1144. Decrease Elements To Make Array Zigzag
There are only two options, even or single. We will iterate through the whole array twice. The first time we pick even and get minimum neighbor i, and decrease itself to i - 1. In the second iteration, we would iterate on odd index elements and do the same operation. The result would be min(odd, even), TC is O(n). Also,we could iterate through once and add it to a stored array in the index i % 2

```
class Solution:
    def movesToMakeZigzag(self, nums: List[int]) -> int:
        even, single = 0, 0
        nums1 = nums[:]
        length = len(nums)

        for i in range(0, length, 2):
            if i - 1 >= 0 and nums[i] <= nums[i - 1]:
                temp = nums[i - 1]
                nums[i - 1] = nums[i] - 1
                even += temp - nums[i - 1]
            if i + 1 < length and nums[i] <= nums[i + 1]:
                temp = nums[i + 1]
                nums[i + 1] = nums[i] - 1
                even += temp - nums[i + 1]
        nums = nums1
        for i in range(1, length, 2):
            if i - 1 >= 0 and nums[i] <= nums[i - 1]:
                temp = nums[i - 1]
                nums[i - 1] = nums[i] - 1
                single += temp - nums[i - 1]
            if i + 1 < length and nums[i] <= nums[i + 1]:
                temp = nums[i + 1]
                nums[i + 1] = nums[i] - 1
                single += temp - nums[i + 1]
        return min(even, single)
```

1145. Binary Tree Coloring Game.
We need to calculate the number of three directions of coloring node, naming as left, right, parent, then we will check whether each one is larger than (n + 1) // 2, if it is, then return True, if not, return False. TC is O(n)

```
class Solution:
    def btreeGameWinningMove(self, root: TreeNode, n: int, x: int) -> bool:
        left, right, _ = self.traverse(root, x)
        rest = n - left - right - 1
        return left >= (n + 1) // 2 or right >= (n + 1) // 2 or rest >= (n + 1) // 2
    def traverse(self, node, des):
        if not node:
            return (0, 0, False)

        left1, right1, isNode = self.traverse(node.left, des)
        if isNode:
            return (left1, right1, True)

        left2, right2, isNode = self.traverse(node.right, des)
        if isNode:
            return (left2, right2, True)

        if node.val == des:
            return (left1, left2, True)
        else:
            return (left1 + left2 + 1, 0, False)
```

1146. Snapshot Array
We will use a nested hashmap to store the changing one, when setting, we will use snap_id as key,< index: value> as value. When getting, we will find available index from snap_id to 0. If it's not available, we will return 0. TC is O(n), Setting is O(1):

```
from collections import defaultdict
class SnapshotArray:

    def __init__(self, length: int):
        self.memo = defaultdict(dict)
        self.snap_time = 0

    def set(self, index: int, val: int) -> None:
        self.memo[self.snap_time][index] = val

    def snap(self) -> int:
        self.snap_time += 1
        return self.snap_time - 1

    def get(self, index: int, snap_id: int) -> int:
        for i in range(snap_id, -1, -1):
            if index in self.memo[i]:
                return self.memo[i][index]
        return 0
```

642. Design Search Autocomplete System

We will use Trie to store every sentences and its searching time. If the node is not the end of a word, its CNT is gonna be 0. CNT will count this sentence’s searching time. For input method, if the input is ‘#’, we will reset buf and add this sentence by 1 time. If not, we will check whether there is a node for this buf and we will find all sentences starting with the current buf and sort them, taking the first 3 as the result.

```
CNT = "COUNT"
Trie = lambda: collections.defaultdict(Trie, {CNT: 0})
class AutocompleteSystem:

    def __init__(self, sentences: List[str], times: List[int]):
        self.trie = Trie()
        self.buf = ''
        for sentence, time in zip(sentences, times):
            self.add_sentence(sentence, time)

    def input(self, c: str) -> List[str]:
        if c == '#':
            self.add_sentence(self.buf, 1)
            self.buf = ''
            return []
        res = []
        self.buf += c
        if self.has_nodes(res):
            return [s for _, s in sorted(res)[:3]]
        else:
            return []

    def has_nodes(self, res):
        node = self.trie
        for i in self.buf:
            if i not in node:
                return False
            node = node[i]

        self.find_sentences(node, self.buf, res)
        return True

    def find_sentences(self, node, path, res):
        for k, v in node.items():
            if k == CNT and v > 0:
                res.append((-v, path))
            elif k != CNT:
```

```
            self.find_sentences(v, path + k, res)



    def add_sentence(self, sentence, time):
        node = self.trie
        for ch in sentence:
            node = node[ch]
        node[CNT] += time
```

283. Move Zeros
We will always track zero’s index, once we iterate through the array and encounter the non-zero number, we will replace their places. And acculating zero_index by 1. O(n)

```
class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        zero = 0

        for i in range(len(nums)):
            if nums[i] != 0:
                nums[zero], nums[i] = nums[i], nums[zero]
                zero += 1
```