Class Notes 3

▶ Python

▶ JavaScript

▼ Java

Question 1

Implement pow(x, n), which calculates x raised to the power n (i.e., xn).

Example 1:

Input: x = 2.00000, n = 10

Output: 1024.00000

TC : O(log n)

SC : O(1)

```
public class Solution {
    public double pow(double x, int n) {
        if(n == 0)
            return 1;
        if(n<0){
            n = -n;
            x = 1/x;
        }
        return (n%2 == 0) ? pow(x*x, n/2) : x*pow(x*x, n/2);
    }
}
```

Question 2

A permutation of an array of integers is an arrangement of its members into a sequence or linear order.

- For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The next permutation of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the next permutation of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

- For example, the next permutation of arr = [1,2,3] is [1,3,2].
- Similarly, the next permutation of arr = [2,3,1] is [3,1,2].
- While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, *find the next permutation of* nums.

The replacement must be in place and use only constant extra memory.

Example 1:

Input: nums = [1,2,3]

Output: [1,3,2]

Time complexity: *O*(*n*). In worst case, only two scans of the whole array are needed.

Space complexity: *O*(1). No extra space is used. In place replacements are done.

```
public class Solution {
    public void nextPermutation(int[] nums) {
        int i = nums.length - 2;
        while (i >= 0 && nums[i + 1] <= nums[i]) {
            i--;
        }
        if (i >= 0) {
            int j = nums.length - 1;
            while (nums[j] <= nums[i]) {
                j--;
            }
            swap(nums, i, j);
        }
        reverse(nums, i + 1);
    }

    private void reverse(int[] nums, int start) {
        int i = start, j = nums.length - 1;
        while (i < j) {
            swap(nums, i, j);
            i++;
            j--;
            }
        }

        private void swap(int[] nums, int i, int j) {
            int temp = nums[i];
            nums[i] = nums[j];
            nums[j] = temp;
        }
    }
```

💡 **Question 3**

Given an array arr[] of distinct elements size N that is sorted and then around an unknown point, the task is to check if the array has a pair with a given sum X.

Examples :

Input: arr[] = {11, 15, 6, 8, 9, 10}, X = 16

Output: true

Explanation: There is a pair (6, 10) with sum 16

Time Complexity: O(n), where n is the length of the input array.

Space Complexity: O(1).

```
public class FindPairInRotatedArray {
    public static boolean findPair(int[] arr, int x) {
        int n = arr.length;
        // find pivot element
        int pivot = 0;
        for (int i = 0; i < n-1; i++) {
            if (arr[i] > arr[i+1]) {
                pivot = i+1;
                break;
            }
        }
        int left = pivot;
        int right = pivot-1;
        while (left != right) {
            if (arr[left] + arr[right] == x) {
                return true;
            }
            else if (arr[left] + arr[right] < x) {
                left = (left+1) % n;
            }
            else {
                right = (right-1+n) % n;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        int[] arr = {11, 15, 6, 8, 9, 10};
        int x = 16;
        System.out.println(findPair(arr, x));
    }
}
```

💡 **Question 4**

Given an array nums with n objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

Example 1:

Input: nums = [2,0,2,1,1,0]

Output: [0,0,1,1,2,2]

Solution:

TC : O(n)

SC : O(1)

```
class Solution {
  /*
  Dutch National Flag problem solution.
  */
  public void sortColors(int[] nums) {
    // for all idx < i : nums[idx < i] = 0
    // j is an index of element under consideration
    int p0 = 0, curr = 0;
    // for all idx > k : nums[idx > k] = 2
    int p2 = nums.length - 1;

    int tmp;
    while (curr <= p2) {
      if (nums[curr] == 0) {
        // swap p0-th and curr-th elements
        // i++ and j++
        tmp = nums[p0];
        nums[p0++] = nums[curr];
        nums[curr++] = tmp;
      }
      else if (nums[curr] == 2) {
        // swap k-th and curr-th elements
        // p2--
        tmp = nums[curr];
        nums[curr] = nums[p2];
        nums[p2--] = tmp;
      }
      else curr++;
    }
  }
}
```

💡 **Question 5**

Given an integer array nums, rotate the array to the right by k steps, where k is non-negative.

Example 1:

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

Output: [5,6,7,1,2,3,4]

Explanation:

rotate 1 steps to the right: [7,1,2,3,4,5,6]

rotate 2 steps to the right: [6,7,1,2,3,4,5]

rotate 3 steps to the right: [5,6,7,1,2,3,4]

Solution:

TC: O(n)

SC: O(1)

```
class Solution {
  public void rotate(int[] nums, int k) {
    k %= nums.length;
    reverse(nums, 0, nums.length - 1);
    reverse(nums, 0, k - 1);
    reverse(nums, k, nums.length - 1);
  }
  public void reverse(int[] nums, int start, int end) {
    while (start < end) {
      int temp = nums[start];
      nums[start] = nums[end];
      nums[end] = temp;
      start++;
      end--;
    }
  }
}
```

💡 **Question 6**

Given a binary array nums, return *the maximum number of consecutive 1's in the array*.

Example 1:

Input: nums = [1,1,0,1,1,1]

Output: 3

Explanation: The first two digits or the last three digits are consecutive 1s. The maximum number of consecutive 1s is 3.

Solution:

TC : O(n)

SC : O(1)

```
class Solution {
  public int findMaxConsecutiveOnes(int[] nums) {
    int count = 0;
    int maxCount = 0;
    for(int i = 0; i < nums.length; i++) {
      if(nums[i] == 1) {
        // Increment the count of 1's by one.
        count += 1;
      } else {
        // Find the maximum till now.
        maxCount = Math.max(maxCount, count);
        // Reset count of 1.
        count = 0;
      }
    }
    return Math.max(maxCount, count);
  }
}
```