ASSIGNMENT 11

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.

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

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

  left, right = 1, x
  result = 0

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

    if mid <= x // mid:
      result = mid
      left = mid + 1
    else:
      right = mid - 1

  return result
```

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

```
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
```

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.*

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

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

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

    return left
```

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.

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

    while left < right:
        mid = left + (right - left) // 2
        count = 0

        for num in nums:
            if num <= mid:
                count += 1

        if count <= mid:
            left = mid + 1
        else:
            right = mid

    return left
```

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**.

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

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

        if arr[mid] == target:
            return True
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return False


def intersection(nums1, nums2):
    nums1.sort()
    result = []

    for num in set(nums2):
        if binary_search(nums1, num):
            result.append(num)

    return result
```

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`.

```
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]
```

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.

```
def search_range(nums, target):
    left = search_left(nums, target)
    right = search_right(nums, target)
    return [left, right]


def search_left(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
        else:
            left = mid + 1

        if nums[mid] == target:
            index = mid

    return index


def search_right(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
        else:
            right = mid - 1

        if nums[mid] == target:
            index = mid

    return index
```

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**.

```
def intersect(nums1, nums2):
    result = []
    nums2.sort()

    for num in nums1:
        index = bisect.bisect_left(nums2, num)
        if index < len(nums2) and nums2[index] == num:
            result.append(num)
            nums2.pop(index)
return result
```