Faire Interview Questions
Backend Software Engineer

Compiled from 1point3acres.com

November 9, 2025

Overview

This document contains interview questions for Backend Software Engineer positions at Faire, compiled from multiple sources:

- 1point3acres.com - Interview experiences from 2022-2025
- Prepfully.com - Behavioral interview questions
- Web search results - CodeSignal OA format and general interview structure

The information represents publicly available interview experiences and may not reflect current or complete interview content. Questions span multiple positions: Backend Engineer, Full Stack Engineer, Data Engineer, and Frontend Engineer/Intern roles.

Interview Process

Complete Interview Timeline

1. **Application & Recruiter Screening**
   - Initial phone call with recruiter (15-20 minutes)
   - Discussion of background, role expectations, compensation range
   - Response time: Usually 1-3 days
2. **Online Assessment (OA)**
   - CodeSignal assessment (4 questions, 70 minutes)
   - Passing score: 750+/850
   - Must be completed within 3-5 days of invitation
3. **Technical Phone Screen**
   - Duration: 45-60 minutes
   - 1-2 coding questions (medium difficulty)
   - Live coding in shared editor (CoderPad or similar)
   - Focus on problem-solving, code quality, edge cases

4. **Virtual Onsite (3-4 rounds, 2.5-3 hours total)**
    - **Round 1:** Coding (45-60 min)
        - Medium to Hard difficulty
        - Strong emphasis on test cases and edge cases
        - May include debugging existing code
    - **Round 2:** Coding (45-60 min)
        - Similar format to Round 1
        - Different problem domain
    - **Round 3:** Behavioral/Cultural Fit (30-45 min)
        - STAR method questions
        - Team collaboration scenarios
        - Why Faire? Career goals
    - **Round 4 (Senior/Staff):** System Design or Pipeline Design (45-60 min)
        - Backend: System design (e.g., design a like functionality)
        - Data Engineering: Pipeline design
        - Focus on scalability, trade-offs, communication
5. **Final Decision**
    - Response time: Same day to 3 days
    - Offer includes: Base salary, equity, benefits discussion

Key Process Notes

- **Low Error Tolerance:** According to recent feedback (2024-2025), even mostly working code may not guarantee passing
- **Fast Process:** Usually hear back same day or next day after each round
- **Responsive HR:** Recruiters provide feedback and are generally helpful
- **Interview Difficulty:** Rated as "Hard" by most candidates
- **Test Case Emphasis:** Interviewers heavily focus on edge cases and testing

1 Coding Questions

1.1 String and Array Problems

1.1.1 Group Anagrams

PROBLEM

Problem Statement:
Given an array of strings `strs`, group the anagrams together. You can return the answer in any order.
An anagram is a word or phrase formed by rearranging the letters of a different word or

phrase, using all the original letters exactly once.

Example 1:

```
Input: strs = ["eat","tea","tan","ate","nat","bat"]
Output: [["bat"],["nat","tan"],["ate","eat","tea"]]
```

Example 2:

```
Input: strs = [""]
Output: [[""]]
```

Example 3:

```
Input: strs = ["a"]
Output: [["a"]]
```

Constraints:

- $1 \leq$ strs.length $\leq 10^4$
- $0 \leq$ strs[i].length ≤ 100
- strs[i] consists of lowercase English letters

Solution Approach:

Method 1: Sorting (Optimal)

- For each string, sort its characters to create a key
- Use a hash map where key = sorted string, value = list of original strings
- All anagrams will have the same sorted key
- Time: $O(N \cdot K \log K)$ where N = number of strings, K = max length
- Space: $O(N \cdot K)$

Method 2: Character Count (Alternative)

- Use character frequency as key (e.g., "a1b2c1" for "abc", "bac")
- Time: $O(N \cdot K)$ - better than sorting
- Space: $O(N \cdot K)$

SOLUTION

```
from collections import defaultdict

def groupAnagrams(strs):
    anagrams = defaultdict(list)

    for s in strs:
        # Use sorted string as key
```

```
        key = ''.join(sorted(s))
        anagrams[key].append(s)

    return list(anagrams.values())

# Alternative: Character count method
def groupAnagrams_v2(strs):
    anagrams = defaultdict(list)

    for s in strs:
        # Use character frequency tuple as key
        count = [0] * 26
        for c in s:
            count[ord(c) - ord('a')] += 1
        anagrams[tuple(count)].append(s)

    return list(anagrams.values())
```

TEST CASES & EDGE CASES

Edge Cases:

- Empty string: `[""]` → `[[""]]`
- Single character: `["a"]` → `[["a"]]`
- All anagrams: `["abc", "bca", "cab"]` → `[["abc", "bca", "cab"]]`
- No anagrams: `["a", "b", "c"]` → `[["a"], ["b"], ["c"]]`
- Duplicate strings: `["a", "a"]` → `[["a", "a"]]`
- Mixed lengths: `["a", "ab", "ba"]` → `[["a"], ["ab", "ba"]]`
- Case sensitivity: Problem specifies lowercase only

Test Cases:

```
# Test 1: Standard case
assert sorted([sorted(g) for g in groupAnagrams(
    ["eat","tea","tan","ate","nat","bat"])]
) == sorted([["bat"], ["nat","tan"], ["ate","eat","tea"]])

# Test 2: Empty strings
assert groupAnagrams([""]) == [[""]]

# Test 3: Single element
assert groupAnagrams(["a"]) == [["a"]]

# Test 4: No anagrams
```

```
result = groupAnagrams(["abc", "def", "ghi"])
assert len(result) == 3

# Test 5: All anagrams
result = groupAnagrams(["abc", "bca", "cab"])
assert len(result) == 1 and len(result[0]) == 3
```

Interview Tips:

- Clarify if input can have empty strings or duplicates
- Discuss both sorting and character count approaches
- Mention trade-offs: sorting is simpler, char count is faster
- For Faire: Emphasize thorough testing and edge cases

Source: Full Stack New Grad (2023-2025)

1.1.2 HTML Format Validation

PROBLEM

Problem Statement:
Given a string representing HTML-like tags with custom delimiters, determine if the tag structure is valid.
Tags are represented as:

- Opening tag: {{`tagName`}}
- Closing tag: {{`/tagName`}}
- Content: Any text between tags

A valid structure must satisfy:

- Every opening tag has a matching closing tag
- Tags are properly nested (no overlapping)
- Closing tags match the most recent unclosed opening tag

Example 1:

```
Input: "{{div}} {{p}} Hello {{/p}} {{/div}}"
Output: true
Explanation: Tags are properly nested
```

