1) [Longest Consecutive Sequence]

🧾 **Problem Statement:**

Given an **unsorted** array of integers nums, return the **length of the longest consecutive elements sequence**.

You must write an algorithm that runs in **O(n)** time.

🧠 **Key Observations:**

- Sorting takes O(n log n) → not allowed.
- We need an **O(n)** solution → Use unordered_set to allow constant-time lookup.

```
int longestConsecutive(vector<int>& nums) {
  unordered_set<int> s(nums.begin(), nums.end());
  int longest = 0;

  for (int num : s) {
    // Only start counting if it's the beginning of a sequence
    if (s.find(num - 1) == s.end()) {
      int currentNum = num;
      int count = 1;

      while (s.find(currentNum + 1) != s.end()) {
        currentNum++;
        count++;
      }

      longest = max(longest, count);
    }
  }

  return longest;
}
```

🧠 Notes:

- We only start a new sequence if ele - 1 is not in the set — to ensure we start from the first number of the sequence.
- unordered_set provides average O(1) time for insert() and find().
- Modifying ele inside the while loop is fine here because ele is a copy from the for(auto ele : st) loop.

📈 Time & Space Complexity:

Type	Complexity
Time	O(n)
Space	O(n)

📌 Why it's used:

Instead of writing:

```
unordered_set<int> s;
for (int num : nums) {
   s.insert(num);
}
```

You can do it in one line using constructor:

```
unordered_set<int> s(nums.begin(), nums.end());
```

2) [Largest subarray with 0 sum]()

Given an array of integers, find the length of the longest subarray with a sum equal to 0.

BRUTE FORCE (TLE):

```
int maxLen(vector<int>& arr) {
   int n = arr.size();
   int max_len = 0;
```

```
for (int i = 0; i < n; i++) {
    // Initialize the current sum for this starting point
    int curr_sum = 0;

    // Try all subarrays starting from 'i'
    for (int j = i; j < n; j++) {

      // Add the current element to curr_sum
      curr_sum += arr[j];

      // If curr_sum becomes 0, update max_len if required
      if (curr_sum == 0)
        max_len = max(max_len, j - i + 1);
    }
  }
  return max_len;
}
```

Time Complexity: O(n2)
Auxiliary Space: O(1)

OPTIMAL: (Prefix sum + Hashmap)

- ***If Si = Sj, then: Sj - Si = 0***

This means the subarray from i+1 to j has a sum of zero.

Illustration:

Consider the array arr = {5, 2, -1, 1, 4}. Calculate the prefix sums:

- ***S0 = 5***
- ***S1 = 5 + 2 = 7***
- ***S2 = 5 + 2 - 1 = 6***
- ***S3 = 5 + 2 - 1 + 1 = 7***

Here, S1 = S3 = 7. This equality tells us that the subarray from index 2 to 3 (subarray [-1, 1]) sums to zero.

```
int maxLen(vector<int>& nums) {
unordered_map<int, int> prefixIndex; // sum -> first index
int sum = 0, maxLen = 0;

for (int i = 0; i < nums.size(); i++) {
   sum += nums[i];

   if (sum == 0) {
      maxLen = i + 1; // // subarray from index 0 to i has sum 0
   }
   else if (prefixIndex.find(sum) != prefixIndex.end()) {
      // subarray from prefixIndex[sum] + 1 to i has sum 0
      maxLen = max(maxLen, i - prefixIndex[sum]);
   }
   else {
      // store first occurrence of this prefix sum
      prefixIndex[sum] = i;
   }
}

return maxLen;
}
```

Time Complexity: O(n), where n is the number of elements in the array.
Auxiliary Space: O(n), for storing the prefixSum in hashmap.

❓ Can "Largest Subarray with 0 Sum" be solved using Sliding Window?

🚫 **Short Answer:**

No, the standard sliding window technique does not work for this problem in general.

❓ **Why Not?**

Sliding Window only works when:

- All numbers are non-negative.
- Or you're working with monotonic properties.

But in this problem:

- The array can contain positive, negative, and zero values.
- Hence, increasing or shrinking the window doesn't guarantee we move toward or maintain a valid 0-sum.

3) Count Number of Pairs With Absolute Difference K

Given an integer array nums and an integer k, return *the number of pairs* (i, j) *where* i < j *such that* |nums[i] - nums[j]| == k.

The value of |x| is defined as:

- x if x >= 0.
- -x if x < 0.

