***Note: All the solutions are in Javascript Language***

**Question 1** Given an integer array nums of 2n integers, group these integers into n pairs (a1, b1), (a2, b2),..., (an, bn) such that the sum of min(ai, bi) for all i is maximized. Return the maximized sum.

**Example 1:** Input: nums = [1,4,3,2] Output: 4

**Explanation:** All possible pairings (ignoring the ordering of elements) are:

1. (1, 4), (2, 3) -> min(1, 4) + min(2, 3) = 1 + 2 = 3
2. (1, 3), (2, 4) -> min(1, 3) + min(2, 4) = 1 + 2 = 3
3. (1, 2), (3, 4) -> min(1, 2) + min(3, 4) = 1 + 3 = 4 So the maximum possible sum is 4

**Answer:**

Let's say in each pair (ai, bi) we have bi >= ai and the sum of all min be Sm where

Sm = min(a1,b1) + min(a2,b2) and so on. The maximum of all would be the sum of all the initial numbers of the pair like a1+a2+a3……+an as all the bi >=ai. Therefore, quickly finding the sum of all numbers in even places will give us the sum of all the min pairing values.

**Code**

| var arrayPairSum = function(nums) {  nums.sort((a, b) => a - b);  let sum = 0;   for (let i = 0; i < nums.length; i++) {  if (i % 2 === 0) sum += nums[i]  }  return sum }; |
| --- |

The time complexity would be O(n log n) as it is using the sorting method and the space complexity would be O(1) as the variable ‘sum’ takes a constant amount of space.

**Question 2**

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.

**Answer:** In this problem above, we have to find the unique number of candies. We can do this with the help of ‘set’ where we can find the unique values and the size of the set will provide the different candy types we have. The only condition is that Alice can eat at most half of the length of the array.

| var distributeCandies = function(candies) {  return Math.min(new Set(candies).size, candies.length / 2); }; |
| --- |

The time complexity of the given solution is O(n) as creating a set would be in ‘n’ length of array. The space complexity would be O(n) as if all the elements in the candies are unique, the set will store all ‘n’ elements.

**Question 3**

We define a harmonious array as an array where the difference between its maximum value and its minimum value is exactly 1.

Given an integer array nums, return the length of its longest harmonious subsequence among all its possible subsequences.

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

**Example 1:** Input: nums = [1,3,2,2,5,2,3,7] Output: 5

**Explanation:** The longest harmonious subsequence is [3,2,2,2,3].

**Answer:**

The approach would be to create a hash map using 2 maps. One map is to track the frequency of each number and the second would track all the numbers already processed. Initially, we will iterate over the array nums and check for the frequency of each element. Next, we will increment the number by 1 and check for that number’s frequency in the array. Later, the while loop will iterate over the nums that have already been processed and check if it exists as a key in the other object. This is how the iteration will continue to the next loop. It will increment the current number by 1 and calculate the sum of the frequencies of both numbers. The time complexity of this code is O(n) as it single passes over the array ’nums’ of length ‘n’. Space complexity would be O(n) as ‘freq’ and ‘already processed’ objects can potentially store all the unique numbers in the ‘nums’ array.

| var findLHS = function(nums) {  let freq = {};  let alreadyProcessed = {};   for(let i=0 ;i<nums.length; i++) {  if(freq[nums[i]]) {  freq[nums[i]]++;  } else {  freq[nums[i]] = 1;  }  }   let maxLength = 0;  let i=0;  while(i<nums.length) {  if(alreadyProcessed[nums[i]]) {  i++  continue  }   if(freq[nums[i] + 1]) {  let sum = freq[nums[i] + 1] + freq[nums[i]];  maxLength = Math.max(maxLength, sum)  }   alreadyProcessed[nums[i]]=1;  i++;  }   return maxLength; }; |
| --- |

**Question 4**

You have a long flowerbed in which some of the plots are planted, and some are not. However, flowers cannot be planted in adjacent plots. Given an integer array flowerbed containing 0's and 1's, where 0 means empty and 1 means not empty, and an integer n, return true if n new flowers can be planted in the flowerbed without violating the no-adjacent-flowers rule and false otherwise.