Example 2:

```
Input: "{{div}} {{p}} Hello {{/div}} {{/p}}"
Output: false
Explanation: Tags overlap incorrectly
```

Example 3:

```
Input: "{{div}} Hello {{/div}} {{p}} World {{/p}}"
Output: true
Explanation: Two separate valid tag pairs
```

Example 4:

```
Input: "{{div}} Hello"
Output: false
Explanation: Missing closing tag
```

Solution Approach:
Use a stack to track opening tags:

- Parse the string to identify tags
- When encountering opening tag: push tag name to stack
- When encountering closing tag: pop stack and verify match
- Final stack must be empty
- Time: $O(N)$ where N = length of string
- Space: $O(T)$ where T = number of tags

SOLUTION

```
def isValidHTML(s):
    stack = []
    i = 0

    while i < len(s):
        # Look for tag start
        if i < len(s) - 1 and s[i:i+2] == '{{':
            # Find tag end
            j = s.find('}}', i + 2)
            if j == -1:
                return False  # Malformed tag

            tag = s[i+2:j]

            if tag.startswith('/'):
                # Closing tag
                tag_name = tag[1:]
                if not stack or stack[-1] != tag_name:
                    return False
                stack.pop()
            else:
                # Opening tag
                stack.append(tag)
```

```
            i = j + 2
        else:
            # Regular content, skip
            i += 1

    return len(stack) == 0

# Test cases
print(isValidHTML("{{div}} {{p}} Hello {{/p}} {{/div}}"))
# True
print(isValidHTML("{{div}} {{p}} Hello {{/div}} {{/p}}"))
# False
print(isValidHTML("{{div}} Hello"))
# False
print(isValidHTML("{{div}} {{/p}}"))
# False (closing without opening)
```

TEST CASES & EDGE CASES

Edge Cases:

- Empty string: `""` → `true`
- No tags, only content: `"Hello World"` → `true`
- Unmatched opening: `"{{div}}"` → `false`
- Unmatched closing: `"{{/div}}"` → `false`
- Wrong order: `"{{div}} {{p}} {{/div}} {{/p}}"` → `false`
- Nested same tags: `"{{div}} {{div}} {{/div}} {{/div}}"` → `true`
- Malformed tags: `"{{div"` → `false`
- Self-closing tags: Clarify requirements
- Case sensitivity: Clarify if `{{Div}}` and `{{div}}` are different

Test Cases:

```
def test_html_validation():
    # Valid cases
    assert isValidHTML("") == True
    assert isValidHTML("Hello World") == True
    assert isValidHTML("{{div}} {{/div}}") == True
    assert isValidHTML("{{div}} {{p}} {{/p}} {{/div}}") == True

    # Invalid cases
    assert isValidHTML("{{div}}") == False
    assert isValidHTML("{{/div}}") == False
```

```
    assert isValidHTML("{{div}} {{p}} {{/div}} {{/p}}") == False
    assert isValidHTML("{{div}} {{p}}") == False

    # Edge cases
    assert isValidHTML("{{div}} {{div}} {{/div}} {{/div}}") == True
    assert isValidHTML("{{a}} {{b}} {{c}} {{/c}} {{/b}} {{/a}}")
           == True

test_html_validation()
```

Interview Tips:

- Clarify tag naming rules (alphanumeric? special chars?)
- Ask about self-closing tags (e.g., `<img />`)
- Discuss handling of attributes if applicable
- Mention similarity to valid parentheses problem
- For Faire: Test thoroughly with malformed input

Source: Full Stack New Grad (2023-2025)

1.1.3 Haiku Finder (5-7-5 Syllables)

PROBLEM

Problem Statement:
Given a list of words where each word has an associated syllable count, find all possible haiku formations. A haiku consists of three consecutive parts:

- First line: exactly 5 syllables
- Second line: exactly 7 syllables
- Third line: exactly 5 syllables

The three parts must be formed from consecutive words in the input list (no gaps allowed).
Input Format:

```
words = ["hello", "world", "foo", "bar", "baz", "test", "data"]
syllables = [2, 1, 1, 1, 1, 1, 2]
```

Output Format: Return all valid haiku formations as tuples of (start_idx, mid_idx, end_idx) where:

- words[start_idx:mid_idx] has 5 syllables
- words[mid_idx:end_idx] has 7 syllables
- words[end_idx:end_idx+k] has 5 syllables (where k determined by counting)

Or return the actual word groups for each line.
Example 1:

```
words = ["birds", "fly", "south", "in", "the", "winter", "cold"]
syllables = [1, 1, 1, 1, 1, 2, 1]

Output:
[
  [["birds", "fly", "south", "in"],     # 1+1+1+1 = 4? No
   ["the", "winter"],                    # 1+2 = 3? No
   ...]
]

Valid example:
words = ["spring", "rain", "falls", "gently", "on",
         "the", "garden", "flowers", "bloom"]
syllables = [1, 1, 1, 2, 1, 1, 2, 2, 1]

One valid haiku:
Line 1: ["spring", "rain", "falls", "gently"]  # 1+1+1+2 = 5
Line 2: ["on", "the", "garden", "flowers"]     # 1+1+2+2 = 6? No

Let me use a clearer example:
words = ["I", "love", "to", "code", "all", "day",
         "long", "writing", "Python"]
syllables = [1, 1, 1, 1, 1, 1, 1, 2, 2]

Valid haiku:
Line 1: ["I", "love", "to", "code", "all"]     # 1+1+1+1+1 = 5
Line 2: ["day", "long", "writing", "Python"]   # 1+1+2+2 = 6? No

Simpler:
words = ["a", "b", "c", "d", "e", "f", "g", "h", "i"]
syllables = [1, 1, 1, 1, 1, 1, 1, 1, 1, ...]

Line 1: words[0:5] = 5 syllables
Line 2: words[5:12] = 7 syllables
Line 3: words[12:17] = 5 syllables
```

Solution Approach:
Use prefix sums for efficient range queries:

- Compute prefix sum array: prefix[i] = sum of syllables[0:i]
- For each starting position i:
  - Use binary search or two pointers to find j where prefix[j] - prefix[i] = 5
  - Then find k where prefix[k] - prefix[j] = 7
  - Then find m where prefix[m] - prefix[k] = 5
  - If all three are found, record haiku

- Time: $O(N^2)$ with two pointers, or $O(N \log N)$ with binary search per position
- Space: $O(N)$ for prefix sum array

