Arrays_3

@kbbhatt04

@June 17, 2023

Longest Consecutive Subsequence

- Approach
  - Brute-force
    - For each element count its consecutive elements present in the array
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Better
    - Sort the array and check for consecutive elements
    - Time Complexity: $O(nlogn)$
    - Space Complexity: $O(1)$
  - Optimal
    - Add all nums in set
    - Then we will run a for loop and check for any number(x) if it is the starting number of the consecutive sequence by checking if the set contains (x-1) or not
    - If x is the starting number of the consecutive sequence we will keep searching for the numbers y = x+1, x+2, x+3, ….. And stop at the first y which is not present in the set
    - Time Complexity: $O(n)$ (assuming set takes O(1) to search)
    - Space Complexity: $O(n)$

```
# Python3
# Brute-force Solution
class Solution:
    def find(self, nums, a):
        for i in nums:
            if i == a:
                return True
        return False

    def longestConsecutive(self, nums: List[int]) -> int:
        if len(nums) == 0:  return 0
        maxi = 1
        for i in nums:
            conseq = 1
            while self.find(nums, i + conseq):
                conseq += 1
            maxi = max(maxi, conseq)
        return maxi
```

```
# Python3
# Better Solution
class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        if len(nums) == 0:  return 0
        nums.sort()
        maxi = 1
        cur_max = 1
        for i in range(1, len(nums)):
            if nums[i] == nums[i-1]:
                continue
            if nums[i] == nums[i-1]+1:
                cur_max += 1
            else:
                cur_max = 1
            maxi = max(maxi, cur_max)
        return maxi
```

```
# Python3
# Optimal Solution
class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        if len(nums) == 0:  return 0
        uset = set(nums)
        maxi = 1
        for i in uset:
            if i-1 in uset:
                continue
            else:
                conseq = 1
                while i + conseq in uset:
                    conseq += 1
                maxi = max(maxi, conseq)
        return maxi
```

```
// C++
// Optimal Solution
#include <bits/stdc++.h>
class Solution{
  public:
    int findLongestConseqSubseq(int arr[], int N)
    {
        unordered_set<int> uset;
        for (int i = 0; i < N; i++) {
            uset.insert(arr[i]);
        }
        int maxi = 1;
        for (auto num: uset) {
            if (uset.count(num - 1)) {continue;}
            int conseq = 1;
            while (uset.count(num + conseq)) {
                conseq++;
            }
            maxi = max(maxi, conseq);
        }
        return maxi;
    }
};
```

Spiral Traversal of NxM Matrix

- Approach
    - Optimal
        - We print square by square
        - First print the outermost square then the inner one and so on
        - Initialize 4 variables `top = 0`, `bottom = N - 1`, `left = 0` and `right = M - 1`
        - Run loop until `top <= bottom` and `left <= right`
        - Print top row and increment top
        - Print right column and decrement right
        - If `top ≤ bottom`, Print bottom row and decrement bottom
        - if `left ≤ right`, Print left column and increase left
        - Time Complexity: $O(n * m)$
        - Space Complexity: $O(n * m)$

```
# Python3
# Optimal Solution
class Solution:
    def spiralOrder(self, matrix: List[List[int]]) -> List[int]:
        res = []
        if len(matrix) == 0:
            return res
        row_begin = 0
        col_begin = 0
        row_end = len(matrix)-1
        col_end = len(matrix[0])-1
        while (row_begin <= row_end and col_begin <= col_end):
            for i in range(col_begin,col_end+1):
                res.append(matrix[row_begin][i])
            row_begin += 1
            for i in range(row_begin,row_end+1):
                res.append(matrix[i][col_end])
            col_end -= 1
            if (row_begin <= row_end):
                for i in range(col_end,col_begin-1,-1):
                    res.append(matrix[row_end][i])
                row_end -= 1
            if (col_begin <= col_end):
                for i in range(row_end,row_begin-1,-1):
                    res.append(matrix[i][col_begin])
                col_begin += 1
        return res
```

```
// C++
// Optimal Solution
vector<int> printSpiral(vector<vector<int>> mat) {
  vector<int> ans;

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

  // Initialize the pointers reqd for traversal.
  int top = 0, left = 0, bottom = n - 1, right = m - 1;

  // Loop until all elements are not traversed.
  while (top <= bottom && left <= right) {

    // For moving left to right
    for (int i = left; i <= right; i++)
      ans.push_back(mat[top][i]);

    top++;

    // For moving top to bottom.
    for (int i = top; i <= bottom; i++)
      ans.push_back(mat[i][right]);

    right--;

    // For moving right to left.
    if (top <= bottom) {
      for (int i = right; i >= left; i--)
       ans.push_back(mat[bottom][i]);

      bottom--;
    }

    // For moving bottom to top.
    if (left <= right) {
      for (int i = bottom; i >= top; i--)
        ans.push_back(mat[i][left]);

      left++;
    }
  }
  return ans;
}
```

Majority Elements (>n/3 times)

Majority Elements(>N/3 times) | Find the elements that appears more than N/3 times in the array - Arrays - Tutorial

Detailed solution for Majority Elements(>N/3 times) | Find the elements that appears more than N/3 times in the array - Problem Statement: Given an array of N integers. Find the elements that appear more than N/3 times in the array. If no such element exists, return an empty vector. Pre-requisite: Majority Element(>N/2 times) Example 1: Input Format: N = 5, array[] = {1,2,2,3,2}

https://takeuforward.org/data-structure/majority-elementsn-3-times-find-the-elements-that-appears-more-than-n-3-times-in-the-array/

