Day 26 / 100 :

Topic - Stack, Queue

1 Problem statement: **Asteroid Collision** (Medium)

We are given an array of asteroids of integers representing asteroids in a row.
For each asteroid, the absolute value represents its size, and the sign represents its direction (positive meaning right, negative meaning left). Each asteroid moves at the same speed.

Find out the state of the asteroids after all collisions. If two asteroids meet, the smaller one will explode. If both are the same size, both will explode. Two asteroids moving in the same direction will never meet.

Example 1:
Input: asteroids = [5,10,-5]
Output: [5,10]
Explanation: The 10 and -5 collide resulting in 10. The 5 and 10 never collide.

Example 2:
Input: asteroids = [8,-8]
Output: []
Explanation: The 8 and -8 collide exploding each other.

Example 3:
Input: asteroids = [10,2,-5]
Output: [10]
Explanation: The 2 and -5 collide resulting in -5. The 10 and -5 collide resulting in 10.

Solutions :
Approach 1 - using stack

Intuition:

According to the question, positive values are moving to the right and negative values are moving to the left. We can apply the concept of relative velocity and make positive values as fixed and negative values now moving with double velocity in the negative direction. But, magnitude of velocity does not matter only the direction matters.

Idea:

Lets consider an example:-

[8, 9, 6, -7, -9, 12, 50, -34]

Start iterating from the beginning. Whenever we encounter a positive value, we don't have to do anything. Since they are fixed, they won't attack anyone. But the moment we sees a negative value, we are sure it is going to attack positive values.

Imagine [8, 9, 6] are happily sitting inside the array. The moment -7 enters, it will start knocking out positive values. This gives an idea we can use a stack to solve this problem.

Explanation:

Lets see how to use this idea to code.

Consider the same example [8, 9, 6, -7, -9, 12, 50, -34]
Initially i = 0.

- Whenever we encounter a positive value, we will simply push it to the stack.
- The moment we encounter a negative value, we know some or all or zero positive values will be knocked out of the stack. The negative value may itself be knocked out or it may enter the stack.
- We will keep on poping the elements from the stack while the asteroids[i] > s.top(). But remember, negative asteroids can never pop another negative asteroids, since they will never meet them. So, the final condition is while(!s.empty() and s.top() > 0 and s.top() < abs(ast[i])), we will pop the elements.
- We have to take care of the case when s.top() == asteroids[i]. In this case one positive element will pop out and negative asteroid won't enter the stack.
- If after knocking out elements stack becomes empty() or s.top() becomes negative, that means the current asteroids[i] will enter the stack.

```
class Solution {
public:
    vector<int> asteroidCollision(vector<int>& ast) {
        int n = ast.size();
        stack<int> s;
        for(int i = 0; i < n; i++) {
            if(ast[i] > 0 || s.empty()) {
                s.push(ast[i]);
            }
            else {
                while(!s.empty() and s.top() > 0 and s.top() < abs(ast[i]))
{
                    s.pop();
                }
                if(!s.empty() and s.top() == abs(ast[i])) {
                    s.pop();
                }
                else {
                    if(s.empty() || s.top() < 0) {
                        s.push(ast[i]);
                    }
                }
            }
        }
            // finally we are returning the elements which remains in the
stack.
            // we have to return them in reverse order.
        vector<int> res(s.size());
        for(int i = (int)s.size() - 1; i >= 0; i--) {
            res[i] = s.top();
            s.pop();
        }
        return res;
    }
};
```

2 Problem statement: Longest substring without repeating characters

(Medium)

Given a string s, find the length of the longest substring without repeating characters.

Example 1:

Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
Example 2:

Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.

Solutions :

Approach 1 - Sliding window

Intuition

We have to find the length of longest substring which does not contain any repeating characters

The first thing which should come in our mind is to traverse in the string and store the frequence of each character in a map type of "map<char ,int>" and try to maintain frequency of all the characters present in the map to 1 and storing the maximum length

Approach

1:- We will first iterate over the string and store the frequencies of all the character we have visited in a map of type "map<char ,int>"

2:- In each iteration we will check if the frequency of character at ith index is greater than one.....if yes it means that that character is repeated twice in our map so we will start erasing all the characters from the starting jth index untill we delete the same character which is repeated twice...... from the left

ex :-

```
 Input: s = "abcabcbb"

i=0 mp:{[a:1]}
|| length =1 , maxlength = 1

i=1 mp:{[a:1] , [b:1]}
|| length =2 , maxlength = 2

i=2 mp:{[a:1] , [b:1] , [c:1]}
|| length =3 , maxlength = 3

i=3 mp:{[a:2] , [b:1] , [c:1]}
|| length =4 , maxlength =3
```

'a' appeared 2 times , so we will start removing characters from the left untill the frequency of 'a' becomes 1 again.

while(mp[a]>1) , we will decrease the frequency of all the characters from left i.e. "mp[s[j++]]--" ......Variable j stores the starting index of our subarray , Initially it is 0 but

as we start deleting characters from left , this j will be get increment....that is why we did j++ there.
when j=0 , mp[s[j++]] will get decrease by one and the frequency of a will again be equal to 1

```
i=3 mp:{[a:1] , [b:1] , [c:1]}
```

|| length =3 , maxlength =3

3:- In each iteration we will take the maximum of (maxlength ,length) and store it in max length

This process will continue till we reach the end of the string and will return maxlength which is the length of longest substring with no repeating characters in it

```
    int lengthOfLongestSubstring(string s) {
        int length=0 , maxlength=0,j=0;
        map<char ,int> mp;
        for(int i=0 ;i<s.size(); i++){
            mp[s[i]]++;
            length++;
                while(mp[s[i]]>1){
                    mp[s[j++]]--;
                    length--;
                }
            maxlength = max(maxlength,length);
        }
        return maxlength;
    }
```

Day 27 / 100 :

Topic - Trees, DP

1 Problem statement: Number of longest Increasing subsequence
(Medium)

Given an integer array nums, return the number of longest increasing subsequences.
Notice that the sequence has to be strictly increasing.