SOLUTION

```
def find_haikus(words, syllables):
    n = len(words)
    if n == 0:
        return []

    # Build prefix sum
    prefix = [0]
    for syl in syllables:
        prefix.append(prefix[-1] + syl)

    haikus = []

    # Try all starting positions
    for i in range(n):
        # Find end of line 1 (5 syllables)
        for j in range(i + 1, n + 1):
            if prefix[j] - prefix[i] == 5:
                # Find end of line 2 (7 syllables)
                for k in range(j + 1, n + 1):
                    if prefix[k] - prefix[j] == 7:
                        # Find end of line 3 (5 syllables)
                        for m in range(k + 1, n + 1):
                            if prefix[m] - prefix[k] == 5:
                                haiku = [
                                    words[i:j],
                                    words[j:k],
                                    words[k:m]
                                ]
                                haikus.append(haiku)
                        break  # Only first valid line 3
                break  # Only first valid line 2
        # Note: Can remove breaks to find ALL combos

    return haikus

# Optimized version with binary search
from bisect import bisect_left

def find_haikus_optimized(words, syllables):
    n = len(words)
    prefix = [0]
    for syl in syllables:
```

```
        prefix.append(prefix[-1] + syl)

    haikus = []

    for i in range(n):
        # Find j: prefix[j] = prefix[i] + 5
        target1 = prefix[i] + 5
        j = bisect_left(prefix, target1, i + 1)
        if j >= len(prefix) or prefix[j] != target1:
            continue

        # Find k: prefix[k] = prefix[j] + 7
        target2 = prefix[j] + 7
        k = bisect_left(prefix, target2, j + 1)
        if k >= len(prefix) or prefix[k] != target2:
            continue

        # Find m: prefix[m] = prefix[k] + 5
        target3 = prefix[k] + 5
        m = bisect_left(prefix, target3, k + 1)
        if m >= len(prefix) or prefix[m] != target3:
            continue

        haikus.append([words[i:j], words[j:k], words[k:m]])

    return haikus
```

TEST CASES & EDGE CASES

Edge Cases:

- Empty input: `words = []` → `[]`
- Insufficient syllables: total ¡ 17 → `[]`
- Exact one haiku: all words form perfect 5-7-5
- Multiple valid haikus: overlapping or non-overlapping
- Single word with many syllables: e.g., "beautiful" (3 syllables)
- Impossible to form 5-7-5: e.g., all words have 2 syllables (can't sum to 5 or 7)
- Very long word list: performance considerations

Test Cases:

```
def test_haiku_finder():
    # Test 1: Simple case with 1-syllable words
    words1 = ["a"]*5 + ["b"]*7 + ["c"]*5
```

```
    syls1 = [1]*17
    result1 = find_haikus(words1, syls1)
    assert len(result1) >= 1

    # Test 2: No valid haiku
    words2 = ["hello", "world"]
    syls2 = [2, 1]  # Only 3 syllables total
    result2 = find_haikus(words2, syls2)
    assert result2 == []

    # Test 3: Multiple word lengths
    words3 = ["I", "love", "coding", "every", "single",
              "day", "it", "makes", "me", "happy"]
    syls3 = [1, 1, 2, 2, 2, 1, 1, 1, 1, 2]
    result3 = find_haikus(words3, syls3)
    # Check if valid haikus found

    # Test 4: Empty input
    assert find_haikus([], []) == []

test_haiku_finder()
```

Interview Tips:

- Clarify if overlapping haikus are allowed
- Ask if we need ALL haikus or just first/any one
- Discuss prefix sum optimization
- Mention binary search for faster lookups
- For Faire: Thoroughly test edge cases (empty, impossible, etc.)

Source: Backend/Full Stack Phone Screen (2024-2025)

1.1.4 Funnel Problem

PROBLEM

Problem Statement:
You are analyzing a conversion funnel for an e-commerce platform. Users go through the following stages in order:

1. **Browse**: User browses product listings
2. **View**: User views product details
3. **Cart**: User adds product to cart
4. **Checkout**: User initiates checkout

5. **Purchase**: User completes purchase

Given a list of user events where each event is a tuple `(user_id, stage, timestamp)`, implement the following:

1. `calculate_conversion_rates()`: Return conversion rate for each stage transition
2. `find_dropoff_stage()`: Return the stage with highest drop-off rate
3. `get_funnel_metrics()`: Return dictionary with:
    - Total users at each stage
    - Conversion rate from previous stage
    - Overall conversion rate (Browse to Purchase)
4. `get_user_journey(user_id)`: Return ordered list of stages user visited

Rules:

- Users must progress through stages in order (can skip stages)
- A user can only be counted once per stage
- Conversion rate = (users in next stage) / (users in current stage)
- Drop-off rate = 1 - conversion rate

Example:

```
events = [
    ('u1', 'Browse', 100),
    ('u1', 'View', 105),
    ('u1', 'Cart', 110),
    ('u1', 'Purchase', 120),
    ('u2', 'Browse', 100),
    ('u2', 'View', 105),
    ('u3', 'Browse', 101),
    ('u3', 'View', 106),
    ('u3', 'Cart', 111),
]

Funnel:
Browse: 3 users (u1, u2, u3)
View: 3 users (100% conversion from Browse)
Cart: 2 users (66.7% conversion from View)
Checkout: 0 users (0% conversion from Cart)
Purchase: 1 user (N/A conversion - no one at Checkout)

Drop-off rates:
Browse -> View: 0%
View -> Cart: 33.3%
Cart -> Checkout: 100%  (highest drop-off)
```

Constraints:

- $1 \leq$ number of events $\leq 10^5$
- $1 \leq$ number of unique users $\leq 10^4$
- Timestamps are in ascending order per user
- Stage names are from the predefined list

