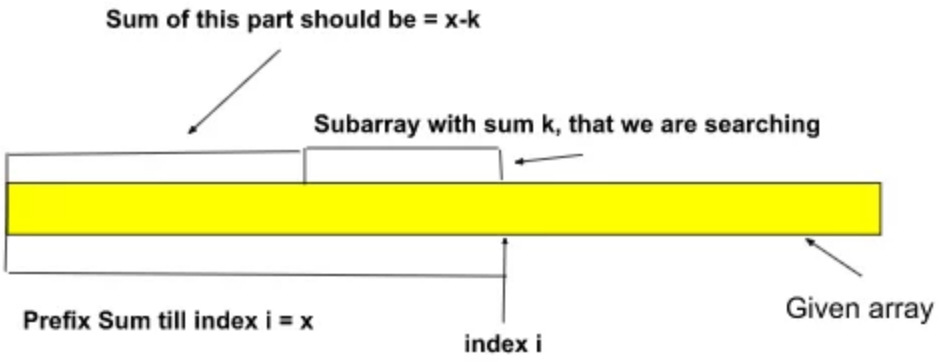
Arrays_2

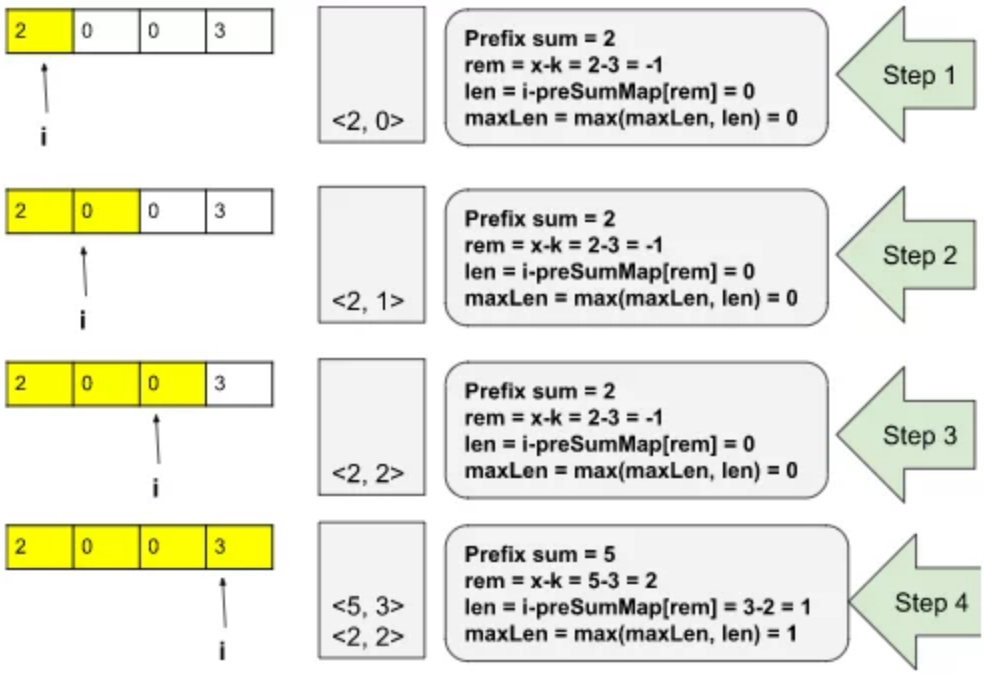
@kbbhatt04

@June 17, 2023

Subarray Sum Equals K

- Approach
  - Brute-force
    - Generate all subarrays and compute sum of each
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Optimized Brute-force
    - If we carefully observe, we can notice that to get the sum of the current subarray we just need to add the current element(i.e. arr[j]) to the sum of the previous subarray i.e. arr[i....j-1]
    - This is how we can remove the third loop and while moving the j pointer, we can calculate the sum.
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Better
    - Use prefix sum and hashmap





- Compute prefix sum and check if `sum[i] == k` then increment count by 1
- Also check if hashmap contains rem where `rem = sum[i] - k`, then increment count by hmap[rem]
- Lastly, if the sum[i] is not in hmap then add it else increment hmap[sum] by 1
- Time Complexity: $O(n)$
- Space Complexity: $O(n)$

