
WEEK 1 – DAY 5

Strings + Hashing, Frequency Thinking, Practical Logging

Sony evaluates:

- Ability to **model frequency/state**
- Choice of **data structures**
- Clean edge-case handling
- Translating logic into **real backend features**

---

1. Strings + Hashing — Core Concept

Spoken Explanation (What You Say Aloud)

"Hashing allows me to store and retrieve character state—such as frequency, position, or presence—in constant time.
For string problems involving comparison, uniqueness, pattern detection, or streaming input, hashing avoids repeated scans and reduces time complexity from quadratic to linear."

When Sony Expects Hashing (Say This Briefly)

Sony expects hashing when the problem involves:

- Character **frequency counting**
- **Uniqueness** or duplicate detection
- **Pattern matching** (anagrams, substrings)
- **Streaming text processing** (real-time input)
- **Window-based string problems**

Warm-Up Interview Questions

1. Why hash map over array?
2. When is fixed-size array better?
3. Hash map vs sorting trade-offs?
4. What about Unicode characters?
5. Memory vs speed considerations?

Warm-Up Interview Follow-Ups (Answered Clearly)

1 Why hash map over array?

Answer:

"Hash maps support dynamic and large character sets like Unicode, while arrays are limited to fixed ranges like ASCII."

---

2 When is a fixed-size array better?

Answer:

"When the character set is small and known—such as lowercase English letters—arrays are faster and more memory efficient."

---

3 Hash map vs sorting trade-offs?

Answer:

"Hashing gives O(n) time with extra space, while sorting uses O(n log n) time but less auxiliary memory. For real-time or streaming data, hashing is preferred."

---

4 What about Unicode characters?

Answer:

"Unicode requires hash maps because characters are not contiguous or limited. Arrays become impractical due to size."

---

5 Memory vs speed considerations?

Answer:

"Hashing trades additional memory for faster lookups. Sony generally prioritizes speed and scalability over minimal memory usage."

---

2. Problem 1: Valid Anagram

Interview Question

Given two strings s and t, return true if t is an anagram of s.

Spoken Explanation (What You Say Aloud)

"Two strings are anagrams if they contain the same characters with the same frequencies.
I use hashing to count character occurrences in the first string and decrement those counts while

scanning the second string.
If all counts balance to zero, the strings are anagrams."

Why Sony Likes This Problem

- Tests **hashing fundamentals**
- Tests **frequency matching**
- Tests **Unicode & streaming awareness**
- Tests **time–space trade-offs**

Key Insight (Say This Clearly)

"Anagrams are about frequency equality, not order. Hashing lets me verify this in linear time."

Recommended Program Names

- **Python:** valid_anagram.py
- **JavaScript:** validAnagram.js

Python Solution (WITH INPUT)

```
# Input
s = "listen"
t = "silent"


def isAnagram(s, t):
    if len(s) != len(t):
        return False


    freq = {}


    # Count frequency of s
    for ch in s:
        freq[ch] = freq.get(ch, 0) + 1
```

```
    # Decrease frequency using t
    for ch in t:
        if ch not in freq or freq[ch] == 0:
            return False
        freq[ch] -= 1

    return True

print(isAnagram(s, t))  # Output: True
```

JavaScript Solution (WITH INPUT)

```
// Input
const s = "listen";
const t = "silent";

function isAnagram(s, t) {
    if (s.length !== t.length) return false;

    const freq = {};

    // Count frequency of s
    for (let ch of s) {
        freq[ch] = (freq[ch] || 0) + 1;
    }

    // Decrease frequency using t
    for (let ch of t) {
        if (!freq[ch]) return false;
        freq[ch]--;
    }
```

```
    return true;
}


console.log(isAnagram(s, t)); // Output: true
```

Complexity (Say This If Asked)

- **Time Complexity:** O(n)
- **Space Complexity:**
    - O(1) for fixed character set (ASCII)
    - O(n) for Unicode or unrestricted input

Sony Follow-Ups

1. Can sorting be used? Complexity?
2. How to handle Unicode?
3. Case-insensitive anagram?
4. Streamed input?
5. Memory-optimized approach?

1️⃣ Can sorting be used? Complexity?

Spoken Explanation (Say This)

“Yes, sorting can be used by sorting both strings and comparing them. However, sorting increases time complexity compared to hashing.”

