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

Ans: To maximize the sum of the minimum elements in each pair, we need to pair the smallest numbers with each other. Sorting the array in ascending order will allow us to easily pair adjacent elements.

Here's the algorithm to solve this problem:

1. Sort the array `nums` in ascending order.

2. Initialize a variable `max\_sum` to 0, which will store the maximum sum.

3. Iterate over the sorted array by jumping two steps in each iteration.

- For each pair `(a, b)`, add the smaller element `a` to `max\_sum`.

4. Return `max\_sum`.

Here's the implementation in Python:

```python

def arrayPairSum(nums):

nums.sort()

max\_sum = 0

for i in range(0, len(nums), 2):

max\_sum += nums[i]

return max\_sum

```

Let's test the function with the example you provided:

```python

nums = [1, 4, 3, 2]

result = arrayPairSum(nums)

print(result)

```

Output:

```

4

```

The maximum possible sum of the minimum elements in each pair is 4, as explained in the example.

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.

Ans: To solve this problem, we need to determine the maximum number of different types of candies Alice can eat while following the doctor's advice of eating only n / 2 candies.

Here's the algorithm to solve this problem:

1. Count the number of unique candy types in the array `candyType` using a set data structure.

2. Calculate the maximum number of candy types Alice can eat by taking the minimum of the number of unique candy types and n / 2.

- If the number of unique candy types is less than or equal to n / 2, Alice can eat all the unique types.

- If the number of unique candy types is greater than n / 2, Alice can only eat n / 2 different types.

3. Return the maximum number of candy types Alice can eat.

Here's the implementation in Python:

```python

def distributeCandies(candyType):

unique\_types = len(set(candyType))

max\_types = min(unique\_types, len(candyType) // 2)

return max\_types

```

Let's test the function with the example you provided:

```python

candyType = [1, 1, 2, 2, 3, 3]

result = distributeCandies(candyType)

print(result)

```

Output:

```

3

```

Alice can eat a maximum of 3 different types of candies out of the 6 available candies, as explained in the example.

1. 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].

Ans: To find the length of the longest harmonious subsequence in an array, we can iterate through the array and count the frequencies of each number. Then, for each number, we check if there exists another number in the array such that the difference between the two numbers is exactly 1. If such a number exists, we calculate the length of the harmonious subsequence starting with the current number by adding the frequencies of both numbers. We keep track of the maximum length found so far and return it as the result.

Here's the algorithm to solve this problem:

1. Initialize an empty dictionary `freq` to store the frequencies of each number.

2. Iterate through the array `nums` and count the frequency of each number by updating the `freq` dictionary.

3. Initialize a variable `max\_length` to 0, which will store the maximum length of the harmonious subsequence found so far.

4. Iterate through the keys of the `freq` dictionary.

- For each key `num`, check if there exists another key `num + 1` in the `freq` dictionary.

- If `num + 1` exists, calculate the length of the harmonious subsequence starting with `num` by adding the frequencies of both `num` and `num + 1`.

- If the length is greater than `max\_length`, update `max\_length` with the new length.

5. Return `max\_length`.

Here's the implementation in Python:

```python

def findLHS(nums):

freq = {}

for num in nums:

freq[num] = freq.get(num, 0) + 1

max\_length = 0

for num in freq:

if num + 1 in freq:

length = freq[num] + freq[num + 1]

max\_length = max(max\_length, length)

return max\_length

```

Let's test the function with the example you provided:

```python

nums = [1, 3, 2, 2, 5, 2, 3, 7]

result = findLHS(nums)

print(result)

```

Output:

```

5

```

The longest harmonious subsequence in the given array is `[3, 2, 2, 2, 3]` with a length of 5, as explained in the example.

1. 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

Ans: To determine if n new flowers can be planted in the flowerbed without violating the no-adjacent-flowers rule, we can iterate through the flowerbed and check if there are enough empty plots available to accommodate the new flowers.

Here's the algorithm to solve this problem:

1. Initialize a variable `count` to 0, which will store the number of new flowers that can be planted.

2. Iterate through the flowerbed.

- If the current plot is empty (0), check if the previous and next plots are also empty (or out of bounds).

- If both the previous and next plots are empty (or out of bounds), increment `count` by 1, indicating a new flower can be planted in the current plot.

- Update the current plot to 1 to mark it as planted.

3. Check if `count` is greater than or equal to n.

- If `count` is greater than or equal to n, return True.

- Otherwise, return False.

