```
                ans += map[s[i]];
            }
        }
        return ans;
    }
};
```

Day 18 / 100 :

Topic - Array

1️⃣ **Problem statement:** [Maximum Length of a Concatenated String with Unique Characters](#) (Medium)

You are given an array of strings arr. A string s is formed by the concatenation of a subsequence of arr that has unique characters.

Return the maximum possible length of s.

A subsequence is an array that can be derived from another array by deleting some or no elements without changing the order of the remaining elements.

Example 1:

Input: arr = ["un","iq","ue"]
Output: 4
Explanation: All the valid concatenations are:
- ""
- "un"
- "iq"
- "ue"
- "uniq" ("un" + "iq")
- "ique" ("iq" + "ue")
Maximum length is 4.

Solutions :
Approach 1 - Brute Force

It first constructs bit masks for each string in the 'arr' vector to track unique characters.

It then iterates over all possible subsets of the 'arr' vector using a bitmask.

For each subset, it checks if the included strings have any overlapping characters based on the bit masks. If there are no overlaps, it calculates the length of the concatenated string for the current subset.

It keeps track of the maximum length found among all valid subsets and returns it as the result.

The time complexity of this approach is exponential, and the space complexity is linear.

Although this approach is straightforward, it may not be efficient for large inputs due to the exponential time complexity.

```
class Solution {
public:
    int maxLength(vector<string>& arr) {
        const size_t n = arr.size();

        vector<int> masks(n);
        for (int i = 0; i < n; i++) {
            for (char c : arr[i]) {
                if (masks[i] & (1 << (c - 'a'))) { /* has repeated letters
so cannot contribute */
                    arr[i] = "";
                    masks[i] = 0;
                    break;
                }
                masks[i] |= (1 << (c - 'a'));
            }
        }

        int ans = 0;
        for (int i = 0; i < (1 << n); i++) {

            int mask = 0; bool ok = true; int tot = 0;
            for (int j = 0; ok && j < n; j++) {
                if (i & (1 << j)) {
                    if (mask & masks[j]) ok = false;
                    mask |= masks[j];
                    tot += arr[j].size();
```

```
                }
            }
            if (ok) ans = max(ans, tot);
        }

        return ans;
    }
};
```

Approach 2 - Backtracking

To solve this problem, we can use a backtracking approach. We'll iterate through each string in the 'arr' array and try to include it in the concatenation or exclude it. We'll keep track of the maximum length found so far.

```
class Solution {
public:
    int maxi=0;
    bool checkRepeat(string s){
        int check=0;
        for(auto it:s){
            int i=it-'a';
            if((check & 1<<i))
                return true;
            check^=1<<i;
        }
        return false;
    }
    void maxPossibleLength(int ind,int size,string s,vector<string>& arr){
        if(checkRepeat(s))
            return;
        if(ind==size){
            int n=s.size();
            maxi=max(maxi,n);
            return;
        }
        maxPossibleLength(ind+1,size,s+arr[ind],arr);
        maxPossibleLength(ind+1,size,s,arr);
    }
```

```
    int maxLength(vector<string>& arr) {
        int size=arr.size();
        maxPossibleLength(0,size,"",arr);
        return maxi;
    }
};
```

We define a helper function called 'checkRepeat' that checks if a given string 's' contains any repeated characters. It uses a bitwise representation of a set to efficiently check for repetition.

The main function 'maxLength' takes the input array 'arr' and performs the following steps:
Initialize 'maxi' to 0, which will store the maximum length found so far.
Call the helper function 'maxPossibleLength' with initial parameters: index (ind) as 0, size as the length of 'arr', current concatenation (s) as an empty string, and the 'arr' array.
Return the value of 'maxi' as the result.
The helper function 'maxPossibleLength' takes the current index (ind), size, current concatenation (s), and the 'arr' array as parameters.
First, it checks if the current concatenation 's' contains repeated characters using the 'checkRepeat' function. If it does, it returns and backtracks.
If the index (ind) reaches the size, it means we have considered all strings in 'arr'. We calculate the length of 's' and update 'maxi' if the length is greater than 'maxi'.
Otherwise, we have two choices:
Include the string 'arr[ind]' in the concatenation by calling 'maxPossibleLength' recursively with the next index (ind+1) and the updated concatenation (s+arr[ind]).
Exclude the string 'arr[ind]' from the concatenation by calling 'maxPossibleLength' recursively with the next index (ind+1) and the same concatenation (s).
After backtracking, the function returns.
Finally, we call the 'maxLength' function with the initial input array 'arr'.

The time complexity of this solution depends on the number of possible concatenations, which can be exponential in the worst case. The space complexity is O(1), as we use recursion and maintain a 'maxi' variable.

2️⃣ **Problem statement:** Circular Array Loop (Medium)

You are playing a game involving a circular array of non-zero integers nums. Each nums[i] denotes the number of indices forward/backward you must move if you are located at index i:
If nums[i] is positive, move nums[i] steps forward, and
If nums[i] is negative, move nums[i] steps backward.
Since the array is circular, you may assume that moving forward from the last element puts you on the first element, and moving backwards from the first element puts you on the last element.
A cycle in the array consists of a sequence of indices seq of length k where:

Following the movement rules above results in the repeating index sequence
seq[0] -> seq[1] -> ... -> seq[k - 1] -> seq[0] -> ...
Every nums[seq[j]] is either all positive or all negative.
k > 1
Return true if there is a cycle in nums, or false otherwise.

