Day 3 / 100 :

Topic - Array

1 Problem statement: [Distribute Candies](#) (Medium)

Alice has n candies, where the ith candy is of type candyType[i]. Alice noticed that she started to gain weight, so she visited a doctor.
The doctor advised Alice to only eat n / 2 of the candies she has (n is always even). Alice likes her candies very much, and she wants to eat the maximum number of different types of candies while still following the doctor's advice.

Given the integer array candyType of length n, return the maximum number of different types of candies she can eat if she only eats n / 2 of them.

Example 1:
Input: candyType = [1,1,2,2,3,3]
Output: 3
Explanation: Alice can only eat 6 / 2 = 3 candies. Since there are only 3 types, she can eat one of each type.
Example 2:
Input: candyType = [1,1,2,3]
Output: 2
Explanation: Alice can only eat 4 / 2 = 2 candies. Whether she eats types [1,2], [1,3], or [2,3], she still can only eat 2 different types.

Solutions :

Approach 1 - Brute Force

1. Initialize a variable, maxTypes, to keep track of the maximum number of different types of candies Alice can eat.
2. Sort the candyType array in ascending order.
3. Initialize a variable, eatLimit, to n/2 (where n is the length of the candyType array).
4. Initialize a variable, currentTypes, to 0, to keep track of the current number of different types of candies Alice can eat.
5. Iterate over the candyType array from the beginning.
    - If the current candy type is different from the previous one and Alice can still eat more candies (i.e., currentTypes < eatLimit), increment currentTypes by 1 and update maxTypes.
6. Return maxTypes.

```
class Solution {
public:
    int distributeCandies(vector<int>& candyType) {
        sort(candyType.begin(),candyType.end());
        int count=1;
        for(int i=0; i<candyType.size()-1;i++){
            if(candyType[i]!=candyType[i+1] &&
count<=(candyType.size()-1)/2){
                count++;
            }
        }
        return count;
    }
};
```

Time Complexity: The time complexity of this brute force algorithm is O(n log n) due to the sorting step, where n is the length of the candyType array.

Solutions :

Approach 2 - Using set

1. Initialize a variable 'n' to store the size of the candyType array.
2. Create an unordered_set 's' and initialize it with the elements of the candyType array using .
3. Check if the size of 's' is greater than or equal to n/2.
    - If true, return n/2, as Alice can eat one candy of each type.
4. If the condition in step 3 is false, return the size of 's', as Alice can eat all the available types.

Overall time complexity: O(n)
Overall space complexity: O(n)

```
class Solution {
public:
    int distributeCandies(vector<int>& candyType) {
        int n = candyType.size();
        unordered_set<int> s (candyType.begin(),candyType.end());
        if(s.size()>=n/2){
```

```
            return n/2;
        }
        return s.size();
    }
};
```

2️⃣ Problem statement: [Over Happy numbers](#) (easy)

Write an algorithm to determine if a number n is happy.
A happy number is a number defined by the following process:

Starting with any positive integer, replace the number by the sum of the squares of its digits.
Repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1.
Those numbers for which this process ends in 1 are happy.
Return true if n is a happy number, and false if not.

Example 1:
Input: n = 19
Output: true
Explanation:
12 + 92 = 82
82 + 22 = 68
62 + 82 = 100
12 + 02 + 02 = 1

Solutions :

Approach 1 - Hash set

1. The solution uses an unordered map to keep track of encountered numbers.
2. It iteratively calculates the sum of the squares of the digits until the number becomes 1 or a cycle is detected.
3. If a number is encountered for the first time, its count is incremented in the map. If a number is encountered again, indicating a cycle, the function returns false.
4. The function returns true if the number eventually becomes 1, indicating it is a happy number. Otherwise, it returns false.
5. This implementation follows the process definition of a happy number but uses a different approach compared to the first solution.

In the best case scenario, where **n is a happy** number and the loop terminates quickly, the time complexity is **O(1).**
In the worst case scenario, where n is not a happy number and the loop runs indefinitely, the time complexity is unbounded.

```
class Solution {
public:
    bool isHappy(int n) {
        unordered_map<int,int> tmp;

        while(n != 1)
        {
            if(tmp[n] == 0)
                tmp[n]++;
            else
                return false;

            int sum = 0;
            while(n != 0)
            {
                sum += pow(n % 10,2);
                n = n / 10;
            }

            n = sum;
        }

        return true;
    }
};
```

Solutions :

Approach 2 -

The algorithm uses two pointers, a slow pointer and a fast pointer, to iterate through the number sequence.

The **digitSquareSum** function calculates the sum of the squares of the digits of a given number n. It iterates through the digits of n by continuously dividing n by 10 and taking the remainder **(tmp)** as the current digit. It adds the square of tmp to the sum variable and updates n by dividing it by 10. This process continues until all digits have been processed, and the final sum is returned.

The **isHappy** function uses the Floyd's cycle detection algorithm to determine if a number is happy. It initializes two variables, slow and fast, with the input number n. The algorithm then enters a do-while loop where the slow pointer is moved one step at a time by calling **digitSquareSum** on itself, while the fast pointer is moved two steps at a time by calling **digitSquareSum** twice on itself.

The loop continues until slow and fast are equal, indicating a cycle has been detected, or until slow becomes 1, indicating the number is happy. If **slow is equal to 1**, the function **returns true**, indicating the number is happy. Otherwise, it **returns false**.

```
int digitSquareSum(int n) {
    int sum = 0, tmp;
    while (n) {
        tmp = n % 10;
        sum += tmp * tmp;
        n /= 10;
    }
    return sum;
}

bool isHappy(int n) {
    int slow, fast;
    slow = fast = n;
    do {
        slow = digitSquareSum(slow);
        fast = digitSquareSum(fast);
        fast = digitSquareSum(fast);
    } while(slow != fast);
    if (slow == 1) return 1;
    else return 0;
}
```

3️⃣ Problem statement: **Move Zeroes** (easy)

Given an integer array nums, move all 0's to the end of it while maintaining the relative order of the non-zero elements.

Note that you must do this in-place without making a copy of the array.

Example 1:

Input: nums = [0,1,0,3,12]
Output: [1,3,12,0,0]
Example 2:

Input: nums = [0]
Output: [0]

Solutions :

1. **Approach 1 - simple check approach**

   It initializes a variable k to keep track of the position where the next non-zero element should be placed. This variable starts at 0 initially.
2. The function uses a for loop to iterate through each element of the array nums.
3. Inside the loop, it checks if the current element (nums[i]) is non-zero.
   - If it is non-zero, the function swaps the current element with the element at position k and increments k by 1. This ensures that all non-zero elements are moved towards the beginning of the array while maintaining their relative order.
   - If it is zero, no swap is performed, and the loop proceeds to the next iteration.
4. After the loop completes, all non-zero elements have been moved towards the beginning of the array, and k points to the next available position after the non-zero elements.
5. Finally, all remaining positions from k to the end of the array are set to 0, effectively moving all zeros to the end of the array while preserving the relative order of non-zero elements.

```cpp
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
    int k =0;
    for(int i=0; i<nums.size() ; i++)
    {
        if(nums[i]!=0)
        swap(nums[i],nums[k++]);
    }
    }
};
```