BRUTE FORCE

EXPERIMENT :1

Aim: To create and display lists with different elements and test their outputs.

Procedure:

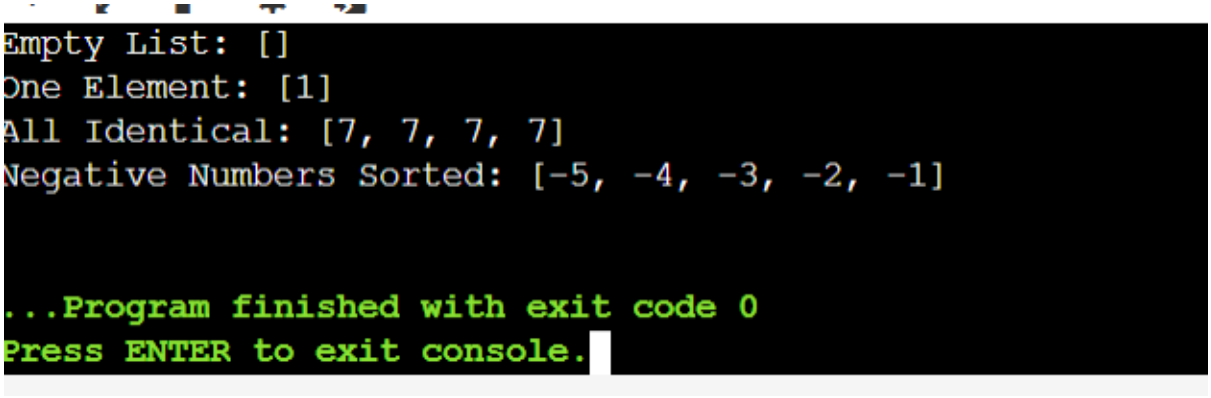
1. Create different lists – empty, single element, identical, and negative numbers.
2. Print each list.
3. Use sorted() for sorting where needed.
4. Display output clearly.
5. Verify expected results.

PROGRAM:

```
# Different types of lists
lst1 = []
lst2 = [1]
lst3 = [7, 7, 7, 7]
lst4 = [-5, -1, -3, -2, -4]

print("Empty List:", lst1)
print("One Element:", lst2)
print("All Identical:", lst3)
print("Negative Numbers Sorted:", sorted(lst4))
```

OUTPUT:



RESULT:

Different types of lists (empty, single element, identical, and negative numbers) were successfully created and displayed.

EXPERIMENT:2

AIM: To implement selection sort and understand its step-by-step working.

PROCEDURE:

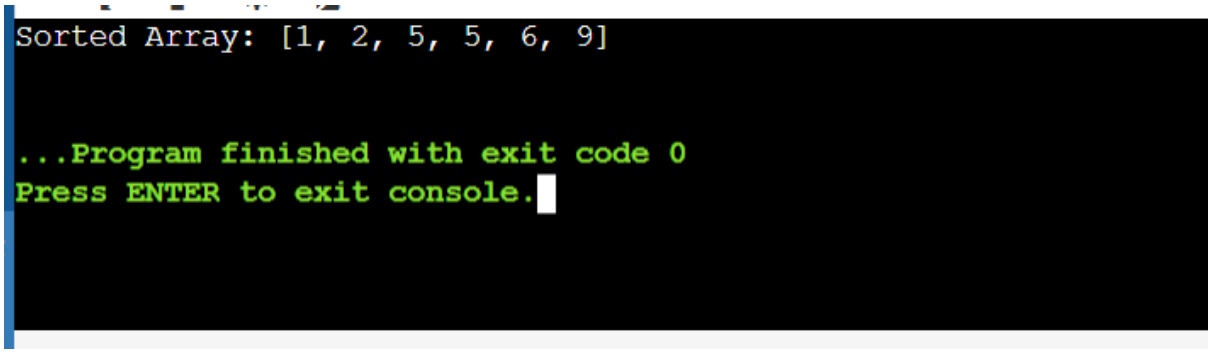
- Find the smallest element in the unsorted part.
- Swap it with the first unsorted element.
- Repeat until the entire list is sorted.
- Selection sort is easy but slow for large data.
- Display sorted output.

PROGRAM:

```python
def selection_sort(arr):
    for i in range(len(arr)):
        min_index = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_index]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr

# Example test
data = [5, 2, 9, 1, 5, 6]
print("Sorted Array:", selection_sort(data))
```

OUTPUT:



RESULT:

The Selection Sort algorithm was implemented successfully.

EXPERIMENT:3

AIM: To modify bubble sort to stop early when the list becomes sorted.

PROCEDURE:

- Compare and swap adjacent elements.

