DAY-1

1. Given an array of strings words, return the first palindromic string in the array. If there is no such string, return an empty string "". A string is palindromic if it reads the same forward and backward.
Example 1:
Input: words = ["abc","car","ada","racecar","cool"]
Output: "ada"
Explanation: The first string that is palindromic is "ada".
Note that "racecar" is also palindromic, but it is not the first.
Example 2:
Input: words = ["notapalindrome","racecar"]
Output: "racecar"
Explanation: The first and only string that is palindromic is "racecar".

CODE:

```
def firstPalindrome(words):

    for word in words:

        if word == word[::-1]:

            return word

    return ""

words1 = ["abc", "car", "ada", "racecar", "cool"]

print(firstPalindrome(words1))

words2 = ["notapalindrome", "racecar"]

print(firstPalindrome(words2))
```

OUTPUT:

ada
racecar

2. You are given two integer arrays nums1 and nums2 of sizes n and m, respectively. Calculate the following values: answer1 : the number of indices i such that nums1[i] exists in nums2. answer2 : the number of indices i such that nums2[i] exists in nums1 Return [answer1,answer2].
Example 1:
Input: nums1 = [2,3,2], nums2 = [1,2]
Output: [2,1]
Explanation:
Example 2:
Input: nums1 = [4,3,2,3,1], nums2 = [2,2,5,2,3,6]
Output: [3,4]
Explanation:
The elements at indices 1, 2, and 3 in nums1 exist in nums2 as well. So answer1 is 3.
The elements at indices 0, 1, 3, and 4 in nums2 exist in nums1. So answer2 is 4.

<u>CODE:</u>

```
def findCommonElements(nums1, nums2):

    answer1 = sum(1 for num in nums1 if num in nums2)

    answer2 = sum(1 for num in nums2 if num in nums1)

    return [answer1, answer2]

nums1 = [2, 3, 2]

nums2 = [1, 2]

print(findCommonElements(nums1, nums2))

nums1 = [4, 3, 2, 3, 1]

nums2 = [2, 2, 5, 2, 3, 6]

print(findCommonElements(nums1, nums2))
```

OUTPUT:

```
[2, 1]
[3, 4]
```

3. You are given a 0-indexed integer array nums. The distinct count of a subarray of nums is defined as: Let nums[i..j] be a subarray of nums consisting of all the indices from i to j such that 0 <= i <= j < nums.length. Then the number of distinct values in nums[i..j] is called the distinct count of nums[i..j]. Return the sum of the squares of distinct counts of all subarrays of nums. A subarray is a contiguous non-empty sequence of elements within an array.

Example 1:
Input: nums = [1,2,1]
Output: 15
Explanation: Six possible subarrays are:
[1]: 1 distinct value
[2]: 1 distinct value
[1]: 1 distinct value
[1,2]: 2 distinct values
[2,1]: 2 distinct values
[1,2,1]: 2 distinct values
The sum of the squares of the distinct counts in all subarrays is equal to 12 + 12 + 12 + 22 + 22 + 22 = 15.
Example 2:
Input: nums = [1,1]
Output: 3
Explanation: Three possible subarrays are:
[1]: 1 distinct value
[1]: 1 distinct value
[1,1]: 1 distinct value
The sum of the squares of the distinct counts in all subarrays is equal to 12 + 12 + 12 = 3.

CODE:

```
def sumOfSquaresOfDistinctCounts(nums):
    n = len(nums)
    total_sum = 0
    for i in range(n):
        distinct_set = set()
        for j in range(i, n):
            distinct_set.add(nums[j])
            total_sum += len(distinct_set) ** 2
    return total_sum

nums1 = [1, 2, 1]
print(sumOfSquaresOfDistinctCounts(nums1))

nums2 = [1, 1]
print(sumOfSquaresOfDistinctCounts(nums2))
```

OUTPUT:

```
15
3
```

4. Given a 0-indexed integer array nums of length n and an integer k, return the number of pairs (i, j) where 0 <= i < j < n, such that nums[i] == nums[j] and (i * j) is divisible by k.
Example 1:
Input: nums = [3,1,2,2,2,1,3], k = 2
Output: 4
Explanation:
There are 4 pairs that meet all the requirements:
- nums[0] == nums[6], and 0 * 6 == 0, which is divisible by 2.
- nums[2] == nums[3], and 2 * 3 == 6, which is divisible by 2.
- nums[2] == nums[4], and 2 * 4 == 8, which is divisible by 2.
- nums[3] == nums[4], and 3 * 4 == 12, which is divisible by 2.
Example 2:
Input: nums = [1,2,3,4], k = 1
Output: 0
Explanation: Since no value in nums is repeated, there are no pairs (i,j) that meet all the requirements.