- Approach
  - Brute-force
    - For all unique elements, we count the occurrences of that element
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Better
    - Use HashMap to store frequency of all elements
    - Time Complexity: $O(n)$
    - Space Complexity: $O(n)$
  - Optimal
    - Apply extended Boyer Moore's Voting Algorithm
    - Time Complexity: $O(n)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def majorityElement(self, nums: List[int]) -> List[int]:
        n = len(nums)
        ans = []
        for i in nums:
            # we check if i is not already a part of ans
            # we compare i with ans[0] because at max there will be 2 majority elements
            if len(ans) == 0 or ans[0] != i:
                cnt = 0
                for j in nums:
                    if i == j:
                        cnt += 1
                if cnt > n // 3:
                    ans += i,
                if len(ans) == 2:
                    break
        return ans
```

```
# Python3
# Better Solution
class Solution:
    def majorityElement(self, nums: List[int]) -> List[int]:
        n = len(nums)
        ans = []
        dic = {}
        for i in nums:
            if i in dic:
                dic[i] += 1
            else:
                dic[i] = 1
        for i in dic:
            if dic[i] > n//3:
                ans += i,
        return ans
```

```
# Python3
# Optimal Solution
class Solution:
    def majorityElement(self, nums: List[int]) -> List[int]:
        n = len(nums)
        if n == 1:  return [nums[0]]
        e0, e1 = nums[0], nums[1]
        c0, c1 = 0, 0

        for i in nums:
            if c0 == 0 and i != e1:
                e0 = i
                c0 = 1
            elif c1 == 0 and i != e0:
                e1 = i
                c1 = 1
            elif i == e0:
                c0 += 1
            elif i == e1:
                c1 += 1
            else:
                c0 -= 1
                c1 -= 1

        ans = []
        c0 = 0
        for i in nums:
            if i == e0:
                c0 += 1
        if c0 > n // 3:
            ans += e0,
        c1 = 0
        for i in nums:
            if i == e1:
                c1 += 1
        if c1 > n // 3 and e1 != e0:
            ans += e1,
        return ans
```

```
// C++
// Optimal Solution
vector<int> majorityElement(vector<int> v) {
    int n = v.size();

    int cnt1 = 0, cnt2 = 0;
    int el1 = INT_MIN;
    int el2 = INT_MIN;

    for (int i = 0; i < n; i++) {
        if (cnt1 == 0 && el2 != v[i]) {
            cnt1 = 1;
            el1 = v[i];
        }
        else if (cnt2 == 0 && el1 != v[i]) {
            cnt2 = 1;
            el2 = v[i];
        }
        else if (v[i] == el1) cnt1++;
        else if (v[i] == el2) cnt2++;
        else {
            cnt1--, cnt2--;
        }
    }

    vector<int> ls;

    cnt1 = 0, cnt2 = 0;
    for (int i = 0; i < n; i++) {
        if (v[i] == el1) cnt1++;
        if (v[i] == el2) cnt2++;
    }

    int mini = int(n / 3) + 1;
    if (cnt1 >= mini) ls.push_back(el1);
    if (cnt2 >= mini && el2 != el1) ls.push_back(el2);

    return ls;
}
```

3Sum

Given an array, return all the triplets `[nums[i], nums[j], nums[k]]` such that `i != j`, `i != k`, `j != k` and `nums[i] + nums[j] + nums[k] == 0`.

Notice that the solution set must not contain duplicate triplets.

- Approach
    - Brute-force

        - We keep three-pointers `i`, `j` and `k`. For every triplet we find the sum of `A[i]+A[j]+A[k]`. If this sum is equal to zero, we've found one of the triplets. We add it to our data structure and continue with the rest.
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(m)$ where m is no. of triplets
    - Better
        - We store the frequency of each elements in a HashMap
        - Based on `a + b + c = 0` we can say that `c = -(a+b)` and based on this we fix two elements `a` and `b` and try to find the `-(a+b)` in HashMap
        - Time Complexity: $O(n^2)$
        - Space Complexity: $O(n + m)$ where m is no. of triplets
    - Optimal
        - Sort the array
        - Based on `a + b + c = 0` we can say that `b + c = -a` and based on this we fix element as `a` and then find `b` and `c` using two pointers `lo` and `hi` (same as in Two Sum Problem)
        - Time Complexity: $O(n^2)$
        - Space Complexity: $O(m)$ where m is no. of triplets

```python
# Python3
# Brute-force Solution
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        ans = set()

        for i in range(n-2):
            for j in range(i+1, n-1):
                for k in range(j+1, n):
                    if nums[i] + nums[j] + nums[k] == 0:
                        ans.add(tuple(sorted((nums[i], nums[j], nums[k]))))

        res = []
        for i in ans:
            res += list(i),
        return res
```

```python
# Python3
# Better Solution
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        hmap = dict()
        for i in range(n):
            if nums[i] in hmap:
                hmap[nums[i]] += 1
            else:
                hmap[nums[i]] = 1

        ans = set()
        for i in range(n-2):
            # decrease count of nums[i] so that we don't take same element twice
            hmap[nums[i]] -= 1
            for j in range(i+1, n-1):
                # decrease count of nums[j] so that we don't take same element twice
                hmap[nums[j]] -= 1
                rem = -(nums[i]+nums[j])
                if rem in hmap and hmap[rem] > 0:
                    ans.add(tuple(sorted((nums[i], nums[j], rem))))
                # again increment the nums[j] so that its reusable
                hmap[nums[j]] += 1
            # again increment the nums[i] so that its reusable
            hmap[nums[i]] += 1

        res = []
        for i in ans:
            res.append(list(i))
        return res
```

