Given an integer array nums sorted in **non-decreasing** order, return an array of ***the squares of each number*** sorted in non-decreasing order.

**Example 1:**

**Input:** nums = [-4,-1,0,3,10]

**Output:** [0,1,9,16,100]

**Explanation:** After squaring, the array becomes [16,1,0,9,100].

After sorting, it becomes [0,1,9,16,100].

Ans: To solve this problem, you can follow the following steps:

1. Initialize an empty result array to store the squares.

2. Iterate through each number in the input array, `nums`.

3. Square each number and append it to the result array.

4. Sort the result array in non-decreasing order.

5. Return the sorted result array.

Here's the implementation in Python:

```python

def sortedSquares(nums):

result = [] # Initialize an empty result array

for num in nums:

result.append(num \* num) # Square each number and append it to the result array

result.sort() # Sort the result array in non-decreasing order

return result

```

Let's test it with the given example:

```python

nums = [-4, -1, 0, 3, 10]

print(sortedSquares(nums))

```

Output:

```

[0, 1, 9, 16, 100]

```

The function returns the squared numbers sorted in non-decreasing order, as expected.

Q. Given two **0-indexed** integer arrays nums1 and nums2, return *a list* answer *of size* 2 *where:*

* answer[0] *is a list of all* ***distinct*** *integers in* nums1 *which are* ***not*** *present in* nums2\*.\*
* answer[1] *is a list of all* ***distinct*** *integers in* nums2 *which are* ***not*** *present in* nums1.

**Note** that the integers in the lists may be returned in **any** order.

**Example 1:**

**Input:** nums1 = [1,2,3], nums2 = [2,4,6]

**Output:** [[1,3],[4,6]]

**Explanation:**

For nums1, nums1[1] = 2 is present at index 0 of nums2, whereas nums1[0] = 1 and nums1[2] = 3 are not present in nums2. Therefore, answer[0] = [1,3].

For nums2, nums2[0] = 2 is present at index 1 of nums1, whereas nums2[1] = 4 and nums2[2] = 6 are not present in nums2. Therefore, answer[1] = [4,6].

Ans: To solve this problem, you can use a set to efficiently check for distinct integers that are not present in the other array. Here's the step-by-step approach:

1. Convert both `nums1` and `nums2` into sets to eliminate duplicate values.

2. Initialize two empty lists, `not\_in\_nums2` and `not\_in\_nums1`, to store distinct integers not present in `nums2` and `nums1`, respectively.

3. Iterate through each number in `nums1` and check if it is in the set representation of `nums2`. If it is not, add it to the `not\_in\_nums2` list.

4. Similarly, iterate through each number in `nums2` and check if it is in the set representation of `nums1`. If it is not, add it to the `not\_in\_nums1` list.

5. Return a list containing `not\_in\_nums1` and `not\_in\_nums2`.

Here's the implementation in Python:

```python

def findDisappearedNumbers(nums1, nums2):

nums1\_set = set(nums1)

nums2\_set = set(nums2)

not\_in\_nums2 = [num for num in nums1 if num not in nums2\_set]

not\_in\_nums1 = [num for num in nums2 if num not in nums1\_set]

return [not\_in\_nums1, not\_in\_nums2]

```

Let's test it with the given example:

```python

nums1 = [1, 2, 3]

nums2 = [2, 4, 6]

print(findDisappearedNumbers(nums1, nums2))

```

Output:

```

[[4, 6], [1, 3]]

```

The function returns a list where the first element is `[4, 6]`, which represents the distinct integers in `nums2` that are not present in `nums1`. The second element is `[1, 3]`, which represents the distinct integers in `nums1` that are not present in `nums2`.

Q. Given two integer arrays arr1 and arr2, and the integer d, return the distance value between the two arrays.

The distance value is defined as the number of elements arr1[i] such that there is not any element arr2[j] where |arr1[i]-arr2[j]| <= d.

**Example 1:**

**Input:** arr1 = [4,5,8], arr2 = [10,9,1,8], d = 2

**Output:** 2

**Explanation:**

For arr1[0]=4 we have:

|4-10|=6 > d=2

|4-9|=5 > d=2

|4-1|=3 > d=2

|4-8|=4 > d=2

For arr1[1]=5 we have:

|5-10|=5 > d=2

|5-9|=4 > d=2

|5-1|=4 > d=2

|5-8|=3 > d=2

For arr1[2]=8 we have:

**|8-10|=2 <= d=2**

**|8-9|=1 <= d=2**

|8-1|=7 > d=2

**|8-8|=0 <= d=2**

Ans: To solve this problem, you can use nested loops to iterate through each element of `arr1` and `arr2` and check the condition |arr1[i] - arr2[j]| <= d. If the condition is not satisfied for any element in `arr2`, increment a counter to keep track of the number of valid elements in `arr1`. Here's the step-by-step approach:

1. Initialize a counter `distance` to 0 to keep track of the number of valid elements in `arr1`.

2. Iterate through each element `num1` in `arr1`.

3. For each element `num1`, iterate through each element `num2` in `arr2`.

4. Check the condition |num1 - num2| <= d. If the condition is satisfied for any element in `arr2`, break the inner loop.

5. If the inner loop completes without breaking, increment the `distance` counter by 1.

6. Return the value of the `distance` counter.

Here's the implementation in Python:

```python

def findTheDistanceValue(arr1, arr2, d):

distance = 0 # Initialize the distance counter

for num1 in arr1:

for num2 in arr2:

if abs(num1 - num2) <= d:

break # Break the inner loop if the condition is satisfied

else:

distance += 1 # Increment the distance counter if the inner loop completes without breaking

return distance

```

Let's test it with the given example:

```python

arr1 = [4, 5, 8]

arr2 = [10, 9, 1, 8]

d = 2

print(findTheDistanceValue(arr1, arr2, d))

```

Output:

```

2

```

The function returns a distance value of 2, which represents the number of elements in `arr1` such that there is no element in `arr2` satisfying the condition |arr1[i] - arr2[j]| <= d.

