1. Odd String Difference You are given an array of equal-length strings words. Assume that the length of each string is n. Each string words[i] can be converted into a difference integer array difference[i] of length n - 1 where difference[i][j] = words[i][j+1] - words[i][j] where 0 <= j <= n - 2. Note that the difference between two letters is the difference between their positions in the alphabet i.e. the position of 'a' is 0, 'b' is 1, and 'z' is 25. For example, for the string "acb", the difference integer array is [2 - 0, 1 - 2] = [2, -1]. All the strings in words have the same difference integer array, except one. You should find that string. Return the string in words that has different difference integer array.

   Code with output:

```
from collections import defaultdict

def odd_string_out(words):
    def get_difference_array(word):
        return [ord(word[i + 1]) - ord(word[i]) for i in range(len(word) - 1)]
    diff_arrays = [get_difference_array(word) for word in words]
    diff_count = defaultdict(int)
    for diff in diff_arrays:
        diff_count[tuple(diff)] += 1
    most_common_diff = max(diff_count, key=diff_count.get)
    for i, diff in enumerate(diff_arrays):
        if tuple(diff) != most_common_diff:
            return words[i]
words1 = ["adc", "wzy", "abc"]
words2 = ["aaa", "bob", "ccc", "ddd"]

print(odd_string_out(words1))  # Output: "abc"
print(odd_string_out(words2))  # Output: "bob"
```

   Time complexity:O(m*n)
   Space complexity:O(m*n)

2. Words Within Two Edits of Dictionary You are given two string arrays, queries and dictionary. All words in each array comprise of lowercase

English letters and have the same length. In one edit you can take a word from queries, and change any letter in it to any other letter. Find all words from queries that, after a maximum of two edits, equal some word from dictionary. Return a list of all words from queries, that match with some word from dictionary after a maximum of two edits. Return the words in the same order they appear in queries.

Code with output:

```
def words_within_two_edits(queries, dictionary):
    def count_differences(word1, word2):
        return sum(1 for a, b in zip(word1, word2) if a != b)

    result = []
    for query in queries:
        for dict_word in dictionary:
            if count_differences(query, dict_word) <= 2:
                result.append(query)
                break

    return result
queries1 = ["word", "note", "ants", "wood"]
dictionary1 = ["wood", "joke", "moat"]

queries2 = ["yes"]
dictionary2 = ["not"]

print(words_within_two_edits(queries1, dictionary1))  # Output:
["word", "note", "wood"]
print(words_within_two_edits(queries2, dictionary2))  # Output: []

#Time complexity:O(n)
#Space complexity:O(1)
```

3. Next Greater Element IV You are given a 0-indexed array of non-negative integers nums. For each integer in nums, you must find its respective second greater integer. The second greater integer of nums[i]

is nums[j] such that: j > i nums[j] > nums[i] There exists exactly one index k such that nums[k] > nums[i] and i < k < j. If there is no such nums[j], the second greater integer is considered to be -1. For example, in the array [1, 2, 4, 3], the second greater integer of 1 is 4, 2 is 3, and that of 3 and 4 is -1. Return an integer array answer, where answer[i] is the second greater integer of nums[i].

Code with output:

Greater element:

```
def next_greater_element_iv(nums):
    n = len(nums)
    result = [-1] * n
    first_greater_stack = []
    second_greater_stack = []

    for i in range(n):
        while first_greater_stack and nums[first_greater_stack[-1]] <
nums[i]:
            idx = first_greater_stack.pop()
            while second_greater_stack and second_greater_stack[-1][0] <
nums[i]:
                first_idx = second_greater_stack.pop()[1]
                result[first_idx] = nums[i]
            second_greater_stack.append((nums[idx], idx))
        first_greater_stack.append(i)

    return result
nums1 = [2, 4, 0, 9, 6]
nums2 = [3, 3]

print(next_greater_element_iv(nums1))  # Output: [9, 6, 6, -1, -1]
print(next_greater_element_iv(nums2))  # Output: [-1, -1]
```

avg of even numbers divisible by 3:

```
def average_value_even_divisible_by_three(nums):
    valid_nums = [num for num in nums if num % 6 == 0]
    if not valid_nums:
        return 0
    return sum(valid_nums) // len(valid_nums)
nums1 = [1, 3, 6, 10, 12, 15]
nums2 = [1, 2, 4, 7, 10]

print(average_value_even_divisible_by_three(nums1))  # Output: 9
print(average_value_even_divisible_by_three(nums2))  # Output: 0
```