```
# Python3
# Optimal Solution
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        ans = []
        nums.sort()
        i = 0
        while i < n - 2:
            # we check this condition so that we dont get same triplets twice
            if i == 0 or (i > 0 and nums[i] != nums[i-1]):
                lo = i + 1
                hi = n - 1

                while lo < hi:
                    if nums[lo] + nums[hi] == -nums[i]:
                        ans += [nums[i], nums[lo], nums[hi]],
                        # increment lo till we get different element so as to not get same triplet twice
                        while lo < hi and nums[lo] == nums[lo + 1]:
                            lo += 1
                        # decrement hi till we get different element so as to not get same triplet twice
                        while lo < hi and nums[hi] == nums[hi - 1]:
                            hi -= 1
                        lo += 1
                        hi -= 1
                    elif nums[lo] + nums[hi] < -nums[i]:
                        lo += 1
                    else:
                        hi -= 1
            i += 1

        return ans
```

```
// C++
// Optimal Solution
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        int n = nums.size();
        vector<vector<int>> ans;
        sort(nums.begin(), nums.end());
        for (int i = 0; i < n - 2; i++) {
            if ((i == 0) || ((i > 0) && (nums[i] != nums[i-1]))) {
                int lo = i + 1;
                int hi = n - 1;

                while (lo < hi) {
                    if (nums[lo] + nums[hi] == -nums[i]) {
                        ans.push_back({nums[i], nums[lo], nums[hi]});

                        while ((lo < hi) && (nums[lo] == nums[lo+1])) {
                            lo++;
                        }
                        while ((lo < hi) && (nums[hi] == nums[hi-1])) {
                            hi--;
                        }
                        lo++;
                        hi--;
                    }
                    else if (nums[lo] + nums[hi] < -nums[i]) {
                        lo++;
                    }
                    else {
                        hi--;
                    }
                }
            }
        }
        return ans;
    }
};
```

4Sum

- Approach
    - Brute-force
        - We keep four-pointers `i`, `j`, `k` and `l`. For every quadruplet, we find the sum of `A[i]+A[j]+A[k]+A[l]`

        - If this sum equals the target, we've found one of the quadruplets and add it to our data structure and continue with the rest
        - Time Complexity: $O(n^4)$
        - Space Complexity: $O(m)$ where m is the number of quadruplets
    - Better
        - We store the frequency of each element in a HashMap
        - Based on `a + b + c + d = target` we can say that `d = target - (a+b+c)` and based on this we fix 3 elements `a` , `b` and `c` and try to find the `-(a+b+c)` in HashMap
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(n+m)$ where m is the number of quadruplets
    - Optimal
        - To get the quadruplets in sorted order, we will sort the entire array in the first step and to get the unique quads, we will simply skip the duplicate numbers while moving the pointers
        - Fix 2 pointers `i` and `j` and move 2 pointers `lo` and `hi`
        - Based on `a + b + c + d = target` we can say that `c + d = target - (a+b)` and based on this we fix element as `a` and `b` then find `c` and `d` using two pointers `lo` and `hi` (same as in 3Sum Problem)
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(m)$ where m is the number of quadruplets

```
# Python3
# Brute-force Solution
class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        n = len(nums)
        ans = set()
        for i in range(n-3):
            for j in range(i+1, n-2):
                for k in range(j+1, n-1):
                    for l in range(k+1, n):
                        if nums[i] + nums[j] + nums[k] + nums[l] == target:
                            ans.add(tuple(sorted((nums[i], nums[j], nums[k], nums[l]))))

        res = []
        for i in ans:
            res += list(i),
        return res
```

```
# Python3
# Better Solution
from collections import defaultdict
class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        n = len(nums)
        ans = set()
        hmap = defaultdict(int)
        for i in nums:
            hmap[i] += 1

        for i in range(n-3):
            hmap[nums[i]] -= 1
            for j in range(i+1, n-2):
                hmap[nums[j]] -= 1
                for k in range(j+1, n-1):
                    hmap[nums[k]] -= 1
                    rem = target-(nums[i] + nums[j] + nums[k])
                    if rem in hmap and hmap[rem] > 0:
                        ans.add(tuple(sorted((nums[i], nums[j], nums[k], rem))))
                    hmap[nums[k]] += 1
                hmap[nums[j]] += 1
            hmap[nums[i]] += 1

        res = []
        for i in ans:
            res += list(i),
        return res
```

```
# Python3
# Optimal Solution
class Solution:
    def fourSum(self, nums: List[int], target: int) -> List[List[int]]:
        n = len(nums)
        nums.sort()
        res = []

        for i in range(n-3):
            # avoid the duplicates while moving i
            if i > 0 and nums[i] == nums[i - 1]:
                continue
            for j in range(i+1, n-2):
                # avoid the duplicates while moving j
                if j > i + 1 and nums[j] == nums[j - 1]:
                    continue
                lo = j + 1
                hi = n - 1
                while lo < hi:
                    temp = nums[i] + nums[j] + nums[lo] + nums[hi]
                    if temp == target:
                        res += [nums[i], nums[j], nums[lo], nums[hi]],

                        # skip duplicates
                        while lo < hi and nums[lo] == nums[lo + 1]:
                            lo += 1
                        lo += 1
                        while lo < hi and nums[hi] == nums[hi - 1]:
                            hi -= 1
                        hi -= 1
                    elif temp < target:
                        lo += 1
                    else:
                        hi -= 1
        return res
```

```
// C++
// Optimal Solution
vector<vector<int>> fourSum(vector<int>& nums, int target) {
    int n = nums.size(); //size of the array
    vector<vector<int>> ans;

    // sort the given array:
    sort(nums.begin(), nums.end());