**Example 1:**

Input: flowerbed = [1,0,0,0,1], n = 1

Output: true

**Answer:**

The approach behind this problem is to initially iterate over the flowerbed array and check the occupied places. If it is occupied it will move to the next, if not then we will check the neighboring element to determine if there is a possible place to plant a flower. But before planting, we have to check whether the previous and the next elements are occupied or not. If both conditions are true we can say the current position is suitable to plant and decrement by the n element by 1. Finally, we will check if n is less than or equal to 0 to make sure we have planted all the flowers in the place. If it is not equal to 0, we will return false else true.

| var canPlaceFlowers = function (flowerbed, n) {  for (let i = 0; i < flowerbed.length; i++) {  if (flowerbed[i] === 1) continue;   if (  (i === 0 || flowerbed[i - 1] === 0) &&  (i === flowerbed.length - 1 || flowerbed[i + 1] === 0)  ) {  flowerbed[i] = 1;  n--;  }  }   return n <= 0; }; |
| --- |

The time complexity would be O(n) as ‘n’ is the length of the flowerbed array. The space complexity would be O(1) as we didn’t use any additional data structure as the input.

**Question 5**

Given an integer array nums, find three numbers whose product is maximum and return the maximum product.

**Example 1:**

Input: nums = [1,2,3]

Output: 6

**Answer:** The approach would be straightforward in this case where we first sort the array in descending order and then return the product of first 3 numbers.

| **var maximumProduct = function(nums) {  nums = nums.sort((a,b) => b-a)  return Math.max(nums[0]\*nums[1]\*nums[2]) };** |
| --- |

The time complexity would be O(n log n) as we are using the sort function, The space complexity would be O(1) as this doesn’t require any additional space.

**Question 6**

Given an array of integers nums which is sorted in ascending order, and an integer target, write a function to search target in nums. If a target exists, then return its index. Otherwise, return -1.

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

Input: nums = [-1,0,3,5,9,12], target = 9

Output: 4

**Explanation:** 9 exists in nums and its index is 4

**Answer:** Binary search would be an efficient method that could be used here to find items from a sorted list of items. It works by repeatedly dividing the search interval by half and comparing the mid-item with the targeted element. If the value of the search is less than the item in the middle we will narrow the interval to the lower half otherwise we will look at the upper half.

The time complexity would be O(log n) since it reduces the search space by half and the max number of iterations to find the target is log base 2 of n where n is the size of an array. The space complexity would be O(1).

| function binarySearch(nums, target) {  let left = 0;  let right = nums.length - 1;    while (left <= right) {  const mid = Math.floor((left + right) / 2);    if (nums[mid] === target) {  return mid;  } else if (nums[mid] < target) {  left = mid + 1;  } else {  right = mid - 1;  }  }    return -1; } |
| --- |

**Question 7**

An array is monotonic if it is either monotone increasing or monotone decreasing.

An array nums is monotone increasing if for all i <= j, nums[i] <= nums[j]. An array nums is

monotone decreasing if for all i <= j, nums[i] >= nums[j].

Given an integer array nums, return true if the given array is monotonic, or false otherwise.

**Example 1:**

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

Output: true

**Answer**

The best approach is to run 2 loops in which one finds for increasing order and the other search for decreasing order. If either of them is true then it is monotonic.

| var isMonotonic = function(A) {    let increasing = true  let decreasing = true    for(let i=0; i<A.length -1; i++){  if(A[i] > A[i+1]){  increasing = false  }  if(A[i] < A[i+1]){  decreasing = false  }    }    return increasing || decreasing }; |
| --- |

The above code has the time complexity of O(n) as it performs a constant number of comparison for array elements of length ‘n’. The space complexity would be O(1) as no extra space is being utilized.

**Question 8**

You are given an integer array nums and an integer k. In one operation, you can choose any index i where 0 <= i < nums.length and change nums[i] to nums[i] + x where x is an integer from the range [-k, k]. You can apply this operation at most once for each index i.

The score of nums is the difference between the maximum and minimum elements in nums.

Return the minimum score of nums after applying the mentioned operation at most once for each index in it.

**Example 1:**

Input: nums = [1], k = 0

Output: 0

**Explanation:** The score is max(nums) - min(nums) = 1 - 1 = 0.

**Answer:** In the above problem, we basically have to follow these steps one by one.

* If num’s size is 1 return 0
* Find max and min values in nums
* Minimize max value (by subtracting K)
* Maximize min value (by adding K)
* Find the difference between max and min
* If the difference is negative, return 0, otherwise return the difference.

| var smallestRangeI = function(nums, k) {  const min = Math.min(...nums);  const max = Math.max(...nums);    if (max - min <= 2\*k) return 0;  return max - min - 2\*k; }; |
| --- |

The time complexity would be O(n) as it performs a single pass over the nums to find min and max. The space complexity would be O(1) as it uses a constant amount of additional space for ‘min’ and ‘max’ variables.