Example 1:
Input: nums = [1,3,5,4,7]
Output: 2
Explanation: The two longest increasing subsequences are [1, 3, 4, 7] and [1, 3, 5, 7].

Example 2:
Input: nums = [2,2,2,2,2]
Output: 5
Explanation: The length of the longest increasing subsequence is 1, and there are 5 increasing subsequences of length 1, so output 5.

Solutions :
Approach 1 - DP

Intuition:
The given problem can be efficiently solved using a dynamic programming approach. We maintain two arrays, dp and count, to keep track of the length of the longest increasing subsequence and the count of such subsequences, respectively. The idea is to iterate through the input array, updating these arrays as we go along.

Approach:
1. Initialize the dp and count arrays with all elements set to 1, as each element is a valid subsequence of length 1.
2. For each element at index i in the input array, iterate through all elements before it (index j from 0 to i-1).
3. Compare the values of nums[i] and nums[j]:
    - If nums[i] is greater than nums[j], we have a potential increasing subsequence.
    - Check if dp[j] + 1 (the length of the LIS ending at index j plus the current element) is greater than dp[i] (the current length of the LIS ending at index

i). If so, update dp[i] to dp[j] + 1, and set count[i] to count[j] since we have found a new longer subsequence ending at i.
    - If dp[j] + 1 is equal to dp[i], it means we have found another subsequence with the same length as the one ending at i. In this case, we add count[j] to the existing count[i], as we have multiple ways to form subsequences with the same length.

4. Keep track of the maxLength of the LIS encountered during the process.
5. Finally, iterate through the dp array again, and for each index i, if dp[i] equals maxLength, add the corresponding count[i] to the result.

Complexity:
Time complexity: O(n^2)
Space complexity: O(n)

```
class Solution {
public:
    int findNumberOfLIS(vector<int>& nums) {
        // vector dp stores length of lonegst inc
subseq ending with nums[i]
        // vector cnt stores the number of subsequences
of length dp[i]

vector<int>dp(nums.size(),1),cnt(nums.size(),1);
        int mxLen=1,res=0;
        for(int i = 0; i < nums.size(); i++){

            for(int j = 0; j < i; j++){

                // if nums[j] < nums[i] we are sure
that we can append nums[i]
                if(nums[i] > nums[j]){
                    // append nums[i] only if length is
incresed
                    if(dp[j]+1 > dp[i]){
                    dp[i] = dp[j]+1;
                    cnt[i] = cnt[j];
                    }
                    // if appending nums[i] after
nums[j] gives length equal to dp[i] -> means there
exists more number of ways to design subseq of this
length so adding them to cnt[i]
                    else if(dp[j]+1 == dp[i])
                        cnt[i] += cnt[j];
                    }
            }
            // to store maxLen
            mxLen = max(mxLen, dp[i]);
        }
        //counting total number of ways to design
subseq of length mxLen
        for(int i=0;i<dp.size();i++){
            if(dp[i] == mxLen)
                res += cnt[i];
        }
        return res;
    } };
```

2 Problem statement: Palindrome Number (Medium)

Given an integer x, return true if x is a
palindrome and false otherwise.

Example 1:
Input: x = 121
Output: true
Explanation: 121 reads as 121 from left to right and from right to left.

Example 2:
Input: x = -121
Output: false
Explanation: From left to right, it reads -121. From right to left, it becomes 121-. Therefore, it is not a palindrome.

Example 3:
Input: x = 10
Output: false
Explanation: Reads 01 from right to left. Therefore, it is not a palindrome.

Solutions :

Approach 1 - Maths

If a reversed string representation of int x is the same as its original (forward) representation, it is a palindrome (return true). Otherwise, it is not, and we return false.

```
class Solution {
public:
    bool isPalindrome(int x) {
        string s = to_string(x); //convert int x to a string
        string s2 = s; //make a copy of the string representation of
int x to reverse
        reverse(s2.begin(), s2.end()); //use reverse to reverse s2
from its beginning to end
        return (s == s2); //*see note below
    }
};
```

Notes, tips, and tricks:

*Since (s == s2) is a conditional, and evaluates to a boolean, true or false, we can simplify the following

```
if(s == s2) {
      return true;
} else {
      return false;
}
to

return (s == s2);
```

which will either return true or false.

Note also that if you wanted the full if-else statement, you can choose to omit the brackets
(i.e.)

```
if(s == s2)
      return true;
else
      return false;
```

which is valid for compilation, and common in competitive programming, but bad practice in the real world. This is because you can only have one statement after the conditional if you omit brackets, and it's easy to miss that there aren't brackets and add another statement while working on a project/codebase. So, include brackets in actual programming work! Keep the quick and dirty stuff for competitive programming!