Q. Given an integer array nums of length n where all the integers of nums are in the range [1, n] and each integer appears **once** or **twice**, return an array of all the integers that appears ***twice***.

You must write an algorithm that runs in O(n) time and uses only constant extra space.

**Example 1:**

**Input:** nums = [4,3,2,7,8,2,3,1]

**Output:**

[2,3]

Ans: To find all the integers that appear twice in the given array `nums`, we can utilize the property that each integer in the array is in the range [1, n] and appears once or twice. This property allows us to use the array itself as a container to keep track of the visited elements.

Here's the step-by-step approach to solve this problem in O(n) time and with constant extra space:

1. Initialize an empty list called `result` to store the integers that appear twice.

2. Iterate through each element `num` in `nums`.

3. Calculate the absolute value of `num` and subtract 1 to convert it to a valid index.

4. Check if the element at the calculated index is positive. If it is, set it to its negation. This step marks the element as visited.

5. If the element at the calculated index is already negative, it means we have encountered it before. In this case, append the absolute value of `num` to the `result` list.

6. Return the `result` list.

Here's the implementation in Python:

```python

def findDuplicates(nums):

result = [] # Initialize the result list

for num in nums:

index = abs(num) - 1

if nums[index] < 0:

result.append(abs(num))

else:

nums[index] \*= -1

return result

```

Let's test it with the given example:

```python

nums = [4, 3, 2, 7, 8, 2, 3, 1]

print(findDuplicates(nums))

```

Output:

```

[2, 3]

```

The function returns a list `[2, 3]`, which represents the integers that appear twice in the given array `nums`.

Q. 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.

Ans: To find the minimum element in a sorted rotated array, we can use a modified binary search algorithm. Since the array is sorted and rotated, we can compare the middle element with the first and last elements to determine which half of the array to search.

Here's the step-by-step approach to solve this problem in O(log n) time:

1. Initialize two pointers, `left` and `right`, to the first and last indices of the array, respectively.

2. Perform a binary search while `left` is less than `right`.

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

4. Compare the middle element with the first and last elements of the array.

- If the middle element is less than the last element, it means the minimum element is in the left half of the array. Set `right = mid`.

- If the middle element is greater than the last element, it means the minimum element is in the right half of the array. Set `left = mid + 1`.

5. Repeat steps 3-4 until `left` is equal to `right`.

6. Return the element at index `left`, which represents the minimum element.

Here's the implementation in Python:

```python

def findMin(nums):

left = 0

right = len(nums) - 1

while left < right:

mid = left + (right - left) // 2

if nums[mid] < nums[right]:

right = mid

else:

left = mid + 1

return nums[left]

```

Let's test it with the given example:

```python

nums = [3, 4, 5, 1, 2]

print(findMin(nums))

```

Output:

```

1

```

The function returns 1, which represents the minimum element in the given sorted rotated array `nums`.

Q. An integer array original is transformed into a **doubled** array changed by appending **twice the value** of every element in original, and then randomly **shuffling** the resulting array.

Given an array changed, return original *if* changed *is a* ***doubled*** *array. If* changed *is not a* ***doubled*** *array, return an empty array. The elements in* original *may be returned in* ***any*** *order*.

**Example 1:**

**Input:** changed = [1,3,4,2,6,8]

**Output:** [1,3,4]

**Explanation:** One possible original array could be [1,3,4]:

* Twice the value of 1 is 1 \* 2 = 2.
* Twice the value of 3 is 3 \* 2 = 6.
* Twice the value of 4 is 4 \* 2 = 8.

Other original arrays could be [4,3,1] or [3,1,4].

Ans: To solve this problem, we can use a dictionary to keep track of the count of each element in the `changed` array. Then, we iterate through the `changed` array and check if the current element's count is greater than zero. If it is, we reduce the count of the current element by one and also check if the element's double is present in the dictionary and its count is greater than zero. If any of these conditions fail, we return an empty array because it means `changed` is not a doubled array.

Here's the step-by-step approach to solve this problem:

1. Initialize an empty dictionary called `count` to store the count of each element in `changed`.

2. Iterate through each element `num` in `changed` and increment its count in the `count` dictionary.

3. Iterate through each element `num` in `changed` and check if its count in the `count` dictionary is greater than zero.

- If the count is zero, continue to the next element.

- If the count is greater than zero, check if the double of `num` is present in the `count` dictionary and its count is greater than zero.

- If the double of `num` is not present or its count is zero, return an empty array.

- If the double of `num` is present and its count is greater than zero, reduce the count of `num` and the double of `num` by one in the `count` dictionary.

4. Return a list containing the elements for which their count is greater than zero in the `count` dictionary.

Here's the implementation in Python:

```python

def findOriginalArray(changed):

if len(changed) % 2 != 0:

return [] # If the length of changed is odd, it cannot be a doubled array

count = {} # Dictionary to store the count of each element in changed

for num in changed:

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

original = []

for num in changed:

if count[num] == 0:

continue

double = num \* 2

if double not in count or count[double] == 0:

return [] # changed is not a doubled array

original.append(num)

count[num] -= 1

count[double] -= 1

return original

```

Let's test it with the given example:

```python

changed = [1, 3, 4, 2, 6, 8]

print(findOriginalArray(changed))

```

Output:

```

[1, 3, 4]

```

The function returns `[1, 3, 4]`, which represents one possible original array that could be transformed into the given `changed` array.