
WEEK 1 – DAY 2

Prefix Sum, Hashing, Analytical Thinking

Sony evaluates here:

- Ability to convert **$O(n^2)$ → $O(n)$**
- Use of **prefix math + hash maps**
- Translating DSA logic into **real product features**

---

1. Prefix Sum – Core Concept

What You Should Say

“Prefix sum is a technique where we preprocess cumulative sums so that range queries and subarray calculations become constant time.”

Prefix Sum Definition

For array arr:

prefix[i] = arr[0] + arr[1] + ... + arr[i]

Range Sum Formula

sum(l, r) = prefix[r] - prefix[l-1]

Interview Warm-Up Questions

1. Why is prefix sum useful?
2. What is the preprocessing cost?
3. Can prefix sum handle negative numbers?
4. When is prefix sum **not sufficient**?
5. How much memory does it consume?

Spoken (Opening)

“Prefix sum is a preprocessing technique where we store cumulative sums so that range sum and subarray calculations can be answered in constant time.”

---

1. Why is prefix sum useful?

Spoken

"Prefix sum is useful because it converts repeated range sum queries from O(n) time to O(1) time after a one-time preprocessing step."

Code Example

```
arr = [2, 4, 6, 8]
prefix = [0] * len(arr)
prefix[0] = arr[0]

for i in range(1, len(arr)):
    prefix[i] = prefix[i-1] + arr[i]

# sum from index 1 to 3
l, r = 1, 3
range_sum = prefix[r] - prefix[l-1]
```

---

2. What is the preprocessing cost?

Spoken

"The preprocessing cost is O(n) time because we traverse the array once to build the prefix sum array."

Code

```
def build_prefix(arr):
    prefix = [0] * len(arr)
    prefix[0] = arr[0]
    for i in range(1, len(arr)):
        prefix[i] = prefix[i-1] + arr[i]
    return prefix
```

---

3. Can prefix sum handle negative numbers?

Spoken

"Yes, prefix sum works correctly with negative numbers because it relies on addition and subtraction, not ordering."

Code

```
arr = [3, -2, 5, -1]
```

```
prefix = [arr[0]]


for i in range(1, len(arr)):
    prefix.append(prefix[-1] + arr[i])


# sum from index 0 to 2
print(prefix[2])  # Output: 6
```

---

4. When is prefix sum not sufficient?

Spoken

“Prefix sum is not sufficient when the array is frequently updated, because each update would require recomputing the prefix sums. In such cases, Fenwick Trees or Segment Trees are better.”

Code (Update Problem Example)

```
# After updating arr[1]
arr[1] = 10


# prefix must be recomputed
for i in range(1, len(arr)):
    prefix[i] = prefix[i-1] + arr[i]
```

---

5. How much memory does prefix sum consume?

Spoken

“Prefix sum uses O(n) extra memory to store cumulative sums.”

Code

```
# Extra array of size n
prefix = [0] * len(arr)
```

Sony-Level One-Line Wrap-Up (Memorize)

“Prefix sum trades O(n) preprocessing time and memory for O(1) range query performance, making it ideal for static arrays with frequent queries.”

---

2. Problem 1: Subarray Sum = K

Interview Question

Given an array of integers and an integer k, return the total number of continuous subarrays whose sum equals k.

Brute Force (Explain First)

- Check all subarrays
- **O(n²)** ❌

Spoken (Problem Restatement)

“Given an integer array and a value k, we need to count how many continuous subarrays sum exactly to k.”

Brute Force (Explain First)

Spoken

“The brute-force approach is to check all possible subarrays and compute their sums, which takes O(n²) time and is not efficient for large inputs.”

Optimized Approach: Prefix Sum + Hash Map

Key Insight (Say This Clearly)

“If two prefix sums differ by k, then the elements between those indices form a subarray whose sum is k.”

Formally:

“If prefix[j] minus prefix[i] equals k, then subarray i+1 to j sums to k.”