SOLUTION

Approach:
Use sets to track unique users at each stage. Build a mapping of user journeys to handle edge cases like skipped stages or repeated events.
Time Complexity: $O(N)$ where N = number of events
Space Complexity: $O(U \times S)$ where U = users, S = stages
Python Solution:

```
from collections import defaultdict

class FunnelAnalyzer:
    def __init__(self, events):
        """
        events: list of tuples (user_id, stage, timestamp)
        """
        self.events = sorted(events, key=lambda x: (x[0], x[2]))
        self.stages = ['Browse', 'View', 'Cart', 'Checkout', 'Purchase']
        self.stage_order = {stage: i for i, stage in enumerate(self.stages)}

        # Build user journeys and stage sets
        self.user_journeys = defaultdict(list)
        self.stage_users = {stage: set() for stage in self.stages}

        self._process_events()

    def _process_events(self):
        """Process events and build user journeys"""
        for user_id, stage, timestamp in self.events:
            if stage not in self.stage_order:
                continue  # Invalid stage

            # Add to user journey
            self.user_journeys[user_id].append((stage, timestamp))

            # Add user to stage (only count once per stage)
            self.stage_users[stage].add(user_id)

    def calculate_conversion_rates(self):
```

```
        """Return conversion rate for each stage transition"""
        conversion_rates = {}

        for i in range(len(self.stages) - 1):
            current_stage = self.stages[i]
            next_stage = self.stages[i + 1]

            current_count = len(self.stage_users[current_stage])
            next_count = len(self.stage_users[next_stage])

            if current_count == 0:
                conversion_rates[f"{current_stage} -> {next_stage}"] = 0.0
            else:
                rate = (next_count / current_count) * 100
                conversion_rates[f"{current_stage} -> {next_stage}"] = round(rate, 2)

        return conversion_rates

    def find_dropoff_stage(self):
        """Return the stage with highest drop-off rate"""
        conversion_rates = self.calculate_conversion_rates()

        max_dropoff = 0
        dropoff_stage = None

        for transition, rate in conversion_rates.items():
            dropoff_rate = 100 - rate
            if dropoff_rate > max_dropoff:
                max_dropoff = dropoff_rate
                dropoff_stage = transition

        return dropoff_stage, max_dropoff

    def get_funnel_metrics(self):
        """Return comprehensive funnel metrics"""
        metrics = {}

        for i, stage in enumerate(self.stages):
            stage_count = len(self.stage_users[stage])

            metric = {
                'total_users': stage_count,
                'conversion_from_previous': None,
                'overall_conversion': None
            }
```

```
            # Conversion from previous stage
            if i > 0:
                prev_stage = self.stages[i - 1]
                prev_count = len(self.stage_users[prev_stage])
                if prev_count > 0:
                    metric['conversion_from_previous'] = round(
                        (stage_count / prev_count) * 100, 2
                    )

            # Overall conversion (from Browse)
            browse_count = len(self.stage_users['Browse'])
            if browse_count > 0:
                metric['overall_conversion'] = round(
                    (stage_count / browse_count) * 100, 2
                )

            metrics[stage] = metric

        return metrics

    def get_user_journey(self, user_id):
        """Return ordered list of stages user visited"""
        if user_id not in self.user_journeys:
            return []

        # Return unique stages in order
        seen = set()
        journey = []
        for stage, _ in self.user_journeys[user_id]:
            if stage not in seen:
                journey.append(stage)
                seen.add(stage)

        return journey

    def validate_user_journey(self, user_id):
        """Check if user followed correct order"""
        journey = self.get_user_journey(user_id)

        for i in range(len(journey) - 1):
            current_idx = self.stage_order[journey[i]]
            next_idx = self.stage_order[journey[i + 1]]

            # Next stage should come after current stage
            if next_idx <= current_idx:
                return False, f"Invalid order: {journey[i]} -> {journey[i+1]}"
```

```
        return True, "Valid journey"


# Example usage
events = [
    ('u1', 'Browse', 100),
    ('u1', 'View', 105),
    ('u1', 'Cart', 110),
    ('u1', 'Purchase', 120),
    ('u2', 'Browse', 100),
    ('u2', 'View', 105),
    ('u3', 'Browse', 101),
    ('u3', 'View', 106),
    ('u3', 'Cart', 111),
]

analyzer = FunnelAnalyzer(events)

print("Conversion Rates:")
print(analyzer.calculate_conversion_rates())

print("\nHighest Drop-off:")
print(analyzer.find_dropoff_stage())

print("\nFunnel Metrics:")
for stage, metrics in analyzer.get_funnel_metrics().items():
    print(f"{stage}: {metrics}")

print("\nUser Journey (u1):")
print(analyzer.get_user_journey('u1'))
```

TEST CASES & EDGE CASES

Edge Cases & Testing:

- Empty events list: all metrics return 0
- Single user, single event: 100% drop-off after first stage
- User skips stages: e.g., Browse -¿ Purchase (validate journey)
- User repeats same stage: count only once
- User goes backwards: Browse -¿ View -¿ Browse (invalid order)
- All users complete funnel: 100% conversion at each stage
- No users reach final stage: Purchase conversion = 0

- Duplicate timestamps: handle tie-breaking
- Invalid stage names: filter out or raise error
- Division by zero: when no users at a stage

Test Cases:

```
def test_funnel_analyzer():
    # Test 1: Basic funnel
    events = [
        ('u1', 'Browse', 1), ('u1', 'View', 2),
        ('u2', 'Browse', 1)
    ]
    analyzer = FunnelAnalyzer(events)
    metrics = analyzer.get_funnel_metrics()
    assert metrics['Browse']['total_users'] == 2
    assert metrics['View']['total_users'] == 1
    assert metrics['View']['conversion_from_previous'] == 50.0

    # Test 2: Complete journey
    events = [
        ('u1', 'Browse', 1), ('u1', 'View', 2),
        ('u1', 'Cart', 3), ('u1', 'Checkout', 4),
        ('u1', 'Purchase', 5)
    ]
    analyzer = FunnelAnalyzer(events)
    journey = analyzer.get_user_journey('u1')
    assert len(journey) == 5
    assert journey[-1] == 'Purchase'

    # Test 3: Drop-off detection
    events = [
        ('u1', 'Browse', 1), ('u1', 'View', 2),
        ('u2', 'Browse', 1), ('u2', 'View', 2),
        ('u3', 'Browse', 1), ('u3', 'View', 2),
        ('u1', 'Cart', 3)  # Only u1 proceeds
    ]
    analyzer = FunnelAnalyzer(events)
    dropoff_stage, rate = analyzer.find_dropoff_stage()
    assert 'View -> Cart' in dropoff_stage
    assert rate > 60  # ~66.7% drop-off

    # Test 4: Duplicate events (same stage)
    events = [
        ('u1', 'Browse', 1),
        ('u1', 'Browse', 2),  # Duplicate
        ('u1', 'View', 3)
```

```
    ]
    analyzer = FunnelAnalyzer(events)
    assert len(analyzer.stage_users['Browse']) == 1

    # Test 5: Empty events
    analyzer = FunnelAnalyzer([])
    metrics = analyzer.get_funnel_metrics()
    assert all(m['total_users'] == 0 for m in metrics.values())

    # Test 6: Skipped stages
    events = [
        ('u1', 'Browse', 1),
        ('u1', 'Purchase', 5)  # Skips View, Cart, Checkout
    ]
    analyzer = FunnelAnalyzer(events)
    journey = analyzer.get_user_journey('u1')
    assert journey == ['Browse', 'Purchase']

    print("All funnel tests passed!")

test_funnel_analyzer()
```

