

# Q: Hashing

---

## 1. Problem Understanding

- Hashing is a technique to map data (key) → a fixed-size array (hash table) using a hash function.
  - Used to quickly:
    - Insert
    - Delete
    - Search
    - — usually in  $O(1)$  average time.
    - A hash function converts keys into indexes in an array.
- Why Hashing?
  - Array search =  $O(n)$
  - Binary Search =  $O(\log n)$
  - Hashing =  $O(1)$  average,  $O(n)$  worst-case (if many collisions)
- **Hash Function**
  - A formula that maps a key → index.
  - Example:
    - `int index = key % size;`
  - Should:
    - Distribute keys uniformly.
    - Avoid collisions.
    - Be deterministic (same key → same index).
- **Collisions**
  - Occur when two keys hash to the same index.
  - Handled by:
    - Chaining: Each array cell stores a `LinkedList` (or list) of keys with the same hash.
    - Open Addressing: Find the next available cell (Linear / Quadratic probing).
- **Common Hash-Based Data Structures (Java)**
  - `HashMap<K,V>` → key–value mapping
  - `HashSet` → stores unique elements (backed by `HashMap`)
  - `LinkedHashMap` → maintains insertion order
  - `TreeMap` → sorted order (Red-Black Tree, not hash-based)
- **DSA Use Cases of Hashing**
  - Count frequencies (numbers or characters)
  - Check duplicates
  - Track seen prefix sums / subarrays

- Map relations (like Roman numeral → value)
  - Fast lookup or existence check
  - String mappings (isomorphic, anagrams)
  - Unique collections (union, intersection)
- **Hash Arrays (Static Hashing)**
    - When elements are numeric and within range, use arrays as hash tables (common in CP / DSA problems).
    - Example:
      - `int[] hash = new int[1000000]; // 10^6 size`
      - `int[] arr = {1,4,2,4,2};`
      - `for (int x : arr) hash[x]++;`
      - `System.out.println(hash[4]); // prints 2`
  - **Memory Constraints in Java (⚠ Important for DSA)**
    - Inside main():
      - `int[] up to 10^6`
      - `boolean[] up to 10^7`
    - Outside main (static/global):
      - `int[] up to 10^7`
      - `boolean[] up to 10^8`
    - Reason:
      - Local variables are stored on stack (limited memory).
      - Static/global arrays are stored on heap (larger memory).
    - Exceeding limits causes:
      - `java.lang.OutOfMemoryError: Java heap space`
  - **Types of Hashing in DSA**
    - Direct Address Table: Use the key directly as index (when range is small).
    - Hash Table: Use hash function to map large keys to smaller indexes.
    - Double Hashing: Use two hash functions to minimize clustering in open addressing
  - **Implementation Snippets**
    - Frequency of characters:
 

```
* String s = "aabbcc"; * int[] freq = new int[26]; * for (char c : s.toCharArray()) freq[c - 'a']++; * System.out.println(freq['b' - 'a']); // freq of 'b'
```
    - Frequency using HashMap:
 

```
* HashMap<Integer, Integer> map = new HashMap<>(); * for (int x : arr) * map.put(x, map.getOrDefault(x, 0) + 1);
```
  - **Time & Space Complexity**
    - Average Case:
      - Insert → O(1)
      - Search → O(1)
      - Delete → O(1)
    - Worst Case: O(n) (many collisions)
    - Space: O(N)

- **Real-World Analogy**

- Like a library shelf system:
  - Hash function = shelf number
  - Book = key/value
  - You go directly to that shelf → fast retrieval.

- **Common Interview Problems (Hashing-Based)**

- Two Sum – LeetCode #1
  - → Store complement in HashMap.
- Subarray Sum Equals K – LeetCode #560
  - → Store prefix sums in HashMap.
- Longest Consecutive Sequence – LeetCode #128
  - → Use HashSet for fast existence checks.
- Group Anagrams – LeetCode #49
  - → HashMap with sorted string as key.
- Top K Frequent Elements – LeetCode #347
  - → HashMap + MinHeap.
- Valid Anagram – LeetCode #242
  - → Count frequency via array or HashMap.
- Intersection of Two Arrays – HashSet for membership check.

