = mid - 1.
    - If the target is larger, set left = mid + 1.
5. Repeat steps 3–4 until the target is found or the pointers cross.
6. If the pointers cross, return -1 (target not found).

---

3. Bubble Sort

1. Start at the first element and compare it with the next element.
2. If the first is greater than the second, swap them.
3. Move to the next pair and repeat step 2 until the end of the list.
4. After the first pass, the largest element will be at the correct position (end of the list).
5. Ignore the last element and repeat the process for the remaining list.
6. Continue until the entire list is sorted.

---

4. Insertion Sort

1. Start with the second element (index 1) as the key.
2. Compare the key with the elements before it.
3. Shift all larger elements one position to the right to make space for the key.
4. Place the key in its correct position.
5. Move to the next element and repeat steps 2–4 until the end of the list is processed.

---

5. Selection Sort

1. Divide the list into two parts: sorted (initially empty) and unsorted (entire list).

2. Find the smallest element in the unsorted part.
3. Swap it with the first element of the unsorted part.
4. Move the boundary of the sorted part one step to the right.
5. Repeat steps 2–4 until the entire list is sorted.

---

6. Merge Sort

1. Divide the list into two halves.
2. Keep dividing each half until every sublist has one element.
3. Merge the sublists back together in sorted order:
   - Compare the first elements of two sublists.
   - Add the smaller element to the sorted list.
   - Repeat until all elements are merged.
4. Continue merging until the entire list is sorted.

---

7. Quick Sort

1. Choose a pivot element (can be any element; typically, the last or a random element).
2. Partition the list into two parts:
   - Elements smaller than the pivot.
   - Elements larger than the pivot.
3. Place the pivot in its correct position.
4. Recursively apply the same steps to the left and right partitions.
5. Continue until all partitions have one or no elements, resulting in a sorted list.

Key Insights:

1. **Searching Algorithms**:
   - **Linear Search** is straightforward and works well for unsorted data.
   - **Binary Search** is much faster but requires sorted input.
2. **Sorting Algorithms**:
   - Simple algorithms like **Bubble Sort**, **Insertion Sort**, and **Selection Sort** are mostly used for educational purposes.
   - **Merge Sort** and **Quick Sort** are highly efficient for large datasets, with Quick Sort being faster in practice due to its lower constant factors in most cases.

3. **Recommendations**:
    - Use **Merge Sort** or **Quick Sort** for sorting large datasets.
    - Use **Binary Search** for searching in sorted data, otherwise, rely on **Linear Search**.