Interview Tips:

- Clarify whether users can skip stages
- Ask about handling duplicate events for same stage
- Discuss how to handle out-of-order events
- Consider time windows for conversion (e.g., must complete within 24h)
- For Faire: **Focus heavily on edge cases and testing**
- Use proper testing framework (JUnit, pytest) if allowed
- Validate input data (null checks, invalid stages)
- Consider performance with large datasets

Note: Interviewer heavily focused on test cases and edge cases. Even after comprehensive testing, be prepared for more edge case questions. Source: Coding interview (2024-2025)

1.1.5 Number to Word Conversion

PROBLEM

Problem Statement:
Given two integers `start` and `end`, convert all numbers in the range $[start, end]$ (inclusive) to their English word representations and calculate the total length of all characters (excluding spaces).
For example: 1 = "one", 2 = "two", ..., 23 = "twenty three"
Return the total length of all word representations.
Example 1:

```
Input: start = 1, end = 3
Output: 11

Explanation:
1 -> "one" (3 chars)
2 -> "two" (3 chars)
3 -> "three" (5 chars)
Total: 3 + 3 + 5 = 11
```

Example 2:

```
Input: start = 10, end = 12
Output: 22

Explanation:
10 -> "ten" (3 chars)
11 -> "eleven" (6 chars)
12 -> "twelve" (6 chars)
Total: 3 + 6 + 6 = 15 (NOT 22, recalculating...)
Actually: 3 + 6 + 6 = 15

Or if counting spaces:
10 -> "ten" (3)
11 -> "eleven" (6)
12 -> "twelve" (6)
= 15 without spaces
```

Constraints:

- $1 \leq start \leq end \leq 1000$
- Count only letters, not spaces

SOLUTION

Solution Approach:
Build number-to-word converter:

- Create maps for 1-19, tens (20, 30,...), and hundreds

- For each number, convert to words recursively
- Sum up character counts
- Time: $O(N)$ where N = end - start + 1
- Space: $O(1)$ for conversion maps

Python Solution:

```
def number_to_words(num):
    """Convert number (1-1000) to English words"""
    if num == 0:
        return "zero"

    ones = ["", "one", "two", "three", "four", "five",
            "six", "seven", "eight", "nine"]
    teens = ["ten", "eleven", "twelve", "thirteen",
             "fourteen", "fifteen", "sixteen",
             "seventeen", "eighteen", "nineteen"]
    tens = ["", "", "twenty", "thirty", "forty", "fifty",
            "sixty", "seventy", "eighty", "ninety"]

    def helper(n):
        if n == 0:
            return ""
        elif n < 10:
            return ones[n]
        elif n < 20:
            return teens[n - 10]
        elif n < 100:
            return tens[n // 10] + \
                   (" " + ones[n % 10] if n % 10 != 0 else "")
        else:
            return ones[n // 100] + " hundred" + \
                   (" " + helper(n % 100) if n % 100 != 0
                    else "")

    if num == 1000:
        return "one thousand"
    return helper(num)

def total_word_length(start, end):
    total = 0
    for num in range(start, end + 1):
        words = number_to_words(num)
        # Remove spaces and count length
        length = len(words.replace(" ", ""))
```

```
        total += length
    return total

# Tests
print(total_word_length(1, 3))   # 11
print(total_word_length(10, 12)) # 15
print(total_word_length(1, 5))   # one+two+three+four+five
                                 # 3+3+5+4+4 = 19
```

TEST CASES & EDGE CASES

Edge Cases & Testing:

- Single number: start = end = 1 → 3
- Teens (11-19): special handling
- Exact tens (20, 30, ...): no "and"
- Hundreds: 100 = "one hundred" (10 chars)
- 1000: "one thousand" (special case)
- Crossing boundaries: [19, 21] includes "nineteen", "twenty", "twenty one"

Test Cases:

```
def test_number_words():
    # Test individual conversions
    assert number_to_words(1) == "one"
    assert number_to_words(11) == "eleven"
    assert number_to_words(20) == "twenty"
    assert number_to_words(100) == "one hundred"
    assert number_to_words(1000) == "one thousand"

    # Test total length
    assert total_word_length(1, 1) == 3  # "one"
    assert total_word_length(1, 3) == 11 # 3+3+5
    assert total_word_length(10, 10) == 3 # "ten"

test_number_words()
```

Interview Tips:

- Clarify if spaces count toward length
- Ask about range limits (problem says ≤ 1000)
- Discuss British vs American English ("and" placement)
- Mention potential optimization for repeated ranges
- For Faire: Test edge cases like 1000, teens, exact hundreds

Source: Backend/Full Stack - Canadian Office (2022)

1.2 Graph and Path Problems

1.2.1 Course Schedule with Minimum Time

PROBLEM

Problem Statement:
You are given:

- `n` courses labeled from 0 to n-1
- `prerequisites`: array where `prerequisites[i] = [a, b]` means course `b` must be completed before course `a`
- `time`: array where `time[i]` is the duration (in days/weeks) to complete course `i`

Find the minimum time required to complete all courses. You can take multiple courses simultaneously if their prerequisites are met.

Example 1:

```
n = 3
prerequisites = [[1, 0], [2, 0]]
time = [1, 2, 3]

Course 0: 1 day, no prerequisites
Course 1: 2 days, requires course 0
Course 2: 3 days, requires course 0

Timeline:
Day 0-1: Complete course 0 (1 day)
Day 1-3: Complete courses 1 and 2 in parallel (max 3 days)

Total: 1 + 3 = 4 days

Output: 4
```

Example 2:

```
n = 4
prerequisites = [[1, 0], [2, 1], [3, 2]]
time = [1, 1, 1, 1]

Linear dependency: 0 -> 1 -> 2 -> 3
Must complete sequentially: 1+1+1+1 = 4 days

Output: 4
```

Example 3:

```
n = 4
prerequisites = [[1, 0], [2, 0], [3, 1], [3, 2]]
time = [1, 2, 3, 1]

Course 0: 1 day
Courses 1, 2: after 0 (2, 3 days) - parallel
Course 3: after both 1 and 2 (1 day)

Timeline:
Day 0-1: Course 0
Day 1-4: Courses 1 (done day 3) and 2 (done day 4) parallel
Day 4-5: Course 3

Total: 5 days

Output: 5
```

SOLUTION

Solution Approach:
Topological Sort + Critical Path Method (CPM):

