BLIND 75

Given an array of integers nums and an integer target, return *indices of the two numbers such that they add up to target*. You may assume that each input would have *exactly* one solution, and you may not use the *same* element twice.You can return the answer in any order.

Example 1:

Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        int n=nums.size();
        for(int i=0;i<n-1;i++){
         for(int j=i+1;j<n;j++){
            if(nums[i]+nums[j]==target){
                return {i,j};
            }
         }
        }
        return {};
    }

};
```

You are given an array prices where prices[i] is the price of a given stock on the i^{th} day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

Example 1:

Input: prices = [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int buy = prices[0];
        int profit = 0;
        for (int i = 1; i < prices.size(); i++) {
            if (prices[i] < buy) {
                buy = prices[i];
            } else if (prices[i] - buy > profit) {
                profit = prices[i] - buy;
            }
        }
        return profit;
    }

};
```

Given an integer array nums, return true if any value appears at least twice in the array, and return false if every element is distinct.

Example 1:

Input: nums = [1,2,3,1]
Output: true

```
class Solution {
public:
    bool containsDuplicate(vector<int>& nums) {
        sort(nums.begin(),nums.end());
        for(int i=1;i<nums.size();i++)
        {
            if(nums[i]==nums[i-1]){
                return true;
            }
        }
        return false;
    }
};
```

Given an integer array nums, return *an array* answer *such that* answer[i] *is equal to the product of all the elements of* nums *except* nums[i].

The product of any prefix or suffix of nums is guaranteed to fit in a 32-bit integer. You must write an algorithm that runs in O(n) time and without using the division operation.

Example 1:

Input: nums = [1,2,3,4]
Output: [24,12,8,6]

```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        int n=nums.size();
        vector<int> left(n);
        vector<int> right(n);
        vector<int> prod(n);
        left[0]=1;
        right[n-1]=1;
        for(int i=1;i<n;i++){
            left[i]=left[i-1]*nums[i-1];
        }
        for(int i=n-2;i>=0;i--){
            right[i]=right[i+1]*nums[i+1];
        }
        for(int i=0;i<n;i++){
            prod[i]=left[i]*right[i];
        }
        return prod;
    }
};
```

Given an integer array nums, find the Subarray with the largest sum, and return *its sum*.

Example 1:

Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: The subarray [4,-1,2,1] has the largest sum 6.

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int n=nums.size();
        int max_sum=nums[0];
        int current_sum=nums[0];

        for(int i=1;i<nums.size();i++)
        {
            current_sum=max(nums[i],current_sum+nums[i]);
            max_sum=max(max_sum,current_sum);
        }
        return max_sum;
    }
};
```

Suppose an array of length n sorted in ascending order is rotated between 1 and n times. For example, the array nums = [0,1,2,4,5,6,7] might become:

- **[4,5,6,7,0,1,2] if it was rotated 4 times.**
- **[0,1,2,4,5,6,7] if it was rotated 7 times.**

Notice that rotating an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums of unique elements, return *the minimum element of this array*.

You must write an algorithm that runs in O(log n) time.

Example 1:

Input: nums = [3,4,5,1,2]
Output: 1
Explanation: The original array was [1,2,3,4,5] rotated 3 times.

```
class Solution {
public:
    int findMin(vector<int>& nums) {
        int min=nums[0];
        for(int i=0;i<nums.size();i++){
            if(min>nums[i]){
                min=nums[i];
            }
        }
        return min;
    }
};
```

There is an integer array nums sorted in ascending order (with distinct values).

Prior to being passed to your function, nums is possibly rotated at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (0-indexed). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

Given the array nums after the possible rotation and an integer target, return *the index of* target *if it is in* nums*, or* -1 *if it is not in* nums.

You must write an algorithm with O(log n) runtime complexity.

Example 1:

Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int low = 0, high = nums.size() - 1;

        while (low <= high) {
            int mid = low + (high - low) / 2;

            if (nums[mid] == target) {
                return mid;
            }

            if (nums[low] <= nums[mid]) {
                if (nums[low] <= target && target < nums[mid]) {
                    high = mid - 1;
                } else {
                    low = mid + 1;
                }
            } else {
                if (nums[mid] < target && target <= nums[high]) {
                    low = mid + 1;
                } else {
                    high = mid - 1;
                }
            }
        }

        return -1;
    }
};
```

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

Notice that the solution set must not contain duplicate triplets.

Example 1:

Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]
Explanation:
nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.
nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.
nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.
The distinct triplets are [-1,0,1] and [-1,-1,2].

Notice that the order of the output and the order of the triplets does not matter.

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> res;
        sort(nums.begin(), nums.end());

        for (int i = 0; i < nums.size(); i++) {
            if (i > 0 && nums[i] == nums[i-1]) {
                continue;
            }

            int j = i + 1;
            int k = nums.size() - 1;

            while (j < k) {
                int total = nums[i] + nums[j] + nums[k];

                if (total > 0) {
                    k--;
                } else if (total < 0) {
                    j++;
                } else {
                    res.push_back({nums[i], nums[j], nums[k]});
                    j++;

                    while (nums[j] == nums[j-1] && j < k) {
                        j++;
                    }
                }
            }
        }
        return res;
    }
};
```

You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the i_{th} 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

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

        while (left < right) {
            maxArea = max(maxArea, (right - left) * min(height[left], height[right]));

            if (height[left] < height[right]) {
                left++;
            } else {
                right--;
            }
        }

        return maxArea;
    }
};
```