    //calculating the quadruplets:
    for (int i = 0; i < n; i++) {
        // avoid the duplicates while moving i:
        if (i > 0 && nums[i] == nums[i - 1]) continue;
        for (int j = i + 1; j < n; j++) {
            // avoid the duplicates while moving j:
            if (j > i + 1 && nums[j] == nums[j - 1]) continue;

            // 2 pointers:
            int k = j + 1;
            int l = n - 1;
            while (k < l) {
                long long sum = nums[i];
                sum += nums[j];
                sum += nums[k];
                sum += nums[l];
                if (sum == target) {
                    vector<int> temp = {nums[i], nums[j], nums[k], nums[l]};
                    ans.push_back(temp);
                    k++; l--;

                    //skip the duplicates:
                    while (k < l && nums[k] == nums[k - 1]) k++;
                    while (k < l && nums[l] == nums[l + 1]) l--;
                }
                else if (sum < target) k++;
                else l--;
            }
        }
    }

    return ans;
}
```

Largest Subarray with 0 Sum

- Approach
    - Brute-force
        - Check sum for all the subarrays and store the length
        - Time Complexity: $O(n^2)$
        - Space Complexity: $O(1)$
    - Optimal
        - If we know sum of `subarray(i…j) = S` and also know sum of `subarray(i…x) = S` and `i < x < j` then we conclude that sum of `subarray(x+1…j) = 0`
        - Thus we store prefix sum of all elements along with its index in a HashMap
        - Then iterate over all the elements in the array and maintain a variable sum which stores sum of elements till now
        - If `sum == 0` then `max_len = i + 1` else we find sum in HashMap, if found then update max_len to `max_len = max(max_len, i - umap[sum])` else store sum with its index in umap
        - Time Complexity: $O(n)$
        - Space Complexity: $O(n)$

```
# Python3
# Brute-force Solution
class Solution:
    def maxLen(self, n, arr):
        maxi = 0
        for i in range(n):
            sum = 0
            for j in range(i, n):
                sum += arr[j]
                if sum == 0:
                    maxi = max(maxi, j-i+1)
        return maxi
```

```
# Python3
# Optimal Solution
class Solution:
    def maxLen(self, n, arr):
        maxi = 0
        sum = 0
        mpp = {}
        for i in range(n):
            sum += arr[i]
            if sum == 0:
                maxi = i + 1
            else:
                if sum in mpp:
                    maxi = max(maxi, i - mpp[sum])
                else:
                    mpp[sum] = i
        return maxi
```

```
// C++
// Optimal Solution
#include <bits/stdc++.h>
class Solution{
    public:
    int maxLen(vector<int>&A, int n)
    {
        unordered_map<int, int> umap;
        int maxi = 0, sum = 0;
        for (int i = 0; i < n; i++) {
            sum += A[i];
            if (sum == 0) {maxi = i + 1;}
            else {
                if (umap.find(sum) != umap.end()) {
                    maxi = max(maxi, i - umap[sum]);
                }
                else {umap[sum] = i;}
            }
        }
        return maxi;
    }
};
```

Count the number of subarrays with xor equals k

- Approach
  - Brute-force
    - Generate all subarrays and maintain count
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Better
    - We don't need the 3rd loop
    - Maintain running xor of elements
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Optimal
    - Use the concept of the prefix XOR
    - Keep the occurrence of the prefix XOR of the subarrays using a map data structure
    - Question: Why do we need to set the value of 0 beforehand?
      Let’s understand this using an example. Assume the given array is [3, 3, 1, 1, 1] and k is 3. Now, for index 0, we get the total prefix XOR as 3, and k is also 3. So, the prefix XOR xr^k will be = 3^3 = 0. Now, if the value is not previously set for the key 0 in the map, we will get the default value 0 and we will add 0 to our answer. This will mean that we have not found any subarray with XOR 3 till now. But this should not be the case as index 0 itself is a subarray with XOR k i.e. 3.
      So, in order to avoid this situation we need to set the value of 0 as 1 on the map beforehand.
    - Time Complexity: $O(n)$

- Space Complexity: $O(n)$

```python
# Python3
# Brute-force Solution
def subarraysWithXorK(a: [int], b: int) -> int:
    n = len(a)  # size of the given array.
    cnt = 0

    # Step 1: Generating subarrays:
    for i in range(n):
        for j in range(i, n):

            # step 2: calculate XOR of all elements:
            xorr = 0
            for K in range(i, j + 1):
                xorr = xorr ^ a[K]

            # step 3: check XOR and count:
            if (xorr == k):
                cnt += 1

    return cnt
```

```python
# Python3
# Better Solution
def subarraysWithXorK(a: [int], b: int) -> int:
    n = len(a)  # size of the given array.
    cnt = 0

    # Step 1: Generating subarrays:
    for i in range(n):
        xorr = 0
        for j in range(i, n):

            # step 2: calculate XOR of all elements:
            xorr = xorr ^ a[j]

            # step 3: check XOR and count:
            if (xorr == k):
                cnt += 1

    return cnt
```

```python
# Python3
# Optimal Solution
def subarraysWithXorK(a: [int], b: int) -> int:
    n = len(a) # size of the given array.
    xr = 0
    mpp = defaultdict(int) # declaring the dictionary.
    mpp[xr] = 1 # setting the value of 0.
    cnt = 0

    for i in range(n):
        # prefix XOR till index i:
        xr = xr ^ a[i]