- Build graph from prerequisites
- Check for cycles (if cycle exists, return -1)
- Use topological sort with earliest start time tracking:
  - `earliest[i]` = earliest time course `i` can start
  - For each course: `earliest[i] = max(earliest[prereq] + time[prereq])` `for all prereqs`
- Answer = `max(earliest[i] + time[i]) for all i`
- Time: $O(V + E)$ where V = courses, E = prerequisites
- Space: $O(V + E)$

Python Solution:

```
from collections import defaultdict, deque

def minimum_time_courses(n, prerequisites, time):
    # Build graph and in-degree
    graph = defaultdict(list)
    in_degree = [0] * n

    for course, prereq in prerequisites:
        graph[prereq].append(course)
        in_degree[course] += 1
```

```
    # Initialize earliest start times
    earliest = [0] * n

    # Topological sort using Kahn's algorithm
    queue = deque()
    for i in range(n):
        if in_degree[i] == 0:
            queue.append(i)

    completed = 0

    while queue:
        course = queue.popleft()
        completed += 1

        # Process neighbors
        for next_course in graph[course]:
            # Update earliest start time for next_course
            earliest[next_course] = max(
                earliest[next_course],
                earliest[course] + time[course]
            )

            in_degree[next_course] -= 1
            if in_degree[next_course] == 0:
                queue.append(next_course)

    # Check for cycle
    if completed != n:
        return -1  # Impossible due to cycle

    # Calculate total time (max finish time)
    max_time = 0
    for i in range(n):
        finish_time = earliest[i] + time[i]
        max_time = max(max_time, finish_time)

    return max_time

# Tests
print(minimum_time_courses(
    3, [[1, 0], [2, 0]], [1, 2, 3]
))  # Output: 4

print(minimum_time_courses(
    4, [[1, 0], [2, 1], [3, 2]], [1, 1, 1, 1]
```

```
))  # Output: 4

print(minimum_time_courses(
    4, [[1, 0], [2, 0], [3, 1], [3, 2]], [1, 2, 3, 1]
))  # Output: 5
```

TEST CASES & EDGE CASES

Edge Cases & Testing:

- No prerequisites: `max(time)` (all parallel)
- Linear chain: `sum(time)` (all sequential)
- Cycle in prerequisites: return -1
- Single course: `time[0]`
- Zero time courses: valid, acts as dependency marker
- Disconnected components: multiple independent course chains
- Course with multiple prerequisites: must wait for ALL

Test Cases:

```
def test_course_schedule_time():
    # Test 1: Parallel courses
    assert minimum_time_courses(
        3, [[1, 0], [2, 0]], [1, 2, 3]
    ) == 4

    # Test 2: Linear sequence
    assert minimum_time_courses(
        3, [[1, 0], [2, 1]], [1, 2, 3]
    ) == 6

    # Test 3: No prerequisites (all parallel)
    assert minimum_time_courses(
        3, [], [1, 2, 3]
    ) == 3

    # Test 4: Cycle detection
    assert minimum_time_courses(
        2, [[0, 1], [1, 0]], [1, 1]
    ) == -1

    # Test 5: Diamond dependency
    assert minimum_time_courses(
        4, [[1, 0], [2, 0], [3, 1], [3, 2]],
```

```
        [1, 2, 3, 1]
    ) == 5

test_course_schedule_time()
```

Interview Tips:

- Recognize this as Critical Path Method (CPM) problem
- Discuss difference from standard Course Schedule
- Mention that parallel execution is key
- Explain why we take max of prerequisite finish times
- For Faire: Test cycle detection and edge cases

Source: Senior SWE Interview (2022)

1.3 LeetCode-style Problems

1.3.1 Max Consecutive Ones

PROBLEM

Problem Statement:
Given a binary array `nums`, return the maximum number of consecutive 1's in the array.
Example 1:

```
Input: nums = [1,1,0,1,1,1]
Output: 3
Explanation: The first two digits or the last three digits
             are consecutive 1s. The maximum is 3.
```

Example 2:

```
Input: nums = [1,0,1,1,0,1]
Output: 2
Explanation: Maximum consecutive 1s is 2.
```

Example 3:

```
Input: nums = [0,0,0]
Output: 0
Explanation: No 1s in the array.
```

Constraints:

- $1 \leq$ nums.length $\leq 10^5$
- nums[i] is either 0 or 1

SOLUTION

Solution Approach:
Single pass with counter:

- Track current consecutive 1s count
- Track maximum seen so far
- Reset counter when encountering 0
- Time: $O(N)$ where N = array length
- Space: $O(1)$

Python Solution:

```
def findMaxConsecutiveOnes(nums):
    max_count = 0
    current_count = 0

    for num in nums:
        if num == 1:
            current_count += 1
            max_count = max(max_count, current_count)
        else:
            current_count = 0

    return max_count

# Alternative: More concise
def findMaxConsecutiveOnes_v2(nums):
    # Convert to string and split on '0'
    return max(len(s) for s in ''.join(map(str, nums)).split('0'))

# Tests
print(findMaxConsecutiveOnes([1,1,0,1,1,1]))  # 3
print(findMaxConsecutiveOnes([1,0,1,1,0,1]))  # 2
print(findMaxConsecutiveOnes([0,0,0]))        # 0
print(findMaxConsecutiveOnes([1]))            # 1
```

TEST CASES & EDGE CASES

Edge Cases & Testing:

- All 1s: [1,1,1,1] $\rightarrow$ 4
- All 0s: [0,0,0] $\rightarrow$ 0
- Single element: [1] $\rightarrow$ 1, [0] $\rightarrow$ 0
- Alternating: [1,0,1,0] $\rightarrow$ 1

- Leading/trailing zeros: [0,1,1,0] → 2
- Multiple sequences: [1,1,0,0,1,1,1,0,1] → 3

Test Cases:

```
def test_max_consecutive_ones():
    assert findMaxConsecutiveOnes([1,1,0,1,1,1]) == 3
    assert findMaxConsecutiveOnes([1,0,1,1,0,1]) == 2
    assert findMaxConsecutiveOnes([0,0,0]) == 0
    assert findMaxConsecutiveOnes([1,1,1,1]) == 4
    assert findMaxConsecutiveOnes([1]) == 1
    assert findMaxConsecutiveOnes([0]) == 0
    assert findMaxConsecutiveOnes([1,0,1,0,1]) == 1

test_max_consecutive_ones()
```

Interview Tips:

- Discuss streaming/online algorithm variant
- Mention follow-up: max consecutive 1s with one flip allowed
- Consider space-optimized approaches
- For Faire: Test boundary cases

