Q. Given a non-negative integer x, return *the square root of* x *rounded down to the nearest integer*. The returned integer should be **non-negative** as well.

You **must not use** any built-in exponent function or operator.

* For example, do not use pow(x, 0.5) in c++ or x \*\* 0.5 in python.

Example-1

Input: x = 4

Output: 2

Explanation: The square root of 4 is 2, so we return 2.

Example-2

Input: x = 8

Output: 2

Explanation: The square root of 8 is 2.82842..., and since we round it down to the nearest integer, 2 is returned.

Ans: To find the square root of a non-negative integer x, rounded down to the nearest integer, we can use the binary search algorithm. Here's the step-by-step process:

1. If x is 0 or 1, the square root is equal to x, so return x.

2. Initialize two variables, 'left' and 'right'. 'left' will be set to 0, and 'right' will be set to x.

3. Perform a binary search until 'left' becomes greater than 'right'.

4. In each iteration of the binary search, calculate the mid-point as (left + right) / 2 and assign it to a variable 'mid'.

5. Check if 'mid' squared is greater than x. If it is, update 'right' to 'mid - 1'.

6. If 'mid' squared is less than or equal to x, update 'left' to 'mid + 1'.

7. After the binary search completes, return the value of 'right' as the square root of x rounded down to the nearest integer.

Here's the implementation in Python:

```python

def mySqrt(x):

if x == 0 or x == 1:

return x

left = 0

right = x

while left <= right:

mid = (left + right) // 2

if mid \* mid > x:

right = mid - 1

else:

left = mid + 1

return right

# Test cases

print(mySqrt(4)) # Output: 2

print(mySqrt(8)) # Output: 2

```

The time complexity of this algorithm is O(log(x)) because it performs a binary search to find the square root.

Q. A peak element is an element that is strictly greater than its neighbors.

Given a **0-indexed** integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.

You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

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

**Example 1:**

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

Output: 2

Explanation: 3 is a peak element and your function should return the index number 2.

**Example 2:**

Input: nums = [1,2,1,3,5,6,4]

Output: 5

Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

Ans: To find a peak element in a 0-indexed array `nums`, we can use a modified version of the binary search algorithm. Here's the step-by-step process:

1. Initialize two variables, `left` and `right`. `left` will be set to 0, and `right` will be set to the length of `nums` minus 1.

2. Perform a binary search until `left` becomes greater than `right`.

3. In each iteration of the binary search, calculate the mid-point as `(left + right) // 2` and assign it to a variable `mid`.

4. Check if `nums[mid]` is greater than its adjacent elements, i.e., `nums[mid - 1]` and `nums[mid + 1]`. If it is, return `mid` as the index of the peak element.

5. If `nums[mid]` is less than `nums[mid + 1]`, update `left` to `mid + 1` as the peak element must be on the right side of `mid`.

6. If `nums[mid]` is less than `nums[mid - 1]`, update `right` to `mid - 1` as the peak element must be on the left side of `mid`.

7. After the binary search completes, return `left` as the index of the peak element.

Here's the implementation in Python:

```python

def findPeakElement(nums):

left = 0

right = len(nums) - 1

while left < right:

mid = (left + right) // 2

if nums[mid] > nums[mid + 1]:

right = mid

else:

left = mid + 1

return left

# Test cases

print(findPeakElement([1, 2, 3, 1])) # Output: 2

print(findPeakElement([1, 2, 1, 3, 5, 6, 4])) # Output: 5

```

The time complexity of this algorithm is O(log n) because it performs a binary search on the array `nums` to find the peak element.

Q. Given an array nums containing n distinct numbers in the range [0, n], return *the only number in the range that is missing from the array.*

**Example 1:**

Input: nums = [3,0,1]

Output: 2

Explanation: n = 3 since there are 3 numbers, so all numbers are in the range [0,3]. 2 is the missing number in the range since it does not appear in nums.

**Example 2:**

Input: nums = [0,1]

Output: 2

Explanation: n = 2 since there are 2 numbers, so all numbers are in the range [0,2]. 2 is the missing number in the range since it does not appear in nums.

**Example 3:**

Input: nums = [9,6,4,2,3,5,7,0,1]

Output: 8

Explanation: n = 9 since there are 9 numbers, so all numbers are in the range [0,9]. 8 is the missing number in the range since it does not appear in nums.

Ans: To find the missing number in the range [0, n] from a distinct number array `nums`, we can utilize the mathematical formula for the sum of consecutive numbers from 0 to n. Here's the step-by-step process:

1. Calculate the expected sum of consecutive numbers from 0 to n using the formula `expected\_sum = (n \* (n + 1)) / 2`.

2. Iterate through each element `num` in `nums` and subtract it from the `expected\_sum`. The remaining value will be the missing number.