- **Best Practices for Hashing in DSA**

- For characters: use arrays (int[26] or int[256])
- For numbers: use HashMap or static arrays.
- Always offset negative numbers by  $+10^5$  or  $+10^6$  before indexing.
- Avoid large arrays if range is unknown → use HashMap.
- Clear hash structures between test cases in coding challenges.
- Use long as key for larger integers (e.g. prefix sums).
- Be aware of hash collisions — use .equals() and hashCode() properly for custom objects.

- **Internal Working of HashMap (Java Internals)**

- 1. Structure
  - Backed by an array of buckets (Node<K,V>[]).
  - Each bucket holds a linked list (or tree) of key-value pairs.
- 2. Node<K,V> Structure
  - static class Node<K,V> {
    - final int hash;
    - final K key;
    - V value;
    - Node<K,V> next;
  - }
- 3. hashCode()
  - Every key has a hashCode() (integer value).
  - HashMap refines this using bit manipulation:
  - `int hash = (key == null) ? 0 : (key.hashCode() ^ (key.hashCode() >>> 16));`

- → spreads hash bits to avoid collisions.
- 4. Index Calculation
  - Index in array = hash & (n - 1) (where n = capacity, always power of 2)
- 5. Collision Handling
  - Before Java 8: LinkedList chaining.
  - After Java 8: Converts to balanced tree (TreeNode) if chain length > 8.
- 6. Load Factor
  - Ratio of number of elements to array size (default = 0.75).
  - When load factor exceeds threshold → rehashing (array size doubles).
- 7. equals() and hashCode()
  - Both must be correctly overridden for user-defined keys.
  - Two keys are equal if:
  - key1.hashCode() == key2.hashCode() && key1.equals(key2)
- 8. Null Keys and Values
  - HashMap allows one null key and multiple null values.
  - Null key is stored in index 0.

- **Common Interview Questions on HashMap Internals**

- How does HashMap achieve O(1) complexity?
- What happens if two keys have same hashCode()?
- How does Java 8 improve collision handling?
- What is load factor and resizing?
- Why is capacity a power of 2?
- Why should hashCode() and equals() be consistent?

- **Optimization Tips**

- Use HashSet for unique element lookups.
- Use LinkedHashMap when insertion order matters.
- Use TreeMap when you need sorted keys.
- Avoid large primitive arrays if the key range is sparse.
- For competitive programming:
  - Prefer static arrays (faster than HashMap).
  - Use HashMap when input constraints are large or negative.

- **Quick Recap (Key Takeaways)**

- Hashing = fast lookup using key → index mapping.
  - Use arrays for small ranges, HashMap for generic data.
  - Memory matters:
    - Inside main:  $10^6$
    - Static/global:  $10^7$  (int),  $10^8$  (boolean)
  - Handle collisions smartly.
  - Understand hashCode() + equals() for interviews.
  - Practice problems: Two Sum, Subarray Sum, Anagrams.
-

# Q95: Longest Consecutive Sequence in an Array

---

## 1. Problem Understanding

- You are given an integer array `nums`.
  - You must find the length of the longest consecutive elements sequence, where the sequence numbers appear in any order.
  - → A “consecutive sequence” means numbers that come one after another with a difference of 1 (like 1,2,3,4).
  - Important: You only care about the length, not the sequence itself.
- 

## 2. Constraints

- $1 \leq \text{nums.length} \leq 10^5$
  - $-10^9 \leq \text{nums}[i] \leq 10^9$
- 

## 3. Edge Cases

- Empty array → return 0
  - Array with duplicates → count only unique numbers
  - Array already consecutive → return full length
  - Negative and mixed values handled normally
- 

## 4. Examples

```
Input: nums = [100, 4, 200, 1, 3, 2]
Output: 4
Explanation: Longest consecutive sequence is [1, 2, 3, 4].
```

## 5. Approaches

### Approach 1: Sorting-Based

#### Idea:

- Sort the array and then count consecutive elements. Skip duplicates and reset when the sequence breaks.

#### Steps:

- Sort the array.

- Use variables currLen and maxLen to track streaks.
- Compare consecutive elements and update counts.