◦ Optimal

- Take two pointers `left = 0` and `right = 0`
- Initialize `sum = 0`
- while `sum < k` increment the right pointer and add nums[right] to sum
- if sum exceeds k then subtract nums[left] from sum and increment left pointer
- Works iff array contains positive numbers
- Time Complexity: $O(n)$
- Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        cnt = 0
        for i in range(len(nums)):
            for j in range(i, len(nums)):
                s = 0

                for t in range(i, j+1):
                    s += nums[t]

                if s == k:  cnt += 1
        return cnt
```

```
# Python3
# Optimized Brute-force Solution
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        cnt = 0
        for i in range(len(nums)):
            s = 0
            for j in range(i, len(nums)):
```

```
                s += nums[j]

                if s == k:  cnt += 1
        return cnt
```

```
# Python3
# Better Solution
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        cnt = 0
        sum = 0
        hmap = {}

        for i in range(len(nums)):
            sum += nums[i]
            if sum == k:
                cnt += 1

            rem = sum - k
            if rem in hmap:
                cnt += hmap[rem]

            if sum not in hmap:
                hmap[sum] = 1
            else:
                hmap[sum] += 1

        return cnt
```

```
// C++
// Better Solution
class Solution {
public:
    int subarraySum(vector<int>& arr, int k) {
```

```
        int n = arr.size();
        int sum = 0;
        int ans = 0;
        unordered_map<int, int> umap;
        for(int i = 0; i < n; i++)
        {
            sum += arr[i];
            if (sum == k)   ans++;

            int rem = sum - k;
            if (umap.find(rem) != umap.end()) {
                ans += umap[rem];
            }

            if (umap.find(sum) == umap.end()) {
                umap[sum] = 1;
            }
            else {
                umap[sum]++;
            }
        }
        return ans;
    }
};
```

```
# Python3
# Optimal Solution
# Works iff array contains positive numbers
def getLongestSubarray(a: [int], k: int) -> int:
    n = len(a)
    left, right = 0, 0
    Sum = a[0]
    cnt = 0
    while right < n:
        while left <= right and Sum > k:
```

```
            Sum -= a[left]
            left += 1

        if Sum == k:
            cnt += 1

        right += 1
        if right < n: Sum += a[right]

    return cnt
```

```
// C++
// Optimal Solution
// Works iff array contains positive numbers
int getLongestSubarray(vector<int>& a, long long k) {
    int n = a.size();

    int left = 0, right = 0;
    long long sum = a[0];
    int cnt = 0;
    while (right < n) {
        while (left <= right && sum > k) {
            sum -= a[left];
            left++;
        }

        if (sum == k) {
            cnt++;
        }

        right++;
        if (right < n) sum += a[right];
    }
```

```
    return cnt;
}
```

Check If It Is a Straight Line

- Approach
  - Optimal
    - Check slope between the first two points and all other points
    - If the slope is same then the points are in-line else not
    - slope = `(y - y1) / (x - x1) = (y1 - y0) / (x1 - x0)`
    - Time Complexity: $O(n)$
    - Space Complexity: $O(1)$

```
# Python3
# Optimal Solution
class Solution:
    def checkStraightLine(self, coordinates: List[List[int]]) ->
        (x1, y1), (x2, y2) = coordinates[0:2]
        for i in range(2, len(coordinates)):
            (x, y) = coordinates[i]
                        # in order to avoid being divided by 0,
            if ((x-x1)*(y-y2)) != ((x-x2)*(y-y1)):
                return False
        return True
```

```
// C++
// Optimal Solution
class Solution {
public:
    bool checkStraightLine(vector<vector<int>>& coordinates) {
        int x1, x2, y1, y2;
        x1 = coordinates[0][0];
```

```
        y1 = coordinates[0][1];
        x2 = coordinates[1][0];
        y2 = coordinates[1][1];