3. Return the remaining value as the missing number.

Here's the implementation in Python:

```python

def missingNumber(nums):

n = len(nums)

expected\_sum = (n \* (n + 1)) // 2

for num in nums:

expected\_sum -= num

return expected\_sum

# Test cases

print(missingNumber([3, 0, 1])) # Output: 2

print(missingNumber([0, 1])) # Output: 2

print(missingNumber([9, 6, 4, 2, 3, 5, 7, 0, 1])) # Output: 8

```

The time complexity of this algorithm is O(n) since we iterate through the entire `nums` array to calculate the missing number.

Q. Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive.

There is only **one repeated number** in nums, return *this repeated number*.

You must solve the problem **without** modifying the array nums and uses only constant extra space

Input: nums = [1,3,4,2,2]

Output: 2

Input: nums = [3,1,3,4,2]

Output: 3

Ans: To find the repeated number in the array `nums` without modifying the array and using only constant extra space, we can utilize the Floyd's Tortoise and Hare algorithm, also known as the cycle detection algorithm. Here's the step-by-step process:

1. Initialize two pointers, `slow` and `fast`, both starting from the first element of `nums`.

2. Move `slow` one step at a time, and `fast` two steps at a time until they meet inside the cycle.

3. Once `slow` and `fast` meet, reset `slow` to the first element of `nums`.

4. Move both `slow` and `fast` one step at a time until they meet again. The point at which they meet is the start of the cycle.

5. Return the value at the meeting point as the repeated number.

Here's the implementation in Python:

```python

def findDuplicate(nums):

slow = nums[0]

fast = nums[0]

# Find the meeting point

while True:

slow = nums[slow]

fast = nums[nums[fast]]

if slow == fast:

break

# Find the start of the cycle

slow = nums[0]

while slow != fast:

slow = nums[slow]

fast = nums[fast]

return slow

# Test cases

print(findDuplicate([1, 3, 4, 2, 2])) # Output: 2

print(findDuplicate([3, 1, 3, 4, 2])) # Output: 3

```

The time complexity of this algorithm is O(n) as we traverse the array once, and the space complexity is O(1) as we only use two pointers.

Q. Given two integer arrays nums1 and nums2, return *an array of their intersection*. Each element in the result must be **unique** and you may return the result in **any order**.

Input: nums1 = [1,2,2,1], nums2 = [2,2]

Output: [2]

Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]

Output: [9,4]

Explanation: [4,9] is also accepted.

Ans: To find the intersection of two integer arrays `nums1` and `nums2`, we can utilize Python's built-in set data structure. Here's the step-by-step process:

1. Convert `nums1` and `nums2` to sets, `set1` and `set2`, respectively.

2. Use the intersection operation `&` between `set1` and `set2` to find the common elements.

3. Convert the resulting set back to a list, `intersection\_list`, and return it.

Here's the implementation in Python:

```python

def intersection(nums1, nums2):

set1 = set(nums1)

set2 = set(nums2)

intersection\_set = set1 & set2

intersection\_list = list(intersection\_set)

return intersection\_list

# Test cases

print(intersection([1, 2, 2, 1], [2, 2])) # Output: [2]

print(intersection([4, 9, 5], [9, 4, 9, 8, 4])) # Output: [9, 4]

```

The time complexity of this algorithm is O(n+m), where n and m are the lengths of `nums1` and `nums2`, respectively. The set operations have a time complexity of O(len(set)), which is typically O(1) on average.

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.

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

Output: 1

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

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

Output: 0

Explanation: The original array was [0,1,2,4,5,6,7] and it was rotated 4 times.

Input: nums = [11,13,15,17]

Output: 11

Explanation: The original array was [11,13,15,17] and it was rotated 4 times.

Ans: To find the minimum element in a sorted rotated array `nums` of unique elements, we can use a modified version of the binary search algorithm. Here's the step-by-step process:

1. Initialize two pointers, `left` and `right`. `left` will be set to 0, and `right` will be set to the last index of `nums`.

2. Perform a binary search until `left` becomes greater than `right`.

3. In each iteration of the binary search, calculate the mid-point as `(left + right) // 2` and assign it to a variable `mid`.

4. Check if `nums[mid]` is greater than `nums[right]`. If it is, the minimum element must be on the right side of `mid`, so update `left` to `mid + 1`.

5. If `nums[mid]` is less than or equal to `nums[right]`, the minimum element must be on the left side of `mid`, so update `right` to `mid`.