- Use a flag to detect if any swaps occur.
- If no swaps, stop early.
- Improves performance on nearly sorted lists.
- Print sorted list.

PROGRAM:

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr

# Test
nums = [64, 25, 12, 22, 11]
print("Sorted List:", bubble_sort(nums))
```

OUTPUT:

```
Sorted List: [11, 12, 22, 25, 64]


...Program finished with exit code 0
Press ENTER to exit console.
```

RESULT:

The optimized Bubble Sort algorithm stopped early when the list became sorted.

EXPERIMENT:4

AIM: To perform insertion sort and handle duplicate values correctly.

PROCEDURE:

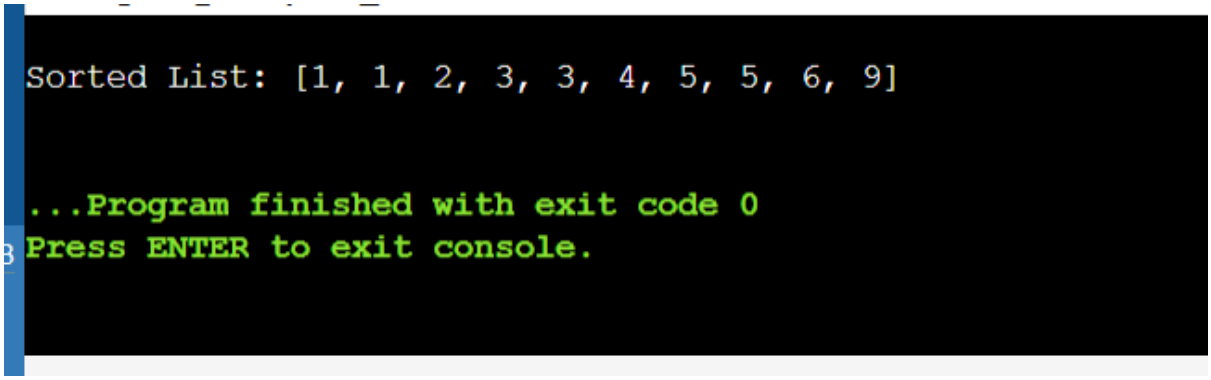
- Iterate through array elements.
- Insert each element into its correct position.
- Duplicates remain in their relative order (stable sort).
- Works well for small datasets.
- Display sorted array.

PROGRAM:

```
.py
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
    return arr

# Test
nums = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
print("Sorted List:", insertion_sort(nums))
```

OUTPUT:



RESULT:

Insertion Sort correctly handled duplicate elements and preserved their relative order.

EXPERIMENT:5

AIM: To find the kth missing positive number from a sorted array.

PROCEDURE:

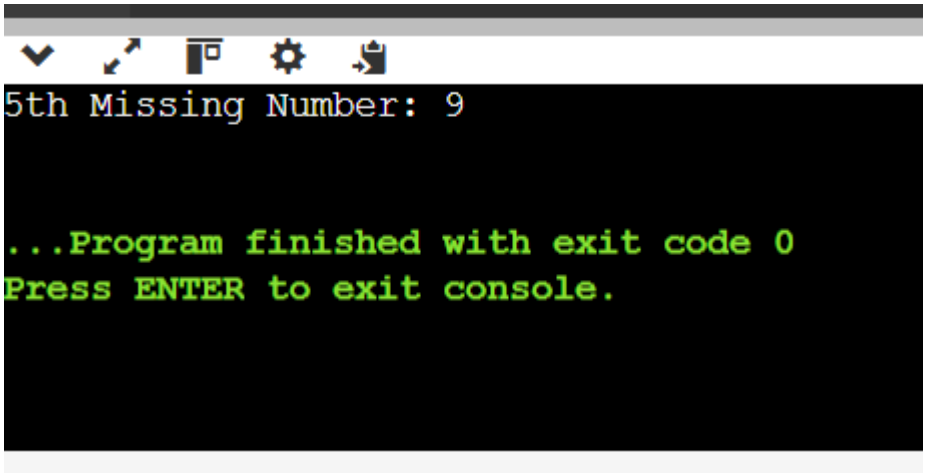
- Initialize counter for missing numbers.
- Loop through positive integers starting from 1.
- Skip numbers found in array.
- Stop when kth missing number is found.
- Display the result.

PROGRAM:

```python
def find_kth_missing(arr, k):
    num = 1
    missing = []
    i = 0
    while len(missing) < k:
        if i < len(arr) and arr[i] == num:
            i += 1
        else:
            missing.append(num)
        num += 1
    return missing[-1]

# Test
print("5th Missing Number:", find_kth_missing([2,3,4,7,11], 5))
```

OUTPUT:



RESULT:

The algorithm correctly found the kth missing positive integer from the given sorted array