Explanation

- Sort s and t
- Compare sorted strings

Complexity

- **Time:** O(n log n)
- **Space:** O(1) or O(n) depending on language

Python (WITH INPUT)

```
s = "listen"

t = "silent"
```

```
def isAnagram_sorting(s, t):
    return sorted(s) == sorted(t)

print(isAnagram_sorting(s, t))  # True
```

JavaScript (WITH INPUT)

```
const s = "listen";
const t = "silent";

function isAnagramSorting(s, t) {
    return s.split('').sort().join('') === t.split('').sort().join('');
}

console.log(isAnagramSorting(s, t)); // true
```

2️⃣ How to handle Unicode?

Spoken Explanation

"Unicode characters are not limited to a fixed range, so I must use a hash map instead of a fixed-size array."

Key Point

- Arrays of size 26 ❌
- Hash map ✅

Python Example

```
def isAnagram(s, t):
    if len(s) != len(t):
        return False

    freq = [0] * 26  # only for 'a' to 'z'

    for ch in s:
        freq[ord(ch) - ord('a')] += 1
```

```
    for ch in t:
        freq[ord(ch) - ord('a')] -= 1
        if freq[ord(ch) - ord('a')] < 0:
            return False

    return True



s = "listen"
t = "silent"

print("Are the strings anagrams?", isAnagram(s, t))
```

JavaScript Example

```
function isAnagram(s, t) {
    if (s.length !== t.length) return false;

    const freq = new Array(26).fill(0);

    for (let ch of s) {
        freq[ch.charCodeAt(0) - 97]++;
    }

    for (let ch of t) {
        freq[ch.charCodeAt(0) - 97]--;
        if (freq[ch.charCodeAt(0) - 97] < 0) return false;
    }

    return true;
}
```

```
const s = "listen";

const t = "silent";


console.log("Are the strings anagrams?", isAnagram(s, t));
```

---

3 Case-Insensitive Anagram?

Spoken Explanation

“I normalize both strings by converting them to lowercase before processing.”

Python

```
s = "Listen"

t = "Silent"


print(isAnagram(s.lower(), t.lower()))  # True
```

JavaScript

```
console.log(isAnagram("Listen".toLowerCase(), "Silent".toLowerCase())); // true
```

---

4 Streamed Input?

Spoken Explanation

“Hashing works well for streaming because character frequencies can be updated incrementally without storing the full string.”

Conceptual Python (Streaming Style)

```
from collections import defaultdict


freq = defaultdict(int)


stream_s = "abc"

stream_t = "bca"


for ch in stream_s:
    freq[ch] += 1


for ch in stream_t:
```

```
    freq[ch] -= 1


print(all(v == 0 for v in freq.values()))  # True
```

Sony Insight

- ✅ Works for logs, chat streams, real-time text
- ❌ Sorting fails for streams

5️⃣ Memory-Optimized Approach?

Spoken Explanation

“If input is restricted to lowercase English letters, I can use a fixed-size array of length 26 instead of a hash map.”

Python (Optimized)

```
s = "anagram"
t = "nagaram"


def isAnagramOptimized(s, t):
    if len(s) != len(t):
        return False

    count = [0] * 26

    for ch in s:
        count[ord(ch) - ord('a')] += 1

    for ch in t:
        idx = ord(ch) - ord('a')
        count[idx] -= 1
        if count[idx] < 0:
            return False

    return True
```

```
print(isAnagramOptimized(s, t))  # True
```

JavaScript (Optimized)

```
function isAnagramOptimized(s, t) {
    if (s.length !== t.length) return false;

    const count = new Array(26).fill(0);

    for (let ch of s) {
        count[ch.charCodeAt(0) - 97]++;
    }

    for (let ch of t) {
        const idx = ch.charCodeAt(0) - 97;
        if (--count[idx] < 0) return false;
    }
    return true;
}

console.log(isAnagramOptimized("anagram", "nagaram")); // true
```

Sony One-Line Summary (Memorize This)

"Hashing gives the most efficient anagram check at O(n), supports Unicode and streaming data, while sorting is simpler but slower."

---

3. Problem 2: First Non-Repeating Character

Interview Question

Given a string, find the index of the first non-repeating character

---

Interview Question

Given a string, find the **index of the first non-repeating character**.
Return -1 if none exists.

Spoken Explanation (What You Say Aloud)

"I solve this using a two-pass hashing approach. In the first pass, I count the frequency of each character. In the second pass, I scan the string in order and return the first index whose character frequency is exactly one. This guarantees correctness while keeping linear time complexity."