➔ ☑ Version 1: Count All Pairs (Allow Duplicates)(Leetcode)

Use Case:

- You are allowed to count multiple occurrences of the same value.
- Pairs like (2, 5) and (5, 2) are both counted.
- Duplicate values increase count.

```
int countPairs(vector<int>& nums, int k) {
  int count = 0;
  unordered_map<int, int> mp;
```

```
  for (int i = 0; i < nums.size(); i++) {
    if (mp.find(nums[i] + k) != mp.end()) {
      count += mp[nums[i] + k];
    }
    if (mp.find(nums[i] - k) != mp.end()) {
      count += mp[nums[i] - k];
    }
    mp[nums[i]]++;
  }

  return count;
}
```

⏱ Time: O(n)

Space: O(n)

🧠 Example:

nums = [1, 5, 3, 3, 2], k = 2

We want all pairs (a, b) such that |a - b| == 2.

Step-by-step Dry Run:

🔁 Start with empty map: mp = {}

We iterate left to right:

➤ i = 0 → num = 1

- 1 + 2 = 3 not in map → skip
- 1 - 2 = -1 not in map → skip
- Insert 1 → mp = {1: 1}

➤ i = 1 → num = 5

- 5 + 2 = 7 ❌
- 5 - 2 = 3 ❌

- Insert → mp = {1: 1, 5: 1}

➤ i = 2 → num = 3

- 3 + 2 = 5 ✅ exists → count += 1
- 3 - 2 = 1 ✅ exists → count += 1
  ✅ Now we have:
  - (3,5)
  - (3,1)
- Insert → mp = {1: 1, 5: 1, 3: 1}

→ count = 2

➤ i = 3 → num = 3 again

- 3 + 2 = 5 ✅ exists → count += 1
- 3 - 2 = 1 ✅ exists → count += 1
  ✅ Again:
  - (3,5)
  - (3,1)
- Insert → mp = {1: 1, 5: 1, 3: 2}

→ count = 4

➤ i = 4 → num = 2

- 2 + 2 = 4 ❌
- 2 - 2 = 0 ❌
  Insert → mp = {1: 1, 5: 1, 3: 2, 2: 1}

→ count = 4 (unchanged)

✅ Total count = 4

☑ Version 2: Count Distinct Pairs Only (GFG)

Use Case:

- Each pair (a, b) should be counted only once.
- (2, 5) and (5, 2) are considered the same pair.
- If a pair exists multiple times, count it once only.

Given an integer array of size n and a non-negative integer k, count all distinct pairs with a difference equal to k, i.e., A[i] - A[j] = k.

```
int TotalPairs(vector<int> nums, int k) {
  int count = 0;
  unordered_set<int> st;
  set<pair<int, int>> uniquePairs; // since unordered set does not allow pairs

  for (int i = 0; i < nums.size(); i++) {
    if (st.find(nums[i] + k) != st.end()) {
      uniquePairs.insert({min(nums[i], nums[i] + k), max(nums[i], nums[i] + k)}); //since duplicates not allowed
    }

    if (st.find(nums[i] - k) != st.end()) {
      uniquePairs.insert({min(nums[i], nums[i] - k), max(nums[i], nums[i] - k)}); //since duplicates not allowed
    }

    st.insert(nums[i]);
  }

  return uniquePairs.size();
}
```

⏱ Time: O(n log n)

Space: O(n)

🔢 Input:(seen same as st)

nums = [1, 5, 3, 3, 2]

k = 2

We want to count all distinct pairs such that:

|a - b| == 2

🧠 Dry Run:

Initial:

- seen = {}
- uniquePairs = {}

➤ num = 1:

- 1+2 = 3 ❌
- 1−2 = -1 ❌
  → Add 1 to seen: {1}

➤ num = 5:

- 5+2 = 7 ❌
- 5−2 = 3 ❌
  → Add 5 to seen: {1, 5}

➤ num = 3:

- 3+2 = 5 ✅ → add (3,5)
- 3−2 = 1 ✅ → add (1,3)
  → Add 3 to seen: {1, 3, 5}
  → uniquePairs = {(1,3), (3,5)}

➤ num = 3 again:

- 3+2 = 5 ✅ but (3,5) already exists
- 3−2 = 1 ✅ but (1,3) already exists
  → No change

➤ num = 2:

- 2+2 = 4 ❌
- 2−2 = 0 ❌
  → Add 2 to seen: {1, 2, 3, 5}

✅ Final uniquePairs = {(1,3), (3,5)}

- So the answer is 2 (distinct pairs)

✅ Output: 2

✅ Summary

Feature	Version 1 (All Pairs)	Version 2 (Distinct Pairs)
Duplicate Pairs Counted?	✅ Yes	❌ No
Order Matters?	✅ Yes	❌ No
Data Structure Used	unordered_map	unordered_set + set
Time Complexity	O(n)	O(n log n)
Best For	Total number of valid pairs	Count of unique pairs