Source: Coding interview (2024-2025), LeetCode 485

1.4 Additional Coding Problems

1.4.1 Date Format Function

PROBLEM

Problem Statement:
Implement a function to convert dates between different formats. Given a date string and its format, convert it to a target format.
Support the following formats:

- "MM/DD/YYYY" (e.g., "03/15/2024")
- "YYYY-MM-DD" (e.g., "2024-03-15")
- "DD.MM.YYYY" (e.g., "15.03.2024")
- "Month DD, YYYY" (e.g., "March 15, 2024")

Return `None` or raise an error if the input date is invalid.
Example 1:

```
Input: date = "03/15/2024", from_format = "MM/DD/YYYY",
       to_format = "YYYY-MM-DD"
```

```
Output: "2024-03-15"
```

Example 2:

```
Input: date = "2024-02-29", from_format = "YYYY-MM-DD",
       to_format = "MM/DD/YYYY"
Output: "02/29/2024"
Explanation: 2024 is a leap year, Feb 29 is valid.
```

Example 3:

```
Input: date = "2023-02-29", from_format = "YYYY-MM-DD",
       to_format = "MM/DD/YYYY"
Output: None
Explanation: 2023 is not a leap year, Feb 29 is invalid.
```

Constraints:

- Year range: 1000-9999
- Validate month (1-12) and day based on month/year
- Handle leap years correctly

SOLUTION

Solution Approach:
Date validation and conversion:

- Parse input format to extract day, month, year
- Validate the date (check month range, day range, leap year)
- Format according to target format
- Time: $O(1)$ - constant time for parsing and formatting
- Space: $O(1)$

Python Solution:

```
from datetime import datetime

def is_leap_year(year):
    """Check if year is a leap year"""
    return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)

def is_valid_date(day, month, year):
    """Validate if date is valid"""
    if month < 1 or month > 12:
        return False
    if year < 1000 or year > 9999:
```

```
        return False

    days_in_month = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    if is_leap_year(year):
        days_in_month[1] = 29

    if day < 1 or day > days_in_month[month - 1]:
        return False

    return True

def convert_date_format(date, from_format, to_format):
    """Convert date from one format to another"""
    month_names = {
        'January': 1, 'February': 2, 'March': 3, 'April': 4,
        'May': 5, 'June': 6, 'July': 7, 'August': 8,
        'September': 9, 'October': 10, 'November': 11, 'December': 12
    }
    month_names_rev = {v: k for k, v in month_names.items()}

    # Parse based on from_format
    try:
        if from_format == "MM/DD/YYYY":
            parts = date.split('/')
            month, day, year = int(parts[0]), int(parts[1]), int(parts[2])
        elif from_format == "YYYY-MM-DD":
            parts = date.split('-')
            year, month, day = int(parts[0]), int(parts[1]), int(parts[2])
        elif from_format == "DD.MM.YYYY":
            parts = date.split('.')
            day, month, year = int(parts[0]), int(parts[1]), int(parts[2])
        elif from_format == "Month DD, YYYY":
            parts = date.replace(',', '').split()
            month = month_names[parts[0]]
            day, year = int(parts[1]), int(parts[2])
        else:
            return None
    except:
        return None

    # Validate
    if not is_valid_date(day, month, year):
        return None

    # Format to target
    if to_format == "MM/DD/YYYY":
```

```
        return f"{month:02d}/{day:02d}/{year}"
    elif to_format == "YYYY-MM-DD":
        return f"{year}-{month:02d}-{day:02d}"
    elif to_format == "DD.MM.YYYY":
        return f"{day:02d}.{month:02d}.{year}"
    elif to_format == "Month DD, YYYY":
        return f"{month_names_rev[month]} {day:02d}, {year}"
    else:
        return None

# Tests
print(convert_date_format("03/15/2024", "MM/DD/YYYY", "YYYY-MM-DD"))
# "2024-03-15"

print(convert_date_format("2024-02-29", "YYYY-MM-DD", "MM/DD/YYYY"))
# "02/29/2024"

print(convert_date_format("2023-02-29", "YYYY-MM-DD", "MM/DD/YYYY"))
# None (invalid)
```

TEST CASES & EDGE CASES

Edge Cases & Testing:

- Leap year: Feb 29, 2024 (valid) vs Feb 29, 2023 (invalid)
- Century leap year: Feb 29, 2000 (valid) vs Feb 29, 1900 (invalid)
- Invalid month: 13/01/2024 → None
- Invalid day: 02/31/2024 → None
- Month boundaries: 30 vs 31 day months
- Different delimiters: handle /, -, .

Test Cases:

```
def test_date_conversion():
    # Valid conversions
    assert convert_date_format("03/15/2024", "MM/DD/YYYY",
                               "YYYY-MM-DD") == "2024-03-15"
    assert convert_date_format("2024-02-29", "YYYY-MM-DD",
                               "MM/DD/YYYY") == "02/29/2024"

    # Invalid dates
    assert convert_date_format("2023-02-29", "YYYY-MM-DD",
                               "MM/DD/YYYY") is None
    assert convert_date_format("13/01/2024", "MM/DD/YYYY",
                               "YYYY-MM-DD") is None
```

```
    assert convert_date_format("02/31/2024", "MM/DD/YYYY",
                               "YYYY-MM-DD") is None

    # Leap year edge cases
    assert convert_date_format("2000-02-29", "YYYY-MM-DD",
                               "MM/DD/YYYY") == "02/29/2000"
    assert convert_date_format("1900-02-29", "YYYY-MM-DD",
                               "MM/DD/YYYY") is None

test_date_conversion()
```

Interview Tips:

- Discuss using datetime library vs manual validation
- Clarify supported formats upfront
- Ask about timezone handling
- Mention error handling strategies
- For Faire: Test leap year and boundary cases thoroughly

Source: Frontend/Full Stack interview (2024)

1.4.2 Maze with Portals

PROBLEM

Problem Statement:
You are given a 2D grid maze where:

- `0` = walkable cell
- `1` = wall (cannot pass)
- `S` = start position
- `E` = end position
- `P` = portal (teleporter)

Additionally, you have a dictionary mapping portal positions to their destinations. When you step on a portal, you are instantly teleported to its destination.
Find the shortest path from S to E. Return the minimum number of steps, or -1 if impossible.
Movement: You can move up, down, left, right (4 directions). Each move counts as 1 step. Portal teleportation does NOT count as an extra step.
Example 1:

```
maze = [
  ['S', '0', '1', '0'],
  ['0', '1', 'P', '0'],
```

```
  ['0', '0', '0', '1'],
  ['1', '0', 'E', '0']
]

portals = {(1, 2): (3, 1)}  # Portal at (1,2) goes to (3,1)

Shortest path:
S(0,0) -> (1,0) -> (2,0) -> (2,1) -> (2,2) -> Portal(1,2)
       -> Teleport to (3,1) -> E(3,2)

Without portal: 6 steps
With portal: 4 steps (reach portal) + 1 step (to E) = 5 steps

Output: 5
```