#### Java Code:

```
public int longestConsecutive(int[] nums) {
    if (nums.length == 0) return 0;
    Arrays.sort(nums);
    int maxLen = 1, currLen = 1;

    for (int i = 1; i < nums.length; i++) {
        if (nums[i] == nums[i - 1]) continue;
        if (nums[i] == nums[i - 1] + 1) currLen++;
        else currLen = 1;
        maxLen = Math.max(maxLen, currLen);
    }
    return maxLen;
}
```

#### Complexity (Time & Space):

- ⏳ Time Complexity
  - Sorting:  $O(n \log n)$
  - Single traversal:  $O(n)$
  - Total:  $O(n \log n)$
- 🗄 Space Complexity
  - $O(1)$  (ignoring sort overhead)

#### Approach 2: Using HashSet (Optimal)

#### Idea:

- Use a HashSet to check existence of consecutive numbers quickly.
- Start counting only when the current number is the start of a sequence (i.e.,  $\text{num} - 1$  is not present).

#### Steps:

- Add all numbers to a HashSet.
- For each number, if  $\text{num} - 1$  doesn't exist, start a new sequence.
- Keep counting  $\text{num} + 1, \text{num} + 2, \dots$  until break.
- Track the maximum streak length.

#### Java Code:

```
public int longestConsecutive(int[] nums) {
    HashSet<Integer> set = new HashSet<>();
    for (int num : nums) set.add(num);

    int maxLen = 0;
```

```

for (int num : set) {
    if (!set.contains(num - 1)) { // sequence start
        int curr = num;
        int len = 1;

        while (set.contains(curr + 1)) {
            curr++;
            len++;
        }
        maxLen = Math.max(maxLen, len);
    }
}
return maxLen;
}

```

### Complexity (Time & Space):

- ⌚ Time Complexity
  - Insert all elements: O(n)
  - Traverse and count sequences: O(n) (each element checked once)
  - Total: O(n)
- 💾 Space Complexity
  - O(n) for HashSet storage

### Approach 3: Using HashMap (Alternate)

#### Idea:

- Use a HashMap to dynamically merge sequences.
- Each number connects to the length of its current sequence, and merging happens when neighboring sequences exist.

#### Steps:

- For each element, get lengths of left and right neighboring sequences.
- New sequence length = left + right + 1.
- Update boundaries (num - left and num + right) with new length.
- Keep track of the longest sequence.

#### Java Code:

```

public int longestConsecutive(int[] nums) {
    HashMap<Integer, Integer> map = new HashMap<>();
    int maxlen = 0;

    for (int num : nums) {
        if (map.containsKey(num)) continue;

        int left = map.getOrDefault(num - 1, 0);

```

```

        int right = map.getOrDefault(num + 1, 0);
        int len = left + right + 1;

        map.put(num, len);
        map.put(num - left, len);
        map.put(num + right, len);

        maxLen = Math.max(maxLen, len);
    }
    return maxLen;
}

```

### Complexity (Time & Space):

- ⌚ Time Complexity
    - Each element processed once:  $O(n)$
    - HashMap operations:  $O(1)$  average
    - Total:  $O(n)$
  - 💾 Space Complexity
    - $O(n)$  for HashMap
- 

## 6. Justification / Proof of Optimality

- Sorting is simpler but slower ( $O(n \log n)$ ).
  - HashSet is optimal and easy to reason about.
  - HashMap approach is powerful for merging intervals (advanced variant).
- 

## 7. Variants / Follow-Ups

- Longest consecutive subsequence in a sorted array.
  - Longest sequence with given difference  $k$ .
  - Used in problems like “Longest Band” or “Union of consecutive ranges.”
- 

## 8. Tips & Observations

- Always check  $num - 1$  to find the start of a sequence.
  - HashSet avoids unnecessary duplicate counting.
  - The logic applies for negative, positive, and unordered values.
  - This is a great example of how to convert  $O(n \log n)$  sorting logic into  $O(n)$  using hashing.
- 

# Q96: Longest Subarray with Sum K

---

## 1. Problem Understanding

- You are given an array `nums` of integers and a target sum `k`.
  - You need to find the length of the longest contiguous subarray whose sum equals `k`.
  - If no such subarray exists, return 0.
- 

## 2. Constraints

- $1 \leq n \leq 10^5$
  - $-10^5 \leq \text{nums}[i] \leq 10^5$
  - $-10^9 \leq k \leq 10^9$
- 

