Question 1

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.

Soln:

```
def sqrt(x):
    if x == 0:
        return 0

    left, right = 1, x

    while left <= right:
        mid = (left + right) // 2
        if mid * mid == x:
            return mid
        elif mid * mid < x:
            left = mid + 1
        else:
            right = mid - 1

    # Return the value of right (the largest integer less than or equal to the square root of x)
    return right
```

Question 2:

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.

Soln:

```
def find_peak_element(nums):
    left, right = 0, len(nums) - 1

    while left < right:
        mid = left + (right - left) // 2

        if nums[mid] < nums[mid + 1]:
            left = mid + 1
        else:
            right = mid

    return left
```

Question 3:

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.*

Soln:

```
def find_missing_number(nums):
    n = len(nums)
    expected_sum = (n * (n + 1)) // 2
    actual_sum = sum(nums)
    missing_number = expected_sum - actual_sum
    return missing_number
```

Question 4

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.

Soln:

```
def find_repeated_number(nums):
    slow = fast = nums[0]

    # Move slow and fast pointers to find the meeting point
    while True:
        slow = nums[slow]
```

```python
        fast = nums[nums[fast]]
        if slow == fast:
            break

    # Reset slow pointer to the start and move both pointers one step at a time
    slow = nums[0]
    while slow != fast:
        slow = nums[slow]
        fast = nums[fast]

    # Return the repeated number
    return slow
```

Question 5

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.

Soln:

```python
def intersection(nums1, nums2):
    set1 = set(nums1)
    set2 = set(nums2)
    intersection_set = set1.intersection(set2)
    return list(intersection_set)
```

Question 6

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.`

Soln:

```
def find_minimum(nums):
    left, right = 0, len(nums) - 1


    while left < right:
        mid = left + (right - left) // 2


        if nums[mid] > nums[right]:
            left = mid + 1
        else:
            right = mid


    return nums[left]
```

Question 7

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.

Soln:

```
def search_range(nums, target):

    # Function to find the leftmost occurrence of the target

    def find_leftmost(nums, target):

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

        index = -1


        while left <= right:

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

            if nums[mid] >= target:

                right = mid - 1

                if nums[mid] == target:

                    index = mid

            else:

                left = mid + 1


        return index
```

```python
    # Function to find the rightmost occurrence of the target
    def find_rightmost(nums, target):
        left, right = 0, len(nums) - 1
        index = -1

        while left <= right:
            mid = left + (right - left) // 2
            if nums[mid] <= target:
                left = mid + 1
                if nums[mid] == target:
                    index = mid
            else:
                right = mid - 1

        return index

    # Perform the binary searches
    leftmost = find_leftmost(nums, target)
    rightmost = find_rightmost(nums, target)

    return [leftmost, rightmost]
```

Question 8:

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**.

Soln:

```
def intersection(nums1, nums2):
    freq = {}
    result = []

    # Count the frequency of elements in nums1
    for num in nums1:
        if num in freq:
            freq[num] += 1
        else:
            freq[num] = 1

    # Check for intersection while iterating through nums2
    for num in nums2:
        if num in freq and freq[num] > 0:
            result.append(num)
            freq[num] -= 1

    return result
```