min beautiful number:

```
def sum_of_digits(n):
    return sum(int(digit) for digit in str(n))

def minimum_beautiful_number(n, target):
    x = 0
    while sum_of_digits(n + x) > target:
        x += 1
    return x
n1, target1 = 16, 6
n2, target2 = 467, 6
n3, target3 = 1, 1

print(minimum_beautiful_number(n1, target1))  # Output: 4
print(minimum_beautiful_number(n2, target2))  # Output: 33
print(minimum_beautiful_number(n3, target3))  # Output: 0

#Time complexity:O(n)
#Space complexity:O(1)
```

4.Minimum Addition to Make Integer Beautiful You are given two positive integers n and target. An integer is considered beautiful if the sum of its digits is less than or equal to target. Return the minimum non-negative integer x such that n + x is beautiful. The input will be generated such that it is always possible to make n beautiful.

Code with output:

```
def sum_of_digits(number):
    return sum(int(digit) for digit in str(number))

def minimum_addition_to_make_beautiful(n, target):
    if sum_of_digits(n) <= target:
        return 0

    addition = 0
    power_of_ten = 1

    while sum_of_digits(n + addition) > target:
        digit_to_increment = 10 - ((n + addition) // power_of_ten % 10)
        addition += digit_to_increment * power_of_ten
        power_of_ten *= 10

    return addition
n1, target1 = 16, 6
n2, target2 = 467, 6
n3, target3 = 1, 1

print(minimum_addition_to_make_beautiful(n1, target1))  # Output: 4
print(minimum_addition_to_make_beautiful(n2, target2))  # Output: 33
print(minimum_addition_to_make_beautiful(n3, target3))  # Output: 0
```

#Time complexity:O(logn)

#Space complexity:O(1)

5. Sort Array by Moving Items to Empty Space You are given an integer array nums of size n containing each element from 0 to n - 1 (inclusive). Each of the elements from 1 to n - 1 represents an item, and the element 0 represents an empty space. In one operation, you can move any item to the empty space. nums is considered to be sorted if the numbers of all the items are in ascending order and the empty space is either at the beginning or at the end of the array. For example, if n = 4, nums is sorted if: ● nums = [0,1,2,3] or ● nums = [1,2,3,0] ...and considered to be unsorted otherwise. Return the minimum number of operations needed to sort nums.

Code with output:

Sorting by moving into empty array:

```
def min_operations_to_sort(nums):
    n = len(nums)
    visited = [False] * n
    cycles = 0

    def find_cycles():
        nonlocal cycles
        for i in range(n):
            if not visited[i]:
                current = i
                cycle_length = 0
                while not visited[current]:
```

```
                visited[current] = True
                current = nums[current]
                cycle_length += 1
            if cycle_length > 1:
                cycles += 1

    find_cycles()

    return cycles - 1 if nums[0] == 0 or nums[-1] == 0 else cycles
nums1 = [4, 2, 0, 3, 1]
nums2 = [1, 2, 3, 4, 0]
nums3 = [1, 0, 2, 4, 3]

print(min_operations_to_sort(nums1))  # Output: 3
print(min_operations_to_sort(nums2))  # Output: 0
print(min_operations_to_sort(nums3))  # Output: 2
```

Applying operations:

```
def min_operations_to_sort(nums):
    n = len(nums)
    visited = [False] * n
    cycles = 0

    def find_cycles():
        nonlocal cycles
```

```
        for i in range(n):
            if not visited[i]:
                current = i
                cycle_length = 0
                while not visited[current]:
                    visited[current] = True
                    current = nums[current]
                    cycle_length += 1
                if cycle_length > 1:
                    cycles += 1

    find_cycles()

    return cycles - 1 if nums[0] == 0 or nums[-1] == 0 else cycles
nums1 = [4, 2, 0, 3, 1]
nums2 = [1, 2, 3, 4, 0]
nums3 = [1, 0, 2, 4, 3]

print(min_operations_to_sort(nums1))  # Output: 3
print(min_operations_to_sort(nums2))  # Output: 0
print(min_operations_to_sort(nums3))  # Output: 2

#Time complexity:O(n)
#Space complexity:O(n)
```