        for (int i = 2; i < coordinates.size(); i++) {
            int x = coordinates[i][0];
            int y = coordinates[i][1];
            // in order to avoid being divided by 0, use multip
            if (((x-x1)*(y-y2)) != ((x-x2)*(y-y1))) return fals
        }
        return true;
    }
};
```

Search in a sorted matrix 1

First element of (i+1)th row will always be greater than last element of (i-1)th row

Input:

1 2 3 4 5

6 7 8 9 10

11 12 13 14 15

- Approach
  - Brute-force
    - Compare each element with the target
    - Time Complexity: $O(n * m)$
    - Space Complexity: $O(1)$
  - Optimal
    - Binary Search
    - Initialize `low = 0`, `high = (m * n) - 1` and `mid = (low + high) / 2`
    - We can get mid element of the matrix using `matrix[middle/m][middle%m]`

- Time Complexity: $O(log(n * m))$
- Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int)
        if len(matrix) == 0:    return False
        n = len(matrix)
        m = len(matrix[0])

        for i in range(n):
            for j in range(m):
                if matrix[i][j] == target:  return True
        return False
```

```
# Python3
# Optimal Solution
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int)
        if len(matrix) == 0:    return False
        n = len(matrix)
        m = len(matrix[0])

        lo = 0
        hi = m * n - 1

        while lo <= hi:
            mid = (lo + hi) // 2
            row = mid // m
            col = mid % m

            if matrix[row][col] == target:  return True
```

```
            if matrix[row][col] < target:   lo = mid + 1
            else:   hi = mid - 1
        return False
```

```
// C++
// Optimal Solution
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        if (!matrix.size()) return false;

        int n = matrix.size();
        int m = matrix[0].size();

        int lo = 0;
        int hi = (n * m) - 1;

        while (lo <= hi) {
            int mid = (lo + (hi - lo) / 2);

            if (matrix[mid/m][mid%m] == target) return true;

            if (matrix[mid/m][mid%m] < target)  lo = mid + 1;
            else    hi = mid - 1;
        }
        return false;
    }
};
```

Search in a sorted matrix 2

First element of (i+1)th row might be smaller than last element of (i-1)th row

Input:

1 3 5 7 9

2 4 6 8 10

5 6 7 8 9

- Approach
  - Brute-force
    - Compare each element with the target
    - Time Complexity: $O(n * m)$
    - Space Complexity: $O(1)$
  - Better
    - Apply Binary Search to each row of matrix
    - Time Complexity: $O(nlogm)$
    - Space Complexity: $O(1)$
  - Optimal
    - Initialize two pointers `i = 0` and `j = len(matrix[0]) - 1`
    - while the pointers are not out of bounds check if `matrix[i][j] == target`
    - if `matrix[i][j] < target` then `i += 1` else `j -= 1`
    - Time Complexity: $O(n + m)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int
        if len(matrix) == 0:    return False
        n = len(matrix)
        m = len(matrix[0])

        for i in range(n):
            for j in range(m):
```

```
                if matrix[i][j] == target:  return True
        return False
```

```
# Python3
# Better Solution
class Solution:
    def matSearch(self,mat, N, M, X):
        for i in range(N):
            lo = 0
            hi = M - 1

            while lo <= hi:
                mid = (lo + hi) // 2
                if mat[i][mid] == X:    return 1
                if mat[i][mid] < X: lo = mid + 1
                else:   hi = mid - 1
        return 0
```

```
# Python3
# Optimal Solution
class Solution:
    def matSearch(self,mat, N, M, X):
        i = 0
        j = M-1
        while i < N and j >= 0:
            if mat[i][j] == X:  return 1
            if mat[i][j] > X:   j -= 1
            else: i += 1
        return 0
```