6. After the binary search completes, return the value of `nums[left]` as 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) // 2

if nums[mid] > nums[right]:

left = mid + 1

else:

right = mid

return nums[left]

# Test cases

print(findMin([3, 4, 5, 1, 2])) # Output: 1

print(findMin([4, 5, 6, 7, 0, 1, 2])) # Output: 0

print(findMin([11, 13, 15, 17])) # Output: 11

```

The time complexity of this algorithm is O(log n) as it performs a binary search on the array `nums` to find the minimum element.

Q. Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

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

Input: nums = [5,7,7,8,8,10], target = 8

Output: [3,4]

Input: nums = [5,7,7,8,8,10], target = 6

Output: [-1,-1]

Input: nums = [], target = 0

Output: [-1,-1]

Ans: To find the starting and ending positions of a given target value in a sorted non-decreasing array `nums`, we can utilize a modified version of the binary search algorithm. Here's the step-by-step process:

1. Initialize two variables, `left` and `right`. `left` will be set to 0, and `right` will be set to the last index of `nums`.

2. Perform two binary searches, one to find the leftmost occurrence of the target and another to find the rightmost occurrence.

a. For the leftmost occurrence, calculate the mid-point as `(left + right) // 2` and assign it to a variable `mid`.

- If `nums[mid]` is greater than or equal to the target, update `right` to `mid`.

- If `nums[mid]` is less than the target, update `left` to `mid + 1`.

b. For the rightmost occurrence, calculate the mid-point as `(left + right) // 2` and assign it to `mid`.

- If `nums[mid]` is greater than the target, update `right` to `mid - 1`.

- If `nums[mid]` is less than or equal to the target, update `left` to `mid`.

3. After the binary searches complete, check if `nums[left]` is equal to the target. If it is not, return `[-1, -1]` to indicate that the target is not found.

4. If `nums[left]` is equal to the target, return `[left, right]` as the starting and ending positions of the target in `nums`.

Here's the implementation in Python:

```python

def searchRange(nums, target):

def binarySearchLeft(nums, target):

left = 0

right = len(nums) - 1

while left <= right:

mid = (left + right) // 2

if nums[mid] < target:

left = mid + 1

else:

right = mid - 1

return left

def binarySearchRight(nums, target):

left = 0

right = len(nums) - 1

while left <= right:

mid = (left + right) // 2

if nums[mid] <= target:

left = mid + 1

else:

right = mid - 1

return right

leftmost = binarySearchLeft(nums, target)

rightmost = binarySearchRight(nums, target)

if leftmost <= rightmost and nums[leftmost] == target:

return [leftmost, rightmost]

else:

return [-1, -1]

# Test cases

print(searchRange([5, 7, 7, 8, 8, 10], 8)) # Output: [3, 4]

print(searchRange([5, 7, 7, 8, 8, 10], 6)) # Output: [-1, -1]

print(searchRange([], 0)) # Output: [-1, -1]

```

The time complexity of this algorithm is O(log n) as it performs two binary searches on the array `nums` to find the starting and ending positions of the target.

Q. Given two integer arrays nums1 and nums2, return *an array of their intersection*. Each element in the result must appear as many times as it shows in both arrays and you may return the result in **any order**.

Input: nums1 = [1,2,2,1], nums2 = [2,2]

Output: [2,2]

Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]

Output: [4,9]

Explanation: [9,4] is also accepted.

Ans: To find the intersection of two integer arrays `nums1` and `nums2` while considering the occurrence of elements, we can use a hash map to store the frequency of elements in `nums1`. Then, iterate through `nums2` and check if the element exists in the hash map and its frequency is greater than zero. If it does, add the element to the result array and decrement its frequency in the hash map.

Here's the step-by-step process:

1. Create a hash map, `freqMap`, to store the frequency of elements in `nums1`.

2. Iterate through `nums1` and update the frequency of each element in `freqMap`.

3. Create an empty list, `intersection`, to store the intersection elements.

4. Iterate through `nums2` and check if the element exists in `freqMap` and its frequency is greater than zero. If it does, add the element to `intersection` and decrement its frequency in `freqMap`.

5. Return the `intersection` list as the result.

Here's the implementation in Python:

```python

from collections import defaultdict

def intersect(nums1, nums2):

freqMap = defaultdict(int)

for num in nums1:

freqMap[num] += 1

intersection = []

for num in nums2:

if freqMap[num] > 0:

intersection.append(num)

freqMap[num] -= 1

return intersection

# Test cases

print(intersect([1, 2, 2, 1], [2, 2])) # Output: [2, 2]

print(intersect([4, 9, 5], [9, 4, 9, 8, 4])) # Output: [4, 9]

```

The time complexity of this algorithm is O(m + n), where m and n are the lengths of `nums1` and `nums2`, respectively. We iterate through both arrays once to populate the frequency map and find the intersection elements.