Here's the implementation in Python:

```python

def canPlaceFlowers(flowerbed, n):

count = 0

size = len(flowerbed)

for i in range(size):

if flowerbed[i] == 0:

if (i == 0 or flowerbed[i-1] == 0) and (i == size-1 or flowerbed[i+1] == 0):

count += 1

flowerbed[i] = 1

return count >= n

```

Let's test the function with the example you provided:

```python

flowerbed = [1, 0, 0, 0, 1]

n = 1

result = canPlaceFlowers(flowerbed, n)

print(result)

```

Output:

```

True

```

In the given flowerbed, there is one empty plot available where a new flower can be planted without violating the no-adjacent-flowers rule. Therefore, the function returns True.

1. 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 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

Ans: To search for a target in a sorted array with a runtime complexity of O(log n), we can use the binary search algorithm. Binary search works by repeatedly dividing the search space in half until the target is found or the search space is exhausted.

Here's the algorithm to solve this problem:

1. Initialize two pointers, `left` and `right`, to the start and end of the array `nums` respectively.

2. While `left` is less than or equal to `right`, do the following:

- Calculate the middle index as `mid` using the formula `mid = (left + right) // 2`.

- Check if the target is equal to the value at index `mid` in `nums`.

- If they are equal, return `mid`.

- If the target is less than the value at index `mid`, update `right` to `mid - 1` to search the left half of the array.

- If the target is greater than the value at index `mid`, update `left` to `mid + 1` to search the right half of the array.

3. If the target is not found after the while loop, return -1.

Here's the implementation in Python:

```python

def search(nums, target):

left, right = 0, len(nums) - 1

while left <= right:

mid = (left + right) // 2

if nums[mid] == target:

return mid

elif nums[mid] < target:

left = mid + 1

else:

right = mid - 1

return -1

```

Let's test the function with the example you provided:

```python

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

target = 9

result = search(nums, target)

print(result)

```

Output:

```

4

```

The target value 9 exists in the given array `nums` and its index is 4, as explained in the example.

1. 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

Ans: To determine if an array is monotonic, we can iterate through the array and check if it is either strictly increasing or strictly decreasing. If both conditions are not met, then the array is not monotonic.

Here's the algorithm to solve this problem:

1. Initialize two variables, `is\_increasing` and `is\_decreasing`, to True.

2. Iterate through the array `nums` starting from index 1.

- If the current element `nums[i]` is less than the previous element `nums[i-1]`, set `is\_increasing` to False.

- If the current element `nums[i]` is greater than the previous element `nums[i-1]`, set `is\_decreasing` to False.

3. Return `is\_increasing` or `is\_decreasing`.

Here's the implementation in Python:

```python

def isMonotonic(nums):

is\_increasing = True

is\_decreasing = True

for i in range(1, len(nums)):

if nums[i] < nums[i-1]:

is\_increasing = False

if nums[i] > nums[i-1]:

is\_decreasing = False

return is\_increasing or is\_decreasing

```

Let's test the function with the example you provided:

```python

nums = [1, 2, 2, 3]

result = isMonotonic(nums)

print(result)

```

Output:

```

True

```

The given array `[1, 2, 2, 3]` is monotonic because it is strictly increasing. Therefore, the function returns True.

1. 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.

Ans: To find the minimum score of an integer array `nums` after applying the mentioned operation at most once for each index, we can make two observations:

1. The minimum score occurs when the minimum and maximum elements in `nums` are as close to each other as possible.

2. By applying the operation, we can increase the minimum value and decrease the maximum value. We want to minimize the difference between the modified minimum and maximum values.

Here's the algorithm to solve this problem:

1. Find the minimum and maximum values in `nums`.

2. Calculate the initial score as `max(nums) - min(nums)`.

3. If `k` is greater than or equal to the difference between the minimum and maximum values, return 0 as the minimum score because we can modify all elements in `nums` to have the same value.

4. Otherwise, calculate the potential minimum and maximum values after applying the operation. The potential minimum value will be `min(nums) + k`, and the potential maximum value will be `max(nums) - k`.

5. Calculate the potential score as the difference between the potential maximum and minimum values.

6. Return the minimum of the initial score and the potential score.

Here's the implementation in Python:

```python

def minimumScore(nums, k):

minimum = min(nums)

maximum = max(nums)

initial\_score = maximum - minimum

if k >= maximum - minimum:

return 0

potential\_min = minimum + k

potential\_max = maximum - k

potential\_score = potential\_max - potential\_min

return min(initial\_score, potential\_score)

```

Let's test the function with the example you provided:

```python

nums = [1]

k = 0

result = minimumScore(nums, k)

print(result)

```

Output:

```

0

```

The minimum score of the array `[1]` after applying the operation at most once for each index is 0, as explained in the example.