```
// C++
// Optimal Solution
class Solution{
public:
```

```
    int matSearch (vector <vector <int>> &mat, int N, int M, int
    {
        int i = 0;
        int j = M - 1;
        while (i < N && j >= 0) {
            if (mat[i][j] == X) return 1;
            if (mat[i][j] < X)  i++;
            else    j--;
        }
        return 0;
    }
};
```

Row with max 1s in matrix

- Approach
  - Brute-force
    - Count the number of 1s in each row
    - Time Complexity: $O(n * m)$
    - Space Complexity: $O(1)$
  - Better
    - Apply Binary Search to find the index of first 1 in each row
    - Time Complexity: $O(nlogm)$
    - Space Complexity: $O(1)$
  - Optimal
    - Starting from the first row, initialize j = len(matrix[0])
    - while j is not out of bounds and `matrix[i][j] == 1` do `j -= 1` and set ans to that row
    - Time Complexity: $O(n + m)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def rowWithMax1s(self,arr, n, m):
        ans = -1
        max_cnt = 0
        for i in range(n):
            cnt = 0
            for j in range(m):
                if arr[i][j] == 1:
                    cnt += 1
            if cnt > max_cnt:
                max_cnt = cnt
                ans = i
        return ans
```

```
# Python3
# Better Solution
class Solution:
    def rowWithMax1s(self,arr, n, m):
        ans = -1
        max_ones_in_row = 0
        for i in range(n):
            l = 0
            h = m - 1
            while l <= h:
                mid = (l + h) // 2
                if arr[i][mid] == 1:
                    h = mid - 1
                else:
                    l = mid + 1
            if max_ones_in_row < m - l:
                max_ones_in_row = m - l
```

```
                ans = i
            return ans
```

```
# Python3
# Optimal Solution
class Solution:
    def rowWithMax1s(self,arr, n, m):
        ans = -1
        j = m - 1
        for i in range(n):
            while j >= 0 and arr[i][j] == 1:
                ans = i
                j -= 1
        return ans
```

```
// C++
// Optimal Solution
class Solution{
public:
    int rowWithMax1s(vector<vector<int> > arr, int n, int m) {
        int ans = -1;
        int j = m - 1;
        for (int i = 0; i < n; i++) {
            while (j >= 0 && arr[i][j] == 1) {
                ans = i;
                j--;
            }
        }
        return ans;
    }

};
```

Two Sum

- Approach
  - Brute-force
    - For every element at index i, run a loop from index i+1 to len(nums) to find if `nums[i]+nums[j] == target`
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Better
    - Store `[nums[i], i]` pair in an array
    - Sort that array
    - Use Two Pointer approach
    - Initialize `a = 0` and `b = len(nums) - 1`
    - increment/decrement a/b until either `a > b` or `nums[a] + nums[b] == target`
    - Time Complexity: $O(nlogn)$
    - Space Complexity: $O(n)$
  - Optimal
    - Initialize an unordered_map/dictionary with keys as elements and values as indices
    - Iterate over the nums and check for every element `i` whether `target-i` exists in the dict/map
    - Time Complexity: $O(n)$
    - Space Complexity: $O(n)$

```
# Python3
# Brute-force Solution
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]
        for i in range(len(nums)):
            for j in range(i+1, len(nums)):
```

```
                if nums[i] + nums[j] == target:
                    return [i, j]
        return [-1, -1]
```

```
# Python3
# Better Solution
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]
        o_nums = []
        for i in range(len(nums)):
            o_nums += [nums[i], i],
        o_nums.sort()

        a = 0
        b = len(nums) - 1

        while a < b:
            if o_nums[a][0] + o_nums[b][0] == target:
                return [o_nums[a][1], o_nums[b][1]]

            if o_nums[a][0] + o_nums[b][0] < target:
                a += 1
            else:
                b -= 1
        return [-1, -1]
```

```
// C++
// Better Solution
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        vector<pair<int, int>> v;
        for (int i = 0; i < nums.size(); i++)   v.push_back({nur
        sort(v.begin(), v.end());
```

```
        vector<int> ans;

        int a = 0, b = nums.size() - 1;
        while (a < b)
        {
            if (v[a].first + v[b].first == target)
            {
                ans.push_back(v[a].second);
                ans.push_back(v[b].second);
                return ans;
            }
            if (v[a].first + v[b].first < target)   a++;
            else    b--;
        }
        return ans;
    }
};
```

```
# Python3
# Optimal Solution
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]
        umap = dict()
        for i in range(len(nums)):
            if target - nums[i] in umap:
                return [i, umap[target - nums[i]]]
            umap[nums[i]] = i
        return [-1, -1]
```

```
// C++
// Optimal Solution
class Solution {
public:
```

```
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> umap;
        for (int i = 0; i < nums.size(); i++) {
            if (umap.find(target-nums[i]) != umap.end()) {
                return {umap[target - nums[i]], i};
            }
            umap[nums[i]] = i;
        }
        return {-1, -1};
    }
};
```

Majority Element (>n/2 times) (Moore's Voting Algorithm)

It is assumed that majority element always exists in the array

- Approach
  - Brute-force
    - For each element count the frequency of that element and check if the `frequency > len(nums)/2`
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Better
    - Maintain HashMap to store frequency of all elements
    - Iterate HashMap and return the element whose frequency > len(nums)/2
    - Time Complexity: $O(n)$ for unordered_map/dict best/average case

      [Worst Case for unordered_map/dict: $O(n^2)$]
    - Time Complexity: $O(nlogn)$ for map
    - Space Complexity: $O(n)$
  - Optimal