Why This Approach Is Optimal

- Preserves **original order**
- Works with **Unicode**
- Simple, predictable, and interview-safe

Recommended Program Names

- **Python:** first_non_repeating_character.py
- **JavaScript:** firstNonRepeatingCharacter.js

Python Solution (WITH INPUT)

```
# Program: first_non_repeating_character.py


def firstUniqChar(s):
    freq = {}


    # First pass: frequency count
    for ch in s:
        freq[ch] = freq.get(ch, 0) + 1


    # Second pass: find first unique
    for i, ch in enumerate(s):
        if freq[ch] == 1:
            return i


    return -1
```

```
# INPUT
s = "leetcode"
print(firstUniqChar(s))  # Output: 0
```

JavaScript Solution (WITH INPUT)

```
// Program: firstNonRepeatingCharacter.js

function firstUniqChar(s) {
    const freq = {};

    // First pass
    for (let ch of s) {
        freq[ch] = (freq[ch] || 0) + 1;
    }

    // Second pass
    for (let i = 0; i < s.length; i++) {
        if (freq[s[i]] === 1) return i;
    }

    return -1;
}

// INPUT
const s = "leetcode";
console.log(firstUniqChar(s)); // Output: 0
```

Complexity

- **Time:** O(n)
- **Space:** O(1) for fixed charset / O(n) for Unicode

Sony Follow-Ups

1. Can this be done in one pass?
2. What if string is very large?
3. Streaming characters?
4. Return character instead of index?
5. How to optimize memory?

1️⃣ Can this be done in one pass?

Spoken Explanation (Say This)

"A strict one-pass solution is not reliable because a character that appears unique at the moment may repeat later. To guarantee correctness, we need at least frequency knowledge, which typically requires two passes or additional data structures."

Key Takeaway

- ❌ Pure one pass → incorrect in general
- ✅ Two passes → deterministic and correct

2️⃣ What if the string is very large?

Spoken Explanation

"The algorithm remains linear in time. For very large strings, the main concern is memory, so I optimize by using fixed-size arrays when the character set is limited, or by processing the string in chunks if it's streamed from disk."

Practical Sony Insight

- Time complexity is optimal: O(n)
- Memory strategy depends on charset and input source

3️⃣ Streaming characters?

Spoken Explanation

"For streaming input, I maintain a frequency map and a queue of candidates. When a character's frequency exceeds one, I remove it from the front of the queue."

Python (Streaming Example with INPUT)

Program: first_non_repeating_stream.py

```
from collections import defaultdict, deque

stream = "leetcode"
freq = defaultdict(int)
queue = deque()

for ch in stream:
    freq[ch] += 1
    queue.append(ch)

    while queue and freq[queue[0]] > 1:
        queue.popleft()

print(queue[0] if queue else -1)  # Output: l
```

JavaScript (Streaming Example with INPUT)

```
// Program: firstNonRepeatingStream.js
const stream = "leetcode";
const freq = {};
const queue = [];

for (let ch of stream) {
    freq[ch] = (freq[ch] || 0) + 1;
    queue.push(ch);

    while (queue.length && freq[queue[0]] > 1) {
        queue.shift();
    }
}

console.log(queue.length ? queue[0] : -1); // l
```

4️⃣ Return character instead of index?

Spoken Explanation

“That’s a small modification. Instead of returning the index in the second pass, I return the character itself.”

Python (WITH INPUT)

```
# Program: first_unique_character.py

def firstUniqueChar(s):
    freq = {}
    for ch in s:
        freq[ch] = freq.get(ch, 0) + 1

    for ch in s:
        if freq[ch] == 1:
            return ch
    return None

print(firstUniqueChar("leetcode"))  # Output: l
```

JavaScript (WITH INPUT)

```
// Program: firstUniqueCharacter.js

function firstUniqueChar(s) {
    const freq = {};
    for (let ch of s) freq[ch] = (freq[ch] || 0) + 1;

    for (let ch of s) {
        if (freq[ch] === 1) return ch;
    }
    return null;
}

console.log(firstUniqueChar("leetcode")); // l
```

---

5 How to optimize memory?

Spoken Explanation

“If the string contains only lowercase English letters, I replace the hash map with a fixed-size array of 26, reducing memory usage and improving cache efficiency.”