        # By formula: x = xr^k:
        x = xr ^ k

        # add the occurrence of xr^k
        # to the count:
        cnt += mpp[x]

        # Insert the prefix xor till index i
        # into the dictionary:
        mpp[xr] += 1

    return cnt
```

```cpp
// C++
// Optimal Solution
int subarraysWithXorK(vector<int> a, int k) {
    int n = a.size(); //size of the given array.
    int xr = 0;
    map<int, int> mpp; //declaring the map.
    mpp[xr]++; //setting the value of 0.
    int cnt = 0;
    for (int i = 0; i < n; i++) {
        // prefix XOR till index i:
        xr = xr ^ a[i];

        //By formula: x = xr^k:
        int x = xr ^ k;

        // add the occurrence of xr^k
        // to the count:
        cnt += mpp[x];

        // Insert the prefix xor till index i
        // into the map:
        mpp[xr]++;
    }
    return cnt;
}
```

Find the repeating and missing numbers

- Approach
  - Brute-force
    - For each element count its frequency
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Better
    - Store the frequency of each element between 1 to N
    - Time Complexity: $O(n)$
    - Space Complexity: $O(n)$
  - Optimal
    - Let X be the repeating number and Y be the missing number
    - Sum of first n natural numbers `Sn = (N * (N + 1)) / 2`
    - Let S be the sum of all the elements given in the array
    - Thus `S - Sn = X - Y` eq. 1
    - Sum of squares of first n natural numbers `S2n = (N * (N + 1) * (2N + 1)) / 6`
    - Let S2 be the sum of squares of all the elements given in the array
    - Thus `S2 - S2n =` X^2 `-` Y^2 eq. 2
    - Thus `X+Y = (S2 - S2n) / (X-Y)` eq. 3
    - Thus we can find X and Y from eq. 1 and eq. 3
    - Time Complexity: $O(n)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def findTwoElement(self, arr, n):
        ans = [-1, -1]
        for i in range(1, n+1):
            freq = 0
            for j in arr:
                if i == j:
                    freq += 1
            if freq == 0:
                ans[1] = i
            elif freq == 2:
                ans[0] = i
        return ans
```

```
# Python3
# Better Solution
from collections import defaultdict
class Solution:
    def findTwoElement(self, arr, n):
        ans = [-1, -1]
        dic = defaultdict(int)
        for i in arr:
            dic[i] += 1
        for i in range(1, n+1):
            if dic[i] == 0:
                ans[1] = i
            elif dic[i] == 2:
                ans[0] = i
        return ans
```

```
# Python3
# Optimal Solution
def findMissingRepeatingNumbers(a: [int]) -> [int]:
    n = len(a)  # size of the array

    # Find Sn and S2n:
    SN = (n * (n + 1)) // 2
    S2N = (n * (n + 1) * (2 * n + 1)) // 6

    # Calculate S and S2:
    S, S2 = 0, 0
    for i in range(n):
        S += a[i]
        S2 += a[i] * a[i]

    # S-Sn = X-Y:
    val1 = S - SN

    # S2-S2n = X^2-Y^2:
    val2 = S2 - S2N

    # Find X+Y = (X^2-Y^2)/(X-Y):
    val2 = val2 // val1

    # Find X and Y: X = ((X+Y)+(X-Y))/2 and Y = X-(X-Y),
    # Here, X-Y = val1 and X+Y = val2:
    x = (val1 + val2) // 2
    y = x - val1

    return [x, y]
```

```
// C++
// Optimal Solution
vector<int> findMissingRepeatingNumbers(vector<int> a) {
    long long n = a.size(); // size of the array

    // Find Sn and S2n:
    long long SN = (n * (n + 1)) / 2;
    long long S2N = (n * (n + 1) * (2 * n + 1)) / 6;

    // Calculate S and S2:
    long long S = 0, S2 = 0;
    for (int i = 0; i < n; i++) {
        S += a[i];
        S2 += (long long)a[i] * (long long)a[i];
    }

    //S-Sn = X-Y:
    long long val1 = S - SN;

    // S2-S2n = X^2-Y^2:
    long long val2 = S2 - S2N;

    //Find X+Y = (X^2-Y^2)/(X-Y):
    val2 = val2 / val1;

    //Find X and Y: X = ((X+Y)+(X-Y))/2 and Y = X-(X-Y),
    // Here, X-Y = val1 and X+Y = val2:
    long long x = (val1 + val2) / 2;
    long long y = x - val1;

    return {(int)x, (int)y};
}
```

Count Inversions

- Approach
    - Brute-force
        - Check for all pairs using two nested loops
        - Time Complexity: $O(n^2)$
        - Space Complexity: $O(1)$
    - Optimal
        - Consider a similar problem, let there be two sorted arrays a1 and a2

        - But in this question, we are given an unsorted array so we need to break it into two sorted halves
        - Merge sort algorithm works in similar way
        - In merge sort algorithm, while we merge two sorted arrays, we can count the inversion pairs
        - Time Complexity: $O(nlogn)$
        - Space Complexity: $O(1)$

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

```
# Python3
# Optimal Solution
from typing import List
import math
```