Example 1:
Input: nums = [2,-1,1,2,2]
Output: true
Explanation: The graph shows how the indices are connected. White nodes are jumping forward, while red is jumping backward.
We can see the cycle 0 --> 2 --> 3 --> 0 --> ..., and all of its nodes are white (jumping in the same direction).

Example 2:
Input: nums = [-1,-2,-3,-4,-5,6]
Output: false
Explanation: The graph shows how the indices are connected. White nodes are jumping forward, while red is jumping backward.
The only cycle is of size 1, so we return false.

Solutions :
Approach 1 - Brute Force

In the brute-force approach, we iterate through each index in the nums array and check if we can find a cycle starting from that index. We maintain a visited set to keep track of the indices we have already visited.

Iterate through each index i from 0 to nums.size() - 1.
If i is already visited, skip it and move to the next index.
Create a new set cycle to store the indices in the current cycle.
Set the current index as start = i.
While the current index start is not in the cycle set:
Add the current index start to the cycle set.
Update the current index start by adding nums[start] to it.
If the updated index is out of bounds, adjust it to wrap around within the array by using the modulus operator %.
Check the size of the cycle set.
If the size is greater than 1 and all the elements in the cycle set have the same sign as nums[i], return true, since we found a valid cycle.
Otherwise, continue to the next index.
If no cycle is found after checking all indices, return false.

The time complexity of this brute-force approach is O(n^2) since we iterate through each index and potentially iterate through the entire cycle for each index.

```
class Solution {
public:
    bool circularArrayLoop(vector<int>& nums) {
        int n=nums.size();
        unordered_set<int> visited;

        for(int i=0;i<n;i++){
            if(visited.count(i))
                continue;

            unordered_set<int> cycle;
            int start=i;

            while(!cycle.count(start)){
                cycle.insert(start);
                start = (start+nums[start]+n)%n;
            }
            if(cycle.size()>1&& hasSameSign(nums[i],nums[start]))
            return true;

            visited.insert(cycle.begin(),cycle.end());
        }
        return false;
    }
    bool hasSameSign(int a, int b){
        return (a>=0 && b>=0) || (a<0 && b<0);
    }
};
```

Solutions :
Approach 2 - Floyd's Tortoise and Hare Algorithm

The intuition behind the approach is to use two pointers, slow and fast, to traverse the array. The slow pointer moves one step at a time, while the fast pointer moves two steps at a time. By comparing the signs of the elements at the prev and current indices and checking if the prev index points to itself (indicating a cycle of size 1), the function isNotCycle determines if a cycle is not possible.

Explanation:

Initialize slow and fast pointers to index 0.
Iterate from 1 to size (number of elements in nums) to handle all possible starting points for the cycle.
Inside the loop, store the current index in prev for both slow and fast pointers.
Update the slow pointer by moving it to the next index based on nums[slow] using the getNextStep function.
Check if the movement from prev to slow indicates that a cycle is not possible using the isNotCycle function. If true, move on to the next starting index and reset the slow and fast pointers to that index.
Set count to 2 and iterate count times to update the fast pointer similarly to the slow pointer. If the isNotCycle condition is met at any point, reset both pointers and continue to the next starting index.
If the slow and fast pointers meet at the same index, a cycle is found, so return true.
If the loop completes without finding any cycle, return false.

The getNextStep function is used to calculate the next index for a given pointer and its movement. It ensures that the index wraps around within the array by using the modulus operator and adjusts negative indices accordingly.

The optimized approach has a time complexity of O(n) as we traverse the array once with two pointers without revisiting any element.

```
class Solution {
public:
    bool isNotCycle(int prev, int current, vector<int> &nums)
    {
        int size=nums.size();

        if((nums[prev]>=0&&nums[current]<0) || (nums[prev]<0
&&nums[current]>=0)||(nums[prev]%size==0))
        {
            return true;
        }
        return false;
    }

    int getNextStep(int pointer, int nextIndex, int size)
    {
        int nextStep = (pointer+nextIndex)%size;

        if(nextStep<0)
```

```
        {
            nextStep+=size;
        }
        return nextStep;
    }

    bool circularArrayLoop(vector<int>& nums)
    {
       int slow=0;
       int fast=0;
       int size=nums.size();

       for(int i=1;i<=size;i++)
       {
           int prev=slow;
           slow=getNextStep(slow,nums[slow],size);

           if(isNotCycle(prev,slow,nums))
           {
               slow=i;
               fast=i;
               continue;
           }

           int count=2;
           bool nextIter=false;
           for(int x=0; x<count;x++)
           {
               prev=fast;
               fast=getNextStep(fast,nums[fast],size);
               if(isNotCycle(prev,fast,nums))
               {
                 slow=i;
                 fast=i;
                 nextIter=true;
                 break;
               }
           }
           if(nextIter)
           {
               continue;
           }
```

```
            if(slow==fast)
            {
                return true;
            }

        }
        return false;
    }
};
```