- Initialize `candidate = nums[0]` and `count = 0`
- Traverse through the array
- if `count == 0` then update the candidate to the current element and increment count
- else if current element is same as candidate then increment count
- else decrement count
- Time Complexity: $O(n)$
- Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        for i in nums:
            cnt = 0
            for j in nums:
                if i == j:
                    cnt += 1
            if cnt > len(nums) // 2:
                return i
        return -1
```

```
# Python3
# Better Solution
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        map = dict()
        for i in nums:
            if i in map:
                map[i] += 1
            else:
                map[i] = 1
```

```
        for i in map:
            if map[i] > len(nums) // 2:
                return i
        return -1
```

```
# Python3
# Optimal Solution (Moore’s Voting Algorithm)
class Solution:
    def majorityElement(self, nums):
        candidate, count = nums[0], 0
        for num in nums:
            if num == candidate:
                count += 1
            elif count == 0:
                candidate = num
                count += 1
            else:
                count -= 1
        return candidate
```

```
// C++
// Optimal Solution (Moore’s Voting Algorithm)
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int candidate = nums[0];
        int cnt = 0;
        for (auto i: nums) {
            if (i == candidate) {cnt++;}
            else if (cnt == 0) {candidate = i; cnt++;}
            else {cnt--;}
        }
        return candidate;
```

```
    }
};
```

Adding Ones

You start with an array **A** of size **N**. Initially all elements of the array A are **zero**. You will be given **K** positive integers. Let **j** be one of these integers, you have to add **1** to all **A[i]**, where **i ≥ j**. Your task is to print the array A after all these **K** updates are done. **Note:** 1-based indexing is used everywhere in this question.

Example 1:

```
Input:
N = 3, K = 4
1 1 2 3
Output:
2 3 4
Explanation:
Initially the array is {0, 0, 0}. After the
first 1, it becomes {1, 1, 1}. After the
second 1 it becomes {2, 2, 2}. After 2,
it becomes {2, 3, 3} and
After 3 it becomes, {2, 3, 4}.
```

Example 2:

```
Input:
N = 2, K = 3
1 1 1
Output:
3 3
Explanation:
Initially the array is {0, 0}. After the
first 1, it becomes {1, 1}. After the
second 1 it becomes {2, 2}. After the
third 1, it becomes {3, 3}.
```

- Approach
  - Brute-force
    - Run loop and increment all element from j to len(arr) by 1
    - Time Complexity: $O(n * k)$
    - Space Complexity: $O(1)$
  - Optimal
    - Increment the indices given in updates array by 1
    - Run loop over arr and do arr[i] += arr[i-1]
    - Time Complexity: $O(n)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def update(self, a, n, updates, k):
        for i in range(k):
            for j in range(updates[i]-1, n):
                a[j] += 1
```

```
# Python3
# Optimal Solution
class Solution:
    def update(self, a, n, updates, k):
        for i in range(k):
            a[updates[i] - 1] += 1
        for i in range(1, n):
            a[i] += a[i-1]
```

```
// C++
// Optimal Solution
```

```
class Solution{
    public:
    void update(int a[], int n, int updates[], int k)
    {
        for(int i = 0; i < k; i++)
            a[updates[i]-1]++;
        for(int i = 1; i < n; i++){
            a[i] += a[i - 1];
        }
    }
};
```

Max sum in subarrays

Given an array, find the maximum sum of smallest and second smallest elements chosen from all possible subarrays.

- Approach
  - Brute-force
    - Generate all subarrays
    - Find add min and max elements from all subarrays and return maximum
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Optimal
    - Indirectly this question is same as the max sum of two consecutive numbers because the two numbers will always be the smallest and second smallest for the subarray of two elements.
    - Time Complexity: $O(n)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def find(self, arr):
        mn, mx = arr[0], arr[1]
        for i in arr:
            if i < mn:
                mn = i
            elif i > mx:
                mx = i
        return mn, mx

    def pairWithMaxSum(self, arr, N):
        ans = arr[0] + arr[1]
        for i in range(N-1):
            for j in range(i+1, N):
                mn, mx = self.find(arr[i:j+1])
                ans = max(ans, mn+mx)
        return ans
```

