Day 2 / 100 :

Topic - Array

1 Problem statement: Container Containing Most Water (Medium)

You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return the maximum amount of water a container can store.

Notice that you may not slant the container.

Input: height = [1,8,6,2,5,4,8,3,7]
Output: 49
Explanation: The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

Solutions :

Approach 1 - Brute Force

1. Initialize a variable maxWater to keep track of the maximum water found, and a variable n to store the size of the input array height.
2. Iterate over each element of the array using the outer loop variable i from 0 to n-1.
    - Within the outer loop, start another loop with a variable j from i+1 to n.
    - Calculate the minimum height between height[i] and height[j], and multiply it by the distance between the two lines (j-i).
    - Update maxWater if the current water calculated is larger than the previously found maximum.
3. After the loops finish, return the maximum water stored in maxWater.

The brute force approach considers all possible pairs of lines and calculates the water contained between them. By updating maxWater whenever a larger amount of water is found, we eventually obtain the maximum possible water.

The time complexity of the brute force solution is O(n^2) because we have nested loops that iterate over the array. For every element, we compare it with every other element once.

```
class Solution {
public:
    int maxArea(vector<int>& height) {
       int maxWater =0;
       int n =height.size();

       for(int i=0;i<n-1;i++){
           for(int j=i+1; j<n;j++){
               int minHeight =min(height[i],height[j]);
               int currWater= minHeight*(j-i);
               maxWater = max(maxWater, currWater);
           }
       }
       return maxWater;
    }
};
```

Solutions :

Approach 2 - Two pointer approach

Intuition:
The intuition behind the optimized solution is that the maximum water can be obtained from the widest container (lines with maximum width) and the highest lines. By starting with the widest container and moving the pointers inward, we ensure that we explore all possible containers and find the maximum water without missing any potential improvements.

Explanation:

1. Initialize a variable maxWater to keep track of the maximum water found, and variables left and right to point to the leftmost and rightmost elements of the array height, respectively.
2. Use a while loop that continues as long as left is less than right.
    - Calculate the minimum height between height[left] and height[right], and multiply it by the width between the two lines (right-left).

- Update maxWater if the current water calculated is larger than the previously found maximum.
- Move the pointer corresponding to the smaller height inward. If height[left] is smaller, increment left by 1; otherwise, decrement right by 1.

3. After the loop finishes, return the maximum water stored in maxWater.

The optimized approach uses the two-pointer technique to find the maximum water more efficiently. By starting with the widest container (represented by the initial left and right pointers) and moving the pointer corresponding to the smaller height, we explore all possible containers and find the maximum water in linear time.

The time complexity of the optimized solution is **O(n)** because we iterate through the array only once, and the two pointers move towards each other at most n-1 times.

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        int maxWater =0;
        int n =height.size();
        int left=0;
        int right = n-1;

        while(left<right){
            int minHeight = min(height[left],height[right]);
            int currWater = minHeight*(right-left);
            maxWater=max(maxWater, currWater);

            if(height[left]<height[right])
                left++;
            else
                right--;
        }
        return maxWater;
    }
};
```

2 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.

1. Iterate through each index i from 0 to nums.size() - 1.
2. If i is already visited, skip it and move to the next index.
3. Create a new set cycle to store the indices in the current cycle.
4. Set the current index as start = i.
5. 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 %.
6. 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.
7. 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 1 - 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:

1. Initialize slow and fast pointers to index 0.
2. Iterate from 1 to size (number of elements in nums) to handle all possible starting points for the cycle.
3. Inside the loop, store the current index in prev for both slow and fast pointers.
4. Update the slow pointer by moving it to the next index based on nums[slow] using the getNextStep function.
5. 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.
6. 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.
7. If the slow and fast pointers meet at the same index, a cycle is found, so return true.
8. 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;
    }
};
```

3 Problem statement: Sunny Buildings (Medium)

The heights of certain Buildings which lie adjacent to each other are given. Light starts falling from left side of the buildings. If there is a building of certain Height, all the buildings to the right side of it having lesser heights cannot see the sun. The task is to find the total number of such buildings that receive light.

Input Format;
First line contains the Number of Testcases T.
Next line contains the totalnumber of buildings
Next line contains the height of n buildings.

Constraints:
1=T-100 1<=N==1001-H=100

Output Format:
Print total Number of buildings that receive light.

Sample Input:
2
8
53 9 6 10 11 8 13
6
341623
Sample Output
5
3

Solutions :

Approach 1

Inside the countBuildings function:

- We initialize the count variable to 1 because the first building is always visible.
- We also initialize the max_height variable with the height of the first building.
- Then, we iterate over the remaining buildings from index 1 to n-1.
- For each building at index i, we compare its height with max_height.

- If the height of the current building is greater than max_height, it means the building can see the sun. In that case, we increment the count variable and update max_height to the height of the current building.

In the main function:

- We read the number of test cases, T, from the input.
- We then enter a loop that iterates T times.
- For each test case, we read the total number of buildings, n, from the input.
- We create vector heights to store the heights of the buildings.
- We read n space-separated integers representing the heights of the buildings and store them in the heights vector.
- We call the countBuildings function, passing n and heights as arguments, and store the result in the result variable.
- Finally, we print the value of result (the count of buildings that receive light) and move to the next test case.

The code follows a straightforward approach, iterating over the heights and counting the number of buildings that can see the sun. The time complexity of the algorithm is **O(n)** since it iterates over the heights once, where n is the total number of buildings. The space complexity is **O(n)** as we use a vector to store the heights.

```
int countBuildings(int n, vector<int>& heights) {
    int count = 1;  // Initialize count with 1 for the first building
    int max_height = heights[0];

    for (int i = 1; i < n; i++) {
        if (heights[i] > max_height) {
            count++;
            max_height = heights[i];
        }
    }

    return count;
}


int main() {
    int T;
```

```
    cin >> T;

    while (T--) {
        int n;
        cin >> n;

        vector<int> heights(n);
        for (int i = 0; i < n; i++) {
            cin >> heights[i];
        }

        int result = countBuildings(n, heights);
        cout << result << endl;
    }

    return 0;
}
```