## 3. Edge Cases

- All numbers are positive  $\rightarrow$  sliding window approach works.
  - Array contains negative numbers  $\rightarrow$  need prefix sum + HashMap.
  - $k = 0$  (check for subarrays summing to zero).
  - No subarray equals `k`  $\rightarrow$  return 0.
- 

## 4. Examples

```
Input: nums = [10, 5, 2, 7, 1, 9], k = 15
Output: 4
Explanation: [5, 2, 7, 1] has sum = 15.
```

---

## 5. Approaches

### Approach 1: Brute Force ( $O(n^2)$ )

#### Idea:

- Try every possible subarray and compute its sum.
- Keep track of the longest one equal to `k`.

#### Steps:

- Use two loops — outer for start index, inner for end index.
- Compute sum for each subarray.
- If sum == `k`  $\rightarrow$  update longest length.

#### Java Code:

```
int longestSubarraySumK(int[] nums, int k) {
    int n = nums.length;
    int maxLen = 0;
    for (int i = 0; i < n; i++) {
        int sum = 0;
```

```

        for (int j = i; j < n; j++) {
            sum += nums[j];
            if (sum == k)
                maxLen = Math.max(maxLen, j - i + 1);
        }
    }
    return maxLen;
}

```

### Complexity (Time & Space):

- ⌚ Time Complexity
  - $O(n^2)$  → nested loops
  - For large  $n$  ( $10^5$ ), not efficient
- 💾 Space Complexity
  - $O(1)$

### Approach 2: Prefix Sum + HashMap (Optimized $O(n)$ )

#### Idea:

- Use a prefix sum to keep track of cumulative sums, and store the first occurrence of each prefix in a HashMap.
- If  $\text{prefixSum} - k$  exists in map → a subarray with sum  $k$  exists from that index to current.

#### Steps:

- Initialize  $\text{sum} = 0$ ,  $\text{maxLen} = 0$ , and a  $\text{HashMap<Integer, Integer>} \text{prefixMap}$ .
- For each element:
  - Add it to  $\text{sum}$ .
  - If  $\text{sum} == k \rightarrow$  update  $\text{maxLen}$ .
  - If  $\text{sum} - k$  exists in map → subarray found, update  $\text{maxLen}$ .
  - If  $\text{sum}$  not in map → store it with current index (only first occurrence).
- Return  $\text{maxLen}$ .

#### Java Code:

```

int longestSubarraySumK(int[] nums, int k) {
    HashMap<Integer, Integer> map = new HashMap<>();
    int sum = 0, maxLen = 0;

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

        if (sum == k) maxLen = i + 1;

        if (map.containsKey(sum - k)) {
            maxLen = Math.max(maxLen, i - map.get(sum - k));
        }
    }
}

```

```

        if (!map.containsKey(sum)) {
            map.put(sum, i);
        }
    }

    return maxLen;
}

```

### Complexity (Time & Space):

- ⌚ Time Complexity
  - $O(n)$  → single pass with  $O(1)$  HashMap lookups
- 💾 Space Complexity
  - $O(n)$  → HashMap storing prefix sums

### Approach 3: Sliding Window (Only for Non-Negative Numbers)

#### Idea:

- When all elements are non-negative, we can use a sliding window to expand/shrink the subarray.

#### Steps:

- Maintain two pointers l and r, and a running sum.
- Expand r to increase sum.
- If sum > k → move l forward.
- If sum == k → update maxLen.

#### Java Code:

```

int longestSubarraySumK(int[] nums, int k) {
    int left = 0, right = 0, sum = 0, maxLen = 0;
    int n = nums.length;

    while (right < n) {
        sum += nums[right];

        while (sum > k) {
            sum -= nums[left];
            left++;
        }

        if (sum == k) {
            maxLen = Math.max(maxLen, right - left + 1);
        }

        right++;
    }

    return maxLen;
}

```

## Complexity (Time & Space):

- ⌚ Time Complexity
    - $O(n)$  → each element added/removed once
  - 💾 Space Complexity
    - $O(1)$
- 