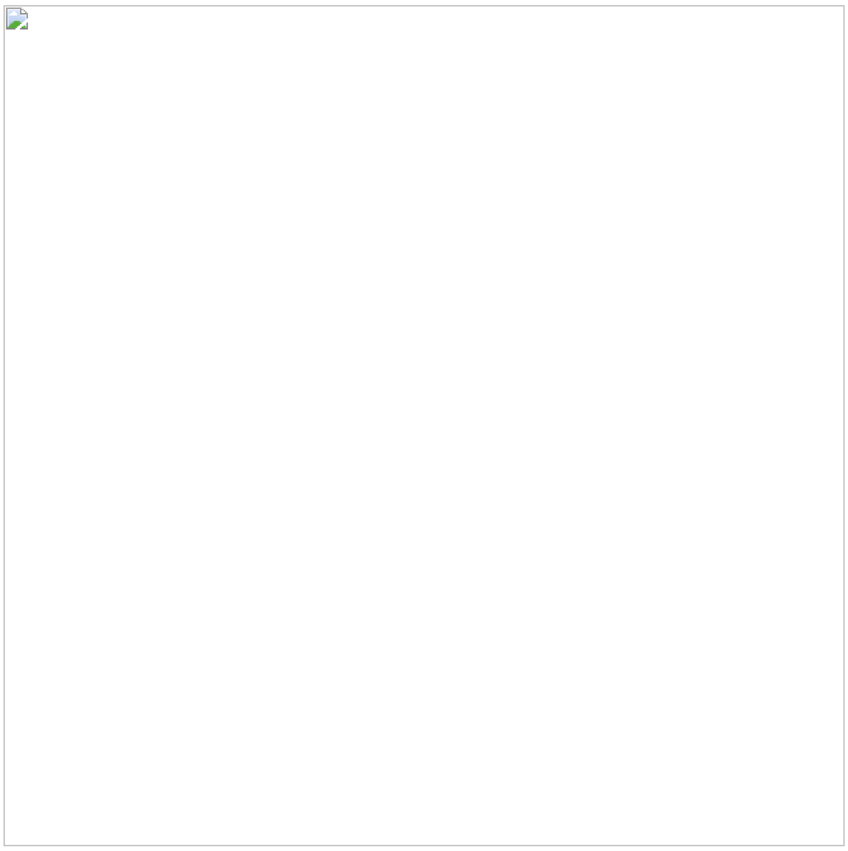
```python
class Solution:
    def inversionCount(self, arr, n):
        def merge(arr : List[int], low : int, mid : int, high : int) -> int:
            temp = []   # temporary array
            left = low  # starting index of left half of arr
            right = mid + 1 # starting index of right half of arr

            cnt = 0     # Modification 1: cnt variable to count the pairs

            # storing elements in the temporary array in a sorted manner
            while (left <= mid and right <= high):
                if (arr[left] <= arr[right]):
                    temp.append(arr[left])
                    left += 1
                else:
                    temp.append(arr[right])
                    cnt += (mid - left + 1)  # Modification 2
                    right += 1

            # if elements on the left half are still left
            while (left <= mid):
                temp.append(arr[left])
                left += 1

            # if elements on the right half are still left
            while (right <= high):
                temp.append(arr[right])
                right += 1

            # transfering all elements from temporary to arr
            for i in range(low, high + 1):
                arr[i] = temp[i - low]

            return cnt   # Modification 3

        def mergeSort(arr : List[int], low : int, high : int) -> int:
            cnt = 0
            if low >= high:
                return cnt
            mid = math.floor((low + high) / 2)
            cnt += mergeSort(arr, low, mid)    # left half
            cnt += mergeSort(arr, mid + 1, high)  # right half
            cnt += merge(arr, low, mid, high)  # merging sorted halves
            return cnt

        return mergeSort(arr, 0, n-1)
```

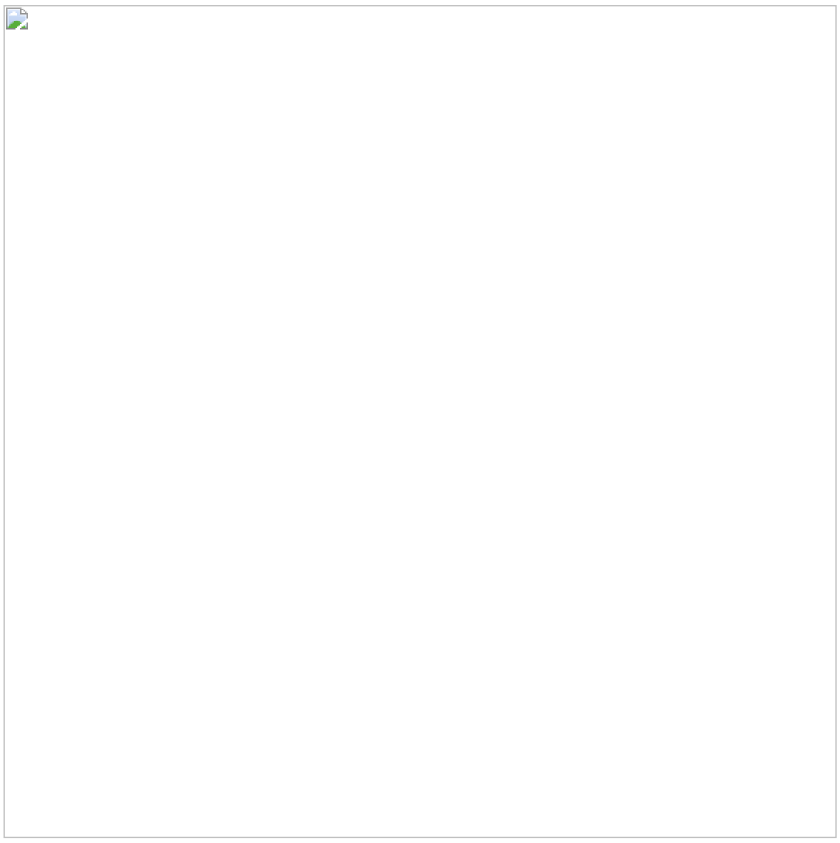
```cpp
// C++
// Optimal Solution
#include <bits/stdc++.h>
using namespace std;

int merge(vector<int> &arr, int low, int mid, int high) {
    vector<int> temp; // temporary array
    int left = low;      // starting index of left half of arr
    int right = mid + 1;   // starting index of right half of arr

    //Modification 1: cnt variable to count the pairs:
    int cnt = 0;

    //storing elements in the temporary array in a sorted manner//

    while (left <= mid && right <= high) {
        if (arr[left] <= arr[right]) {
            temp.push_back(arr[left]);
            left++;
        }
        else {
            temp.push_back(arr[right]);
            cnt += (mid - left + 1); //Modification 2
            right++;
        }
    }

    // if elements on the left half are still left //

    while (left <= mid) {
        temp.push_back(arr[left]);
        left++;
    }

    //  if elements on the right half are still left //
    while (right <= high) {
        temp.push_back(arr[right]);
        right++;
    }

    // transfering all elements from temporary to arr //
    for (int i = low; i <= high; i++) {
        arr[i] = temp[i - low];
    }

    return cnt; // Modification 3
}

int mergeSort(vector<int> &arr, int low, int high) {
    int cnt = 0;
    if (low >= high) return cnt;
    int mid = (low + high) / 2 ;
    cnt += mergeSort(arr, low, mid);  // left half
    cnt += mergeSort(arr, mid + 1, high); // right half
    cnt += merge(arr, low, mid, high);  // merging sorted halves
    return cnt;
}

int numberOfInversions(vector<int>&a, int n) {

    // Count the number of pairs:
    return mergeSort(a, 0, n - 1);
}
```