Core Code (Reference Solution)

Python

```python
def subarraySum(nums, k):
    count = 0
    prefix_sum = 0
    freq = {0: 1}

    for num in nums:
        prefix_sum += num
        if prefix_sum - k in freq:
            count += freq[prefix_sum - k]
        freq[prefix_sum] = freq.get(prefix_sum, 0) + 1

    return count
```

JavaScript

```
function subarraySum(nums, k) {
    let count = 0;
    let prefixSum = 0;
    let freq = new Map();
    freq.set(0, 1);

    for (let num of nums) {
        prefixSum += num;
        if (freq.has(prefixSum - k)) {
            count += freq.get(prefixSum - k);
        }
        freq.set(prefixSum, (freq.get(prefixSum) || 0) + 1);
    }
    return count;
}
```

Complexity

- **Time:** O(n)
- **Space:** O(n)

Sony Follow-Ups

1. Why does {0:1} exist?
2. Can sliding window work here?
3. What if all numbers are positive?

4. How to return indices instead of count?
5. How would this work for real-time streaming data?

Sony Follow-Up Questions (Spoken + Code)

---

1. Why does {0:1} exist?

Spoken

“The entry {0:1} handles subarrays that start at index 0.
If the prefix sum itself equals k, then prefixSum − k becomes zero, and this case must be counted.”

Code Illustration

```
# nums = [3, 4], k = 7

# prefixSum = 7

# prefixSum - k = 0 → found in freq
```

---

2. Can sliding window work here?

Spoken

“No, sliding window does not work in the general case because the array may contain negative numbers, which break the window-shrinking logic.”

---

3. What if all numbers are positive?

Spoken

“If all numbers are positive, then sliding window becomes valid because expanding the window increases the sum and shrinking it decreases the sum.”

Sliding Window Code (Positive Only)

```
def subarraySumPositive(nums, k):

    left = 0

    curr_sum = 0

    count = 0


    for right in range(len(nums)):

        curr_sum += nums[right]
        while curr_sum > k:
            curr_sum -= nums[left]
            left += 1

        if curr_sum == k:
            count += 1

    return count
```

---

4. How do you return indices instead of count?

Spoken

“Instead of storing frequency, I store indices for each prefix sum. Whenever a match is found, I extract the start and end indices.”

Code

```
def subarraySumIndices(nums, k):
    prefix_sum = 0
    index_map = {0: [-1]}
    result = []

    for i, num in enumerate(nums):
        prefix_sum += num
        if prefix_sum - k in index_map:
            for start in index_map[prefix_sum - k]:
                result.append((start + 1, i))
        index_map.setdefault(prefix_sum, []).append(i)

    return result
```

---

5. How would this work for real-time streaming data?

Spoken

“This approach works naturally for streaming data because each element is processed once and we only maintain a running prefix sum and a frequency map.”

Streaming Code

```
class StreamingSubarraySum:
    def __init__(self, k):
        self.k = k
        self.prefix_sum = 0
        self.freq = {0: 1}
        self.count = 0

    def add(self, num):
        self.prefix_sum += num
        self.count += self.freq.get(self.prefix_sum - self.k, 0)
        self.freq[self.prefix_sum] = self.freq.get(self.prefix_sum, 0) + 1
        return self.count
```

Complexity (Say This)

"The algorithm runs in O(n) time and uses O(n) extra space due to the hash map."

Sony-Level One-Line Wrap-Up (Memorize)

"Using prefix sum with a hash map allows us to count subarrays summing to k in linear time, even when negative numbers are present."

---

3. Problem 2: Range Sum Query

Interview Question

Given an array and multiple queries [l, r], return sum of elements between l and r.

Spoken (Problem Restatement)

"Given an array and multiple queries of the form [l, r], we need to return the sum of elements between indices l and r efficiently."

Optimal Approach: Prefix Sum Preprocessing

Spoken (Core Explanation)

"I preprocess the array into a prefix sum array where each index stores the cumulative sum up to that point. This allows every range sum query to be answered in constant time using subtraction."

Code Implementation

Python

```
class RangeSum:
    def __init__(self, nums):
        self.prefix = [0] * (len(nums) + 1)
        for i in range(len(nums)):
            self.prefix[i + 1] = self.prefix[i] + nums[i]

    def query(self, l, r):
        return self.prefix[r + 1] - self.prefix[l]
```

JavaScript

```
class RangeSum {
    constructor(nums) {
        this.prefix = new Array(nums.length + 1).fill(0);
        for (let i = 0; i < nums.length; i++) {
            this.prefix[i + 1] = this.prefix[i] + nums[i];
        }
    }

    query(l, r) {
        return this.prefix[r + 1] - this.prefix[l];
    }
}
```

Complexity (Say This Clearly)

"Preprocessing takes O(n) time, and each query runs in O(1) time."