Python (Memory-Optimized, WITH INPUT)

```
# Program: first_non_repeating_optimized.py

def firstUniqCharOptimized(s):
    count = [0] * 26

    for ch in s:
        count[ord(ch) - ord('a')] += 1

    for i, ch in enumerate(s):
        if count[ord(ch) - ord('a')] == 1:
            return i

    return -1

print(firstUniqCharOptimized("leetcode"))  # Output: 0
```

JavaScript (Memory-Optimized, WITH INPUT)

```
// Program: firstNonRepeatingOptimized.js

function firstUniqCharOptimized(s) {
    const count = new Array(26).fill(0);

    for (let ch of s) {
        count[ch.charCodeAt(0) - 97]++;
    }

    for (let i = 0; i < s.length; i++) {
        if (count[s.charCodeAt(i) - 97] === 1) return i;
    }
```

```
    return -1;
}

console.log(firstUniqCharOptimized("leetcode")); // 0
```

Sony One-Line Summary (Memorize This)

"Two-pass hashing guarantees correctness, streaming requires a queue-based approach, and memory can be optimized using fixed arrays when the charset is known."

---

4. Sony-Level Variations (Very Likely)

Expect **at least one**:

1. Group anagrams
2. Check palindrome permutation
3. Longest palindrome length
4. Find all anagram indices
5. Most frequent character

1️⃣ Group Anagrams

Spoken Explanation (Say This)

"I group words by a canonical representation. For each word, I compute a frequency signature (or sorted form) and use it as a hash key. Words with the same key are anagrams."

Recommended Program Names

- **Python:** group_anagrams.py
- **JavaScript:** groupAnagrams.js

Python (WITH INPUT)

```
def groupAnagrams(strs):
    from collections import defaultdict
    groups = defaultdict(list)

    for word in strs:
        key = tuple(sorted(word))
        groups[key].append(word)
```

```
    return list(groups.values())


# INPUT
words = ["eat", "tea", "tan", "ate", "nat", "bat"]
print(groupAnagrams(words))
```

JavaScript (WITH INPUT)

```
function groupAnagrams(strs) {
    const map = new Map();

    for (let word of strs) {
        const key = word.split('').sort().join('');
        if (!map.has(key)) map.set(key, []);
        map.get(key).push(word);
    }
    return Array.from(map.values());
}

// INPUT
console.log(groupAnagrams(["eat","tea","tan","ate","nat","bat"]));
```

Sony Follow-Ups

- Sorting vs frequency key → frequency gives O(n) per word.
- Unicode → hashing still works.
- Large input → prefer frequency array over sorting.

2️⃣ Check Palindrome Permutation

Spoken Explanation

"A string can form a palindrome if at most one character has an odd frequency."

Recommended Program Names

- **Python:** palindrome_permutation.py
- **JavaScript:** palindromePermutation.js

Python (WITH INPUT)

```
def canPermutePalindrome(s):
    freq = {}
    for ch in s:
        freq[ch] = freq.get(ch, 0) + 1

    odd_count = sum(1 for v in freq.values() if v % 2 == 1)
    return odd_count <= 1

# INPUT
print(canPermutePalindrome("carrace"))  # True
```

JavaScript (WITH INPUT)

```
function canPermutePalindrome(s) {
    const freq = {};
    for (let ch of s) freq[ch] = (freq[ch] || 0) + 1;

    let odd = 0;
    for (let key in freq) {
        if (freq[key] % 2 === 1) odd++;
        if (odd > 1) return false;
    }
    return true;
}

console.log(canPermutePalindrome("carrace")); // true
```

Sony Follow-Ups

- Case insensitive → normalize input.
- Streaming → track odd counts dynamically.

---

3 Longest Palindrome Length

Spoken Explanation

“I add all even frequencies fully. For odd frequencies, I add freq − 1 and allow one odd character in the center.”

Recommended Program Names

- **Python:** longest_palindrome_length.py
- **JavaScript:** longestPalindromeLength.js

Python (WITH INPUT)

```
def longestPalindrome(s):
    freq = {}
    for ch in s:
        freq[ch] = freq.get(ch, 0) + 1

    length = 0
    odd_found = False

    for count in freq.values():
        length += (count // 2) * 2
        if count % 2 == 1:
            odd_found = True

    return length + 1 if odd_found else length

# INPUT
print(longestPalindrome("abccccdd"))  # 7
```

JavaScript (WITH INPUT)

```
function longestPalindrome(s) {
    const freq = {};
    for (let ch of s) freq[ch] = (freq[ch] || 0) + 1;

    let len = 0;
    let odd = false;
```

```
    for (let k in freq) {
        len += Math.floor(freq[k] / 2) * 2;
        if (freq[k] % 2 === 1) odd = true;
    }
    return odd ? len + 1 : len;
}

console.log(longestPalindrome("abcccdd")); // 7
```

Sony Follow-Ups

- Unicode → hash map still valid.
- Memory optimization → fixed array if charset known.