## 6. Justification / Proof of Optimality

- The HashMap + Prefix Sum approach is the most optimal because:
    - It handles both positive and negative numbers.
    - It gives  $O(n)$  time complexity.
- 

## 7. Variants / Follow-Ups

- Count of subarrays with sum K (instead of longest length).
  - Subarray sum divisible by K
  - Longest subarray with sum  $\leq K$
- 

## 8. Tips & Observations

- Always store first occurrence of prefix sum — it gives longest subarray.
  - Resetting map or overwriting sums can shorten valid subarrays.
  - For strictly positive arrays → sliding window is even simpler and faster.
- 

# Q97: Count subarrays with given sum

---

## 1. Problem Understanding

- You are given an integer array `nums` and an integer `k`.
  - You need to count all subarrays whose sum equals `k`.
  - Subarrays are contiguous sequences within the array.
- 

## 2. Constraints

- $1 \leq \text{nums.length} \leq 10^5 \rightarrow$  implies  $O(n^2)$  brute force may TLE, we need  $O(n)$ .
  - $-1000 \leq \text{nums}[i] \leq 1000 \rightarrow$  elements can be negative.
  - $-10^7 \leq k \leq 10^7$ .
- 

## 3. Edge Cases

- Array contains negative numbers → prefix sum with hashmap works.
  - Array has all zeros,  $k = 0$  → multiple subarrays sum to 0.
  - Single element array equal to  $k$ .
- 

## 4. Examples

```
Input: [1, 1, 1], k = 2 → Output: 2 → [1,1], [1,1]
```

```
Input: [1, 2, 3], k = 3 → Output: 2 → [1,2], [3]
```

```
Input: [3, 1, 2, 4], k = 6 → Output: 1 → [2,4]
```

---

## 5. Approaches

### Approach 1: Brute Force

#### Idea:

- Check all possible subarrays and sum them.

#### Steps:

- Initialize count = 0
- Loop i from 0 to  $n-1$
- Loop j from i to  $n-1$
- Sum  $\text{nums}[i..j]$ , if sum ==  $k \rightarrow \text{count}++$

#### Java Code:

```
public int subarraySum(int[] nums, int k) {  
    int count = 0;  
    for (int i = 0; i < nums.length; i++) {  
        int sum = 0;  
        for (int j = i; j < nums.length; j++) {  
            sum += nums[j];  
            if (sum == k) count++;  
        }  
    }  
    return count;  
}
```

---

#### Complexity (Time & Space):

- ⏳ Time Complexity:
  - $O(n^2)$
- ⌂ Space Complexity:

- O(1)

## Approach 2: Prefix Sum + HashMap (Optimized)

### Idea:

- Use a prefix sum and store counts of prefix sums in a hashmap.
- For each new prefix sum currSum, check if  $(currSum - k)$  exists in the map → add its frequency to result.

### Steps:

- Initialize map = {0:1}, sum = 0, count = 0
- Loop over each element in nums:
- $sum += nums[i]$
- If  $(sum - k)$  exists in map → count += map.get(sum - k)
- $map[sum] = map.getOrDefault(sum, 0) + 1$

### Java Code:

```
import java.util.HashMap;

public class Solution {
    public int subarraySum(int[] nums, int k) {
        HashMap<Integer, Integer> map = new HashMap<>();
        map.put(0, 1); // base case
        int sum = 0, count = 0;
        for (int num : nums) {
            sum += num;
            if (map.containsKey(sum - k)) {
                count += map.get(sum - k);
            }
            map.put(sum, map.getOrDefault(sum, 0) + 1);
        }
        return count;
    }
}
```

### Complexity (Time & Space):

- ⏳ Time Complexity:
  - O(n) → single pass over array
- 📁 Space Complexity:
  - O(n) → hashmap storing prefix sums

## 6. Justification / Proof of Optimality

- The hashmap keeps track of how many times a prefix sum occurred.
- $sum - k$  exists → there exists a previous prefix sum that forms a subarray summing to k.