CODE:

```
def countPairs(nums, k):
    n = len(nums)
    count = 0
    for i in range(n):
        for j in range(i + 1, n):
            if nums[i] == nums[j] and (i * j) % k == 0:
                count += 1
    return count


nums1 = [3, 1, 2, 2, 2, 1, 3]
k1 = 2
print(countPairs(nums1, k1))


nums2 = [1, 2, 3, 4]
k2 = 1
print(countPairs(nums2, k2))
```

OUTPUT:
4
0

5. Write a program FOR THE BELOW TEST CASES with least time complexity
Test Cases: -
1) Input: {1, 2, 3, 4, 5} Expected Output: 5
2) Input: {7, 7, 7, 7, 7} Expected Output: 7
3) Input: {-10, 2, 3, -4, 5} Expected Output: 5

CODE:

```
def findMax(nums):

    return max(nums)

nums1 = [1, 2, 3, 4, 5]

print(findMax(nums1))

nums2 = [7, 7, 7, 7, 7]

print(findMax(nums2))

nums3 = [-10, 2, 3, -4, 5]

print(findMax(nums3))
```

OUTPUT:

```
5
7
5
```

6. You have an algorithm that process a list of numbers. It firsts sorts the list using an efficient sorting algorithm and then finds the maximum element in sorted list. Write the code for the same.
Test Cases
1. Empty List
1. Input: []
2. Expected Output: None or an appropriate message indicating that the list is empty.
2. Single Element List
1. Input: [5]
2. Expected Output: 5
3. All Elements are the Same
1. Input: [3, 3, 3, 3, 3]
2. Expected Output: 3

CODE:

```
def processList(nums):

    if not nums:

        return "The list is empty"

    sorted_nums = sorted(nums)

    return sorted_nums[-1]

nums1 = []

print(processList(nums1))

nums2 = [5]

print(processList(nums2))

nums3 = [3, 3, 3, 3, 3]

print(processList(nums3))
```

OUTPUT:

```
The list is empty
5
3
```

7. Write a program that takes an input list of n numbers and creates a new list containing only the unique elements from the original list. What is the space complexity of the algorithm?
Test Cases
Some Duplicate Elements
- **Input: [3, 7, 3, 5, 2, 5, 9, 2]**
- **Expected Output: [3, 7, 5, 2, 9] (Order may vary based on the algorithm used)**

Negative and Positive Numbers
- **Input: [-1, 2, -1, 3, 2, -2]**
- **Expected Output: [-1, 2, 3, -2] (Order may vary)**

List with Large Numbers
- **Input: [1000000, 999999, 1000000]**
- **Expected Output: [1000000, 999999]**

CODE:

```
def uniqueElements(nums):

    return list(set(nums))

nums1 = [3, 7, 3, 5, 2, 5, 9, 2]

print(uniqueElements(nums1))

nums2 = [-1, 2, -1, 3, 2, -2]

print(uniqueElements(nums2))

nums3 = [1000000, 999999, 1000000]

print(uniqueElements(nums3))
```

OUTPUT:

```
[2, 3, 5, 7, 9]
[2, 3, -1, -2]
[1000000, 999999]
```

8. Sort an array of integers using the bubble sort technique. Analyze its time complexity using Big-O notation. Write the code

CODE:

```
def bubbleSort(arr):

    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]

    return arr

nums = [64, 34, 25, 12, 22, 11, 90]

sorted_nums = bubbleSort(nums)

print(sorted_nums)
```

Time Complexity:

- The time complexity of bubble sort is $O(n$^2) in the average and worst-case scenarios, where n is the number of elements in the array.
- This is because there are two nested loops: the outer loop runs nn times and the inner loop runs up to $n-i-1$ times for each iteration of the outer loop.
- The best case occurs when the array is already sorted, which results in a time complexity of $O(n)$ if an optimization is implemented to stop the algorithm when no swaps are made during a pass.

OUTPUT:

[11, 12, 22, 25, 34, 64, 90]

9. Checks if a given number x exists in a sorted array arr using binary search. Analyze its time complexity using Big-O notation.
Test Case:
Example X={ 3,4,6,-9,10,8,9,30} KEY=10
Output: Element 10 is found at position 5
Example X={ 3,4,6,-9,10,8,9,30} KEY=100
Output : Element 100 is not found

CODE:

```
def binarySearch(arr, key):
    arr.sort()  # Sort the array
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            left = mid + 1
        else:
            right = mid - 1
    return -1
arr = [3, 4, 6, -9, 10, 8, 9, 30]
key1 = 10
index1 = binarySearch(arr, key1)
if index1 != -1:
    print(f"Element {key1} is found at position {index1}")
else:
    print(f"Element {key1} is not found")
key2 = 100
index2 = binarySearch(arr, key2)
if index2 != -1:
    print(f"Element {key2} is found at position {index2}")
else:
    print(f"Element {key2} is not found")
```

Time Complexity:

- The time complexity for sorting the array is $O(n \log n)$ using a sorting algorithm like Timsort (which is used in Python).
- The time complexity for binary search itself is $O(\log n)$.
- Therefore, the overall time complexity of this implementation is $O(n \log n)$ due to the sorting step.

OUTPUT:

Element 10 is found at position 6
Element 100 is not found

10. Given an array of integers nums, sort the array in ascending order and return it. You must solve the problem without using any built-in functions in O(nlog(n)) time complexity and with the smallest space complexity possible.

CODE:

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)
def heapSort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr
nums = [3, 4, 6, -9, 10, 8, 9, 30]
sorted_nums = heapSort(nums)
print(sorted_nums)
```

Time Complexity:

- The time complexity of this implementation is O(nlogn)$O(n\log n)$ due to the heap operations.

Space Complexity:

- The space complexity is O(1)$O(1)$ because the sorting is done in-place.

OUTPUT:

[-9, 3, 4, 6, 8, 9, 10, 30]