4️⃣ Find All Anagram Indices

Spoken Explanation

"I use a sliding window with frequency matching. When window size equals pattern length and frequencies match, I record the index."

Recommended Program Names

- **Python:** find_all_anagrams.py
- **JavaScript:** findAllAnagrams.js

Python (WITH INPUT)

```
def findAnagrams(s, p):
    from collections import Counter
    res = []
    p_count = Counter(p)
    window = Counter()

    for i, ch in enumerate(s):
        window[ch] += 1
        if i >= len(p):
            left_char = s[i - len(p)]
            if window[left_char] == 1:
```

```
            del window[left_char]
        else:
            window[left_char] -= 1

    if window == p_count:
        res.append(i - len(p) + 1)

return res

# INPUT
print(findAnagrams("cbaebabacd", "abc"))  # [0, 6]
```

JavaScript (WITH INPUT)

```
function findAnagrams(s, p) {
  const res = [];
  const pCount = {};
  const window = {};

  for (let ch of p) pCount[ch] = (pCount[ch] || 0) + 1;

  for (let i = 0; i < s.length; i++) {
    window[s[i]] = (window[s[i]] || 0) + 1;

    if (i >= p.length) {
      const left = s[i - p.length];
      if (window[left] === 1) delete window[left];
      else window[left]--;
    }

    if (JSON.stringify(window) === JSON.stringify(pCount)) {
      res.push(i - p.length + 1);
    }
```

```
    }
    return res;
}

console.log(findAnagrams("cbaebabacd", "abc"));
```

Sony Follow-Ups

- Optimize equality check using counts.
- Streaming compatible.

---

5 Most Frequent Character

Spoken Explanation

“I count frequencies and track the character with the maximum count in a single pass.”

Recommended Program Names

- **Python:** most_frequent_character.py
- **JavaScript:** mostFrequentCharacter.js

Python (WITH INPUT)

```
def mostFrequentChar(s):
    freq = {}
    max_char = s[0]
    for ch in s:
        freq[ch] = freq.get(ch, 0) + 1
        if freq[ch] > freq[max_char]:
            max_char = ch
    return max_char

# INPUT
print(mostFrequentChar("success"))  # s
```

JavaScript (WITH INPUT)

```
function mostFrequentChar(s) {
    const freq = {};
    let maxChar = s[0];
```

```
  for (let ch of s) {
    freq[ch] = (freq[ch] || 0) + 1;
    if (freq[ch] > freq[maxChar]) maxChar = ch;
  }
  return maxChar;
}

console.log(mostFrequentChar("success")); // s
```

Sony Follow-Ups

- Tie-breaking → first occurrence.
- Large strings → linear scan optimal.

Sony Final Summary (Memorize This)

"Hashing enables linear-time solutions for grouping, counting, and pattern matching problems, which is critical for Sony's real-time and large-scale systems."

---

5. Project: Implement Audit Logging (Basic)

Interview Framing

"Implement a basic audit logging mechanism for ControlHub."

This is **not about frameworks** — it's about **clean, reliable logging**.

Core Requirements

- Log **user_id**
- Log **action**
- Log **timestamp**
- Append-only
- Non-blocking

Option A: Simple Database-Backed Logging (Flask Example)

Audit Log Model

```
from datetime import datetime
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

class AuditLog(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, nullable=False)
    action = db.Column(db.String(100), nullable=False)
    timestamp = db.Column(db.DateTime, default=datetime.utcnow)
```

Logging Helper Function

```
def log_action(user_id, action):
    log = AuditLog(user_id=user_id, action=action)
    db.session.add(log)
    db.session.commit()
```

Usage in API

```
@app.route("/update-profile", methods=["POST"])
def update_profile():
    user_id = get_current_user_id()
    # business logic here
    log_action(user_id, "UPDATE_PROFILE")
    return {"status": "success"}
```

Option B: File-Based Logging (Very Fast)

```
import logging

logging.basicConfig(
    filename="audit.log",
    level=logging.INFO,
```

```
    format="%(asctime)s | %(message)s"
)

def log_action(user_id, action):
    logging.info(f"user_id={user_id}, action={action}")
```

What Sony Likes You to Say

“I’d start with a simple logging abstraction so the storage backend can evolve without changing business logic.”

Interview Follow-Ups on Logging

1. Should logging be async?
2. What if logging fails?
3. How to prevent tampering?
4. How to add request_id?
5. How to scale logging?

---

6. How to Explain (Script)

“I encapsulate audit logging behind a helper so it’s consistent and non-intrusive. Writes are append-only, and the design allows moving to async or external systems later.”