## 7. Variants / Follow-Ups

- Count subarrays with sum  $\leq k \rightarrow$  need sliding window for positive numbers.
  - Longest subarray with sum = k  $\rightarrow$  similar prefix sum + hashmap, store first index.
- 

## 8. Tips & Observations

- Always initialize map.put(0,1)  $\rightarrow$  handles subarrays starting from index 0.
  - Works for negative numbers too.
  - For longest subarray, store first occurrence of each prefix sum.
- 

# Q98: Count subarrays with given xor K

---

## 1. Problem Understanding

- You are given an integer array nums and an integer k.
  - You need to count all subarrays whose XOR of elements equals k.
  - XOR (^) is bitwise exclusive OR.
- 

## 2. Constraints

- $1 \leq \text{nums.length} \leq 10^5 \rightarrow O(n^2)$  brute force may TLE, need  $O(n)$ .
  - $1 \leq \text{nums}[i] \leq 10^9 \rightarrow$  elements are positive integers.
  - $1 \leq k \leq 10^9$ .
- 

## 3. Edge Cases

- Single element equal to k  $\rightarrow$  counts as one subarray.
  - Multiple subarrays with same XOR.
  - Large numbers  $\rightarrow$  ensure XOR calculations are correct (int is fine in Java).
- 

## 4. Examples

```
Input: [4, 2, 2, 6, 4], k = 6 → Output: 4 → [4,2], [4,2,2,6,4], [2,2,6], [6]
```

```
Input: [5, 6, 7, 8, 9], k = 5 → Output: 2 → [5], [5,6,7,8,9]
```

```
Input: [5, 2, 9], k = 7 → Output: 0
```

## 5. Approaches

## Approach 1: Brute Force

### Idea:

- Check XOR for all subarrays.

### Steps:

- Initialize count = 0
- Loop i from 0 to n-1
- Loop j from i to n-1 → xor = XOR(nums[i..j])
- If xor == k → count++

### Java Code:

```
public int subarrayXor(int[] nums, int k) {  
    int count = 0;  
    for (int i = 0; i < nums.length; i++) {  
        int xor = 0;  
        for (int j = i; j < nums.length; j++) {  
            xor ^= nums[j];  
            if (xor == k) count++;  
        }  
    }  
    return count;  
}
```

### Complexity (Time & Space):

- ⏳ Time Complexity:
  - $O(n^2)$  → TLE for large n
- 💫 Space Complexity:
  - $O(1)$

## Approach 2: Prefix XOR + HashMap (Optimized)

### Idea:

- Let prefixXOR[i] = XOR of all elements from 0 to i.
- For subarray nums[l..i], XOR = prefixXOR[i] ^ prefixXOR[l-1]
- If prefixXOR[i] ^ k exists in map → there exists a subarray ending at i with XOR k.

### Steps:

- Initialize map = {0:1}, xor = 0, count = 0
- Loop over each element in nums:
  - $xor \wedge= nums[i]$
  - If  $(xor \wedge k)$  exists in map → count += map.get(xor  $\wedge k$ )
  - map[xor] = map.getOrDefault(xor, 0) + 1

## Java Code:

```
import java.util.HashMap;

public class Solution {
    public int subarrayXor(int[] nums, int k) {
        HashMap<Integer, Integer> map = new HashMap<>();
        map.put(0, 1); // base case
        int xor = 0, count = 0;
        for (int num : nums) {
            xor ^= num;
            count += map.getOrDefault(xor ^ k, 0);
            map.put(xor, map.getOrDefault(xor, 0) + 1);
        }
        return count;
    }
}
```

## Complexity (Time & Space):

- ⌚ Time Complexity:
    - $O(n)$  → single pass over array
  - 💾 Space Complexity:
    - $O(n)$  → hashmap storing prefix XOR counts
- 

## 6. Justification / Proof of Optimality

- The hashmap stores frequency of prefix XORs.
  - $xor \wedge k$  in map → there exists a previous prefix XOR that will form a subarray with  $XOR = k$ .
  - Works for large positive integers.
- 

## 7. Variants / Follow-Ups

- Count subarrays with  $XOR \leq k$  → needs trie-based approach.
  - Find longest subarray with  $XOR = k$  → use first occurrence of prefix XOR in map.
- 

## 8. Tips & Observations

- XOR is reversible:  $a \wedge b = c \rightarrow a = b \wedge c$
  - Initialize  $map.put(0, 1)$  → handles subarrays starting from index 0.
  - Similar template to prefix sum with hashmap, just replace  $+$  with  $\wedge$ .
-