Sony Follow-Ups

1. What if the array updates frequently?
2. How would you handle millions of queries?
3. When would you use Fenwick Tree?
4. Can this be parallelized?
5. What about memory constraints?

Sony Follow-Up Questions (Spoken + Code/Concept)

---

1. What if the array updates frequently?

Spoken

“Prefix sum is inefficient for frequent updates because any change requires recomputing the array. In that case, I would use a Fenwick Tree or Segment Tree.”

---

2. How would you handle millions of queries?

Spoken

“Prefix sum is ideal for this scenario because preprocessing is done once, and each query is constant time, making it highly scalable.”

---

3. When would you use Fenwick Tree?

Spoken

“I use a Fenwick Tree when both range queries and point updates need to be handled efficiently in logarithmic time.”

Fenwick Tree (Python)

```
class FenwickTree:
    def __init__(self, n):
        self.bit = [0] * (n + 1)


    def update(self, i, delta):
        i += 1
        while i < len(self.bit):
            self.bit[i] += delta
            i += i & -i
    def query(self, i):
        s = 0
        i += 1
        while i > 0:
            s += self.bit[i]
            i -= i & -i
        return s

    def range_sum(self, l, r):
        return self.query(r) - self.query(l - 1)
```

---

4. Can this be parallelized?

Spoken

“Prefix sum construction can be parallelized using scan algorithms, but query operations are already constant time and do not benefit from parallelism.”

---

5. What about memory constraints?

Spoken

“Prefix sum requires O(n) extra memory. If memory is constrained, I would compute sums on the fly or use block-based decomposition.”

---

4. Sony-Level Variations (Very Important)

Expect **one or more**:

1. Count subarrays with sum divisible by k
2. Longest subarray with sum k
3. Minimum size subarray ≥ k
4. 2D prefix sum (matrix sum query)
5. Prefix XOR instead of sum

1️⃣ Count Subarrays with Sum Divisible by K

Spoken

"If two prefix sums have the same remainder when divided by k, then the subarray between them has a sum divisible by k. I count how many times each remainder occurs using a hash map."

Python

```python
def subarraysDivByK(nums, k):
    count = 0
    prefix = 0
    freq = {0: 1}

    for num in nums:
        prefix = (prefix + num) % k
        count += freq.get(prefix, 0)
        freq[prefix] = freq.get(prefix, 0) + 1

    return count
```

JavaScript

```javascript
function subarraysDivByK(nums, k) {
    let count = 0;
    let prefix = 0;
    let freq = new Map();
    freq.set(0, 1);

    for (let num of nums) {
        prefix = ((prefix + num) % k + k) % k;
        count += freq.get(prefix) || 0;
        freq.set(prefix, (freq.get(prefix) || 0) + 1);
    }
    return count;
}
```

---

2 Longest Subarray with Sum = K

Spoken

“I store the first occurrence of each prefix sum. When I see prefixSum minus k again, I compute the distance to get the longest subarray.”

Python

```
def longestSubarray(nums, k):
    prefix = 0
    index_map = {0: -1}
    max_len = 0

    for i, num in enumerate(nums):
        prefix += num
        if prefix - k in index_map:
            max_len = max(max_len, i - index_map[prefix - k])
        if prefix not in index_map:
            index_map[prefix] = i

    return max_len
```

JavaScript

```
function longestSubarray(nums, k) {
    let prefix = 0;
    let map = new Map();
    map.set(0, -1);
    let maxLen = 0;

    for (let i = 0; i < nums.length; i++) {
        prefix += nums[i];
        if (map.has(prefix - k)) {
            maxLen = Math.max(maxLen, i - map.get(prefix - k));
        }
        if (!map.has(prefix)) {
            map.set(prefix, i);
        }
    }
    return maxLen;
}
```

3️⃣ Minimum Size Subarray ≥ K

(Positive numbers only)

Spoken

“Since all values are positive, I use a sliding window. I expand the window until the sum is at least k, then shrink it to find the minimum length.”