Count Reverse Pairs

Given an array of numbers, you need to return the count of reverse pairs. Reverse Pairs are those pairs where `i < j` and `arr[i] > 2 * arr[j]`

- Approach
    - Brute-force
        - Check for all pairs using two nested loops
        - Time Complexity: $O(n^2)$
        - Space Complexity: $O(1)$
    - Optimal





- Time Complexity: $O(2nlogn)$
- Space Complexity: $O(1)$

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

```
# Python3
# Optimal Solution
class Solution:
    def reversePairs(self, nums: List[int]) -> int:
        def merge(arr, low, mid, high):
            temp = []
            left = low
            right = mid + 1

            while left <= mid and right <= high:
                if arr[left] <= arr[right]:
                    temp.append(arr[left])
                    left += 1
                else:
                    temp.append(arr[right])
                    right += 1

            while left <= mid:
                temp.append(arr[left])
                left += 1
            while right <= high:
                temp.append(arr[right])
                right += 1

            for i in range(low, high + 1):
                arr[i] = temp[i - low]
        def mergeSort(arr, start, end):
            cnt = 0
            if start >= end: return cnt
            mid = (start + end) // 2
            cnt += mergeSort(arr, start, mid)
            cnt += mergeSort(arr, mid+1, end)
            cnt += countPairs(arr, start, mid, end)
            merge(arr, start, mid, end)
            return cnt

        def countPairs(arr, low, mid, high):
            cnt = 0
            right = mid + 1
            for i in range(low, mid + 1):
                while right <= high and arr[i] > 2 * arr[right]:
                    right += 1
                cnt += (right - (mid + 1))
            return cnt

        return mergeSort(nums, 0, len(nums)-1)
```

```
// C++
// Optimal Solution
class Solution {
public:
    void merge(vector<int>& nums, int low, int mid, int high) {
        vector<int> temp;
        int left = low;
        int right = mid + 1;

        while (left <= mid && right <= high) {
            if (nums[left] <= nums[right]) {
                temp.push_back(nums[left]);
                left++;
            }
            else {
                temp.push_back(nums[right]);
                right++;
            }
        }
        while (left <= mid) {
            temp.push_back(nums[left]);
            left++;
        }
        while (right <= high) {
            temp.push_back(nums[right]);
            right++;
        }

        for (int i = low; i <= high; i++) {
            nums[i] = temp[i-low];
        }
    }

    int mergeSort(vector<int>& nums, int start, int end) {
        int cnt = 0;
        if (start >= end) {return cnt;}
        int mid = start + ((end - start) / 2);
        cnt += mergeSort(nums, start, mid);
        cnt += mergeSort(nums, mid+1, end);
        cnt += countPairs(nums, start, mid, end);
        merge(nums, start, mid, end);
        return cnt;
    }

    int countPairs(vector<int>& nums, int low, int mid, int high) {
        int cnt = 0;
        int right = mid + 1;
        for (int i = low; i <= mid; i++) {
            while (right <= high && long(nums[i]) > 2*long(nums[right])) {
                right++;
            }
            cnt += (right - (mid + 1));
        }
        return cnt;
    }

    int reversePairs(vector<int>& nums) {
        return mergeSort(nums, 0, nums.size()-1);
    }
};
```

Maximum Product Subarray

- Approach
  - Brute-force
    - Generate all subarrays
    - Time Complexity: $O(n^3)$
    - Space Complexity: $O(1)$
  - Better
    - We don't need 3rd loop present in above solution
    - Time Complexity: $O(n^2)$
    - Space Complexity: $O(1)$
  - Optimal
    - There are 4 cases:
      - Only positive elements present in the array
        - Answer is product of all elements in the array
      - Even number of negative numbers in array and rest positive numbers
        - Answer is product of all elements in the array
      - Odd number of negative numbers in array and rest positive numbers
        - We need to skip 1 negative element
        - Example: arr = {2, 4, -6, 3, 5, -1, 7, -9, 8}
          - If we skip -6, the answer could be prefix product till -6 or suffix product till -6
          - If we skip -1, the answer could be prefix product till -1 or suffix product till -1
          - If we skip -9, the answer could be prefix product till -9 or suffix product till -9
        - Thus answer could lie in prefix or suffix of the array
      - Zeroes present along with positive and negative numbers
        - Example: arr = {2, 4, 0, -6, 3, 5, 0, -1, 7, 0, -9, 8}
        - When 0 is encountered, set `prefix_prod = 1` and `suffix_prod = 1`
    - Time Complexity: $O(n)$
    - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        maxi = -float("inf")
        for i in range(len(nums)):
            for j in range(i, len(nums)):
                prod = 1
                for k in range(i, j+1):
                    prod *= nums[k]
                maxi = max(maxi, prod)
        return maxi
```