```
# Python3
# Optimal Solution
class Solution:
    def pairWithMaxSum(self, arr, N):
        maxi = arr[0] + arr[1]
        for i in range(1, N):
            maxi = max(maxi, arr[i]+arr[i-1])
        return maxi
```

```
// C++
// Optimal Solution
#include <bits/stdc++.h>
class Solution{
    public:
```

```
        long long pairWithMaxSum(long long arr[], long long N)
        {
            long long ans = arr[0] + arr[1];
            for (int i = 1; i < N; i++) {
                ans = max(ans, arr[i]+arr[i-1]);
            }
            return ans;
        }
};
```

Rearrange array

There's an array 'A' of size 'N' with an equal number of positive and negative elements. Without altering the relative order of positive and negative elements, you must return an array of alternately positive and negative values.

- Approach
  - Brute-force
    - Take 2 array pos and neg
    - For each element in nums, append positive element in pos and negative element in neg array
    - Copy one pos and one neg element to original array
    - Time Complexity: $O(n)$
    - Space Complexity: $O(n)$
  - Optimal
    - Take an answer array of len(nums) and all elements initially 0
    - We know that all positive elements will be at `2*i` index and negative elements at `2*i+1` index
    - Maintain 2 pointers `ps = 0`, `ng = 1`
    - For each element in nums, if `nums[i] ≥ 0` then store it at `answer[ps] = nums[i]` and do `ps += 2` else store it at `answer[ng] = nums[i]` and do `ng += 2`

- One-Pass
- Time Complexity: $O(n)$
- Space Complexity: $O(n)$

```
# Python3
# Brute-force Solution
class Solution:
    def rearrangeArray(self, nums: List[int]) -> List[int]:
        pos = []
        neg = []
        for i in nums:
            if i < 0:
                neg += i,
            else:
                pos += i,

        for i in range(len(nums)//2):
            nums[2*i] = pos[i]
            nums[2*i + 1] = neg[i]

        return nums
```

```
# Python3
# Optimal Solution
class Solution:
    def rearrangeArray(self, nums: List[int]) -> List[int]:
        ans = [0 for i in range(len(nums))]
        pos, neg = 0, 1
        for i in range(len(nums)):
            if nums[i] < 0:
                ans[neg] = nums[i]
                neg += 2
            else:
                ans[pos] = nums[i]
```

```
                pos += 2
        return ans
```

```
// C++
// Optimal Solution
class Solution {
public:
    vector<int> rearrangeArray(vector<int>& nums) {
        vector<int> v(nums.size(), 0);
        int pos = 0, neg = 1;
        for (int i = 0; i < nums.size(); i++) {
            if (nums[i] < 0) {
                v[neg] = nums[i];
                neg += 2;
            }
            else {
                v[pos] = nums[i];
                pos += 2;
            }
        }
        return v;
    }
};
```

Leaders in an array

- Approach
  - Brute-force
    - For each element check if it is greater than all the elements on the right of it
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Optimal

- Initialize an empty stack
- Iterate over the array in reverse and push element nums[i] onto stack if `nums[i] ≥ stack[-1]`
- Reverse the stack and return stack
- Time Complexity: $O(n)$
- Space Complexity: $O(n)$

```
# Python3
# Brute-force Solution
class Solution:
    def leaders(self, A, N):
        leader = []
        for i in range(N):
            is_leader = True
            for j in range(i+1, N):
                if A[j] > A[i]:
                    is_leader = False
                    break
            if is_leader:
                leader += A[i],
        return leader
```

```
# Python3
# Optimal Solution
class Solution:
    def leaders(self, A, N):
        stack = []
        for i in range(N-1, -1, -1):
            if len(stack) == 0 or A[i] >= stack[-1]:
                stack += A[i],
        stack.reverse()
        return stack
```

```
// C++
// Optimal Solution
class Solution{
    public:
    vector<int> leaders(int a[], int n){
        vector<int> stack;
        stack.push_back(a[n-1]);
        for (int i = n - 2; i >= 0; i--) {
            if (a[i] >= stack.back()) {
                stack.push_back(a[i]);
            }
        }
        reverse(stack.begin(), stack.end());
        return stack;
    }
};
```

Template

- Approach
  - Brute-force
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Better
    -
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Optimal
    -

- Time Complexity: $O(n^3)$
- Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
```

```
# Python3
# Better Solution
```

```
# Python3
# Optimal Solution
```

```
// C++
// Optimal Solution
```