Python

```
def minSubArrayLen(k, nums):
    left = 0
    curr_sum = 0
    ans = float('inf')

    for right in range(len(nums)):
        curr_sum += nums[right]
        while curr_sum >= k:
            ans = min(ans, right - left + 1)
            curr_sum -= nums[left]
            left += 1

    return 0 if ans == float('inf') else ans
```

JavaScript

```
function minSubArrayLen(k, nums) {
    let left = 0;
    let sum = 0;
    let ans = Infinity;
```

```
  for (let right = 0; right < nums.length; right++) {
    sum += nums[right];
    while (sum >= k) {
      ans = Math.min(ans, right - left + 1);
      sum -= nums[left++];
    }
  }
  return ans === Infinity ? 0 : ans;
}
```

4️⃣ 2D Prefix Sum (Matrix Range Sum Query)

Spoken

“I preprocess a 2D prefix sum matrix so that any submatrix sum can be computed in constant time using inclusion–exclusion.”

Python

```
def buildPrefix2D(matrix):
  m, n = len(matrix), len(matrix[0])
  prefix = [[0]*(n+1) for _ in range(m+1)]

  for i in range(m):
    for j in range(n):
      prefix[i+1][j+1] = (
        matrix[i][j]
        + prefix[i][j+1]
        + prefix[i+1][j]
        - prefix[i][j]
      )
  return prefix
```

```
def query(prefix, r1, c1, r2, c2):
    return (
        prefix[r2+1][c2+1]
        - prefix[r1][c2+1]
        - prefix[r2+1][c1]
        + prefix[r1][c1]
    )
```

JavaScript

```
function buildPrefix2D(matrix) {
    const m = matrix.length, n = matrix[0].length;
    const prefix = Array.from({ length: m + 1 }, () => Array(n + 1).fill(0));

    for (let i = 0; i < m; i++) {
        for (let j = 0; j < n; j++) {
            prefix[i+1][j+1] =
                matrix[i][j]
                + prefix[i][j+1]
                + prefix[i+1][j]
                - prefix[i][j];
        }
    }
    return prefix;
}

function query(prefix, r1, c1, r2, c2) {
    return (
        prefix[r2+1][c2+1]
        - prefix[r1][c2+1]
        - prefix[r2+1][c1]
        + prefix[r1][c1]
    );
}
```

5️⃣ Prefix XOR Instead of Sum

Spoken

“Prefix XOR works the same way as prefix sum, but uses XOR. If two prefix XOR values are equal, the subarray between them has XOR zero.”

Python

```
def prefixXOR(nums):
    prefix = [0]
    for num in nums:
        prefix.append(prefix[-1] ^ num)
    return prefix
```

JavaScript

```
function prefixXOR(nums) {
    let prefix = [0];
    for (let num of nums) {
        prefix.push(prefix[prefix.length - 1] ^ num);
    }
    return prefix;
}
```

🎯 Sony Power Statement (Memorize)

“Prefix sum and its variations form the backbone of subarray and range query problems, and combining them with hash maps or sliding windows enables optimal linear-time solutions.”

---

5. Project (Concept): Audit Log Schema Design

Interview Framing

“Design an audit logging system to track user actions in ControlHub.”

Core Requirements

- Track **who did what and when**

- Immutable records
- Query-friendly
- Compliant with security audits

Basic Schema (Start Here)

SQL-Style

```
audit_logs (
   id BIGINT PRIMARY KEY,
   user_id BIGINT NOT NULL,
   action VARCHAR(100),
   resource VARCHAR(100),
   metadata JSON,
   ip_address VARCHAR(45),
   user_agent TEXT,
   created_at TIMESTAMP
)
```

MongoDB-Style

```
{
  "_id": ObjectId,
  "user_id": "U123",
  "action": "LOGIN",
  "resource": "CONTROL_PANEL",
  "metadata": {
    "status": "SUCCESS"
  },
  "ip_address": "192.168.1.10",
  "timestamp": "2025-01-01T10:30:00Z"
}
```

Design Decisions Sony Likes

- Why timestamp is indexed
- Why logs are append-only
- How to prevent log tampering
- Log retention strategy
- Separation from transactional DB

Scaling & Security (Bonus)

- Write-heavy optimization
- Partition by date
- Use Kafka for async ingestion
- Encrypt sensitive fields
- Role-based access to logs

Interview Follow-Ups on Audit Logs

1. How to ensure logs aren't modified?
2. How to handle high-volume logging?
3. How long to retain logs?
4. What if logs contain PII?
5. How to search efficiently?

---

6. How to Explain in Interview (Script)

"I'd design an append-only audit log with indexed timestamps and user IDs. For scale, logs would be written asynchronously and stored separately from core business data."