```
# Python3
# Better Solution
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        maxi = -float("inf")
        for i in range(len(nums)):
            prod = nums[i]
            for j in range(i+1, len(nums)):
                maxi = max(maxi, prod)
                prod *= nums[j]
            maxi = max(maxi, prod)
        return maxi
```

```
# Python3
# Optimal Solution
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        n = len(nums)
        zero_present = False

        prefix_prod = 1
        max_prefix_prod = -float("inf")
        for i in range(n):
            if nums[i] == 0:
                zero_present = True
                prefix_prod = 1
                continue
            prefix_prod *= nums[i]
            max_prefix_prod = max(max_prefix_prod, prefix_prod)

        suffix_prod = 1
        max_suffix_prod = -float("inf")
        for i in range(n-1, -1, -1):
            if nums[i] == 0:
                zero_present = True
                suffix_prod = 1
                continue
            suffix_prod *= nums[i]
            max_suffix_prod = max(max_suffix_prod, suffix_prod)

        if zero_present:
            return max(0, max(max_prefix_prod, max_suffix_prod))
        return max(max_prefix_prod, max_suffix_prod)
```

```
// C++
// Optimal Solution
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int n = nums.size();
        bool zero_present = false;

        int prefix_prod = 1;
        int max_prefix_prod = INT_MIN;
        for (int i = 0; i < n; i++) {
            if (nums[i] == 0) {
                zero_present = true;
                prefix_prod = 1;
                continue;
            }
            prefix_prod *= nums[i];
            max_prefix_prod = max(max_prefix_prod, prefix_prod);
        }

        int suffix_prod = 1;
        int max_suffix_prod = INT_MIN;
        for (int i = n-1; i >= 0; i--) {
            if (nums[i] == 0) {
                zero_present = true;
                suffix_prod = 1;
                continue;
            }
            suffix_prod *= nums[i];
            max_suffix_prod = max(max_suffix_prod, suffix_prod);
        }

        if (zero_present) {
            return max(0, max(max_prefix_prod, max_suffix_prod));
        }
        return max(max_prefix_prod, max_suffix_prod);
    }
};
```

K Radius Subarray Averages

- Approach
    - Brute-force
        - For each element in the array, we check if we can compute the average for that point (i.e. if it is in boundary)

        - If the element is not in the boundary then we append -1 to our answer array
        - Else we loop from nums[i-k] to nums[i+k] and compute the average of all the elements in that subarray and append the average to our answer array
        - Time Complexity: $O(n * k)$
        - Space Complexity: $O(n)$ for storing answer else $O(1)$
    - Optimal (Using Prefix Sum)
        - Compute Prefix Sum
        - We know that we cannot compute the average for the first and last k elements in the given array as they are out of boundary and thus we append -1 to our answer array
        - For all valid nums[i], calculate average by `nums[i+k] - nums[i-k-1] // (2 * k + 1)` and append it to answer array
        - Time Complexity: $O(n)$
        - Space Complexity: $O(n)$ for storing answer else $O(1)$
    - Optimal (Using Sliding Window)
        - Initialize `sum = 0` and `vector<int> v(n, -1)` (n = nums.size() and all initial elements as -1)
        - Loop over all the elements in the array
        - Add current element to the sum variable
        - We get valid subarrays when `i ≥ 2 * k` so then we subtract the element which is out of boundary from the sum variable (i.e. `sum -= nums[i - 2*k - 1]`) and set `v[i-k] = sum / (2 * k + 1)`
        - Time Complexity: $O(n)$
        - Space Complexity: $O(n)$ for storing answer else $O(1)$

```python
# Python3
# Brute-force Solution
class Solution:
    def getAverages(self, nums: List[int], k: int) -> List[int]:
        n = len(nums)
        if n < k:   return [-1] * n
        if k == 0:  return nums

        ans = []
        for i in range(n):
            if i < k or i + k >= n:
                ans += -1,
            else:
                sum = 0
                for j in range(i-k, i+k+1):
                    sum += nums[j]
                sum //= (2 * k + 1)
                ans += sum,
        return ans
```

```python
# Python3
# Optimal Solution
# Prefix Sum
class Solution:
    def getAverages(self, nums: List[int], k: int) -> List[int]:
        # trivial cases
        n = len(nums)
        if n < k:   return [-1] * n
        if k == 0:  return nums

        # prefix sum
        for i in range(1, n):
            nums[i] += nums[i-1]

        ans = []
        i = 0
        while i < n:
            if i < k or i + k >= n:
                ans += -1,
            else:
                ans += (nums[i+k] - (nums[i-k-1] if (i-k-1) >= 0 else 0)) // (2*k + 1),
            i += 1
        return ans
```

```
// C++
// Optimal Solution
// Sliding Window
class Solution {
public:
    vector<int> getAverages(vector<int>& nums, int k) {
        int n = nums.size();
        vector<int> v(n, -1);
        if (n < k) {
            return v;
        }
        if (k == 0) {return nums;}

        long sum = 0;
        for (int i = 0; i < n; i++) {
            sum += nums[i];
            if (i >= 2 * k) {
                sum -= (i-2*k-1 >= 0) ? nums[i-2*k-1] : 0;
                v[i-k] = sum / (2 * k + 1);
            }
        }
        return v;
    }
};
```

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
```