Example 2:

```
maze = [
  ['S', '1', 'E']
]
Output: -1 (impossible, wall blocks path)
```

SOLUTION

Solution Approach:
Use BFS (Breadth-First Search) with portal handling:

- Standard BFS tracks (row, col, distance)
- When visiting a cell:
  - If it's a portal, add BOTH normal neighbors AND portal destination to queue
  - Portal destination inherits same distance (instant teleport)
- Use visited set to avoid cycles
- Time: $O(R \times C)$ where R = rows, C = columns
- Space: $O(R \times C)$ for queue and visited set

Python Solution:

```
from collections import deque

def shortest_path_with_portals(maze, portals):
    if not maze or not maze[0]:
        return -1

    rows, cols = len(maze), len(maze[0])
```

```
    # Find start and end
    start, end = None, None
    for r in range(rows):
        for c in range(cols):
            if maze[r][c] == 'S':
                start = (r, c)
            elif maze[r][c] == 'E':
                end = (r, c)

    if not start or not end:
        return -1

    # BFS
    queue = deque([(start[0], start[1], 0)])  # (row, col, dist)
    visited = {start}
    directions = [(0,1), (1,0), (0,-1), (-1,0)]

    while queue:
        r, c, dist = queue.popleft()

        # Check if reached end
        if (r, c) == end:
            return dist

        # Check if current cell is a portal
        if (r, c) in portals:
            pr, pc = portals[(r, c)]
            if (pr, pc) not in visited and \
               0 <= pr < rows and 0 <= pc < cols and \
               maze[pr][pc] != '1':
                visited.add((pr, pc))
                queue.append((pr, pc, dist))

        # Explore normal neighbors
        for dr, dc in directions:
            nr, nc = r + dr, c + dc

            if (0 <= nr < rows and 0 <= nc < cols and
                (nr, nc) not in visited and
                maze[nr][nc] != '1'):

                visited.add((nr, nc))
                queue.append((nr, nc, dist + 1))

    return -1  # No path found
```

```
# Test
maze1 = [
    ['S', '0', '1', '0'],
    ['0', '1', 'P', '0'],
    ['0', '0', '0', '1'],
    ['1', '0', 'E', '0']
]
portals1 = {(1, 2): (3, 1)}
print(shortest_path_with_portals(maze1, portals1))
```

TEST CASES & EDGE CASES

Edge Cases & Testing:

- Empty maze: return -1
- No start or end: return -1
- Start = End: return 0
- No path exists (walls block): return -1
- Portal leads to wall: ignore portal
- Portal leads out of bounds: ignore portal
- Multiple portals: test chaining portals
- Portal at start position: can immediately teleport
- Portal at end position: doesn't affect result
- Bidirectional portals: clarify if portals work both ways
- Portal cycles: A → B, B → A (visited set handles this)

Test Cases:

```
def test_maze_portals():
    # Test 1: With portal shortcut
    maze1 = [['S', '0', 'P'], ['1', '1', '0'], ['E', '0', '0']]
    portals1 = {(0, 2): (2, 0)}
    assert shortest_path_with_portals(maze1, portals1) == 2

    # Test 2: No path
    maze2 = [['S', '1', 'E']]
    portals2 = {}
    assert shortest_path_with_portals(maze2, portals2) == -1

    # Test 3: Direct path better than portal
    maze3 = [['S', '0', 'E']]
```

```
    portals3 = {(0, 1): (0, 0)}  # Portal doesn't help
    assert shortest_path_with_portals(maze3, portals3) == 2

    # Test 4: Start = End
    maze4 = [['S']]
    portals4 = {}
    # Modify to handle S=E: return 0 if (r,c) == end initially

test_maze_portals()
```

Interview Tips:

- Clarify if portals are one-way or bidirectional
- Ask if portal teleportation counts as a step
- Discuss handling of portal cycles/loops
- Mention BFS is optimal for shortest path
- For Faire: Test portal edge cases thoroughly

Source: Backend Engineer Phone Screen (2024)

1.4.3 String Parsing Problem

PROBLEM

Problem Statement:
Parse a log file string containing structured data and extract specific information. Each log entry has the format:
`[TIMESTAMP] LEVEL: message | key1=value1 key2=value2`
Given a log string with multiple entries (separated by newlines), return a list of dictionaries where each dictionary contains:

- `timestamp`: The timestamp string
- `level`: The log level (INFO, ERROR, WARNING, DEBUG)
- `message`: The message text
- `metadata`: Dictionary of key-value pairs

Example 1:

```
Input:
"[2024-01-15 10:30:00] INFO: User login | user_id=123 ip=192.168.1.1
[2024-01-15 10:31:00] ERROR: Failed query | query_id=456 error_code=500"

Output: [
  {
```

```
    'timestamp': '2024-01-15 10:30:00',
    'level': 'INFO',
    'message': 'User login',
    'metadata': {'user_id': '123', 'ip': '192.168.1.1'}
  },
  {
    'timestamp': '2024-01-15 10:31:00',
    'level': 'ERROR',
    'message': 'Failed query',
    'metadata': {'query_id': '456', 'error_code': '500'}
  }
]
```

Constraints:

- $1 \leq$ number of log entries ≤ 1000
- Metadata may have 0 or more key-value pairs
- Keys and values do not contain spaces or special characters

SOLUTION

Solution Approach:
String parsing with regex or manual parsing:

- Split input by newlines to get individual log entries
- For each entry, extract components using string operations or regex
- Parse metadata by splitting on spaces and equals signs
- Time: $O(N \times M)$ where N = entries, M = avg entry length
- Space: $O(N \times M)$ for storing parsed results

Python Solution:

```
import re

def parse_logs(log_string):
    """Parse log entries from string"""
    if not log_string.strip():
        return []

    entries = []
    lines = log_string.strip().split('\n')

    for line in lines:
        if not line.strip():
            continue
```

```
        # Extract timestamp
        timestamp_match = re.search(r'\[(.*?)\]', line)
        if not timestamp_match:
            continue
        timestamp = timestamp_match.group(1)

        # Extract level
        level_match = re.search(r'\]\s+(\w+):', line)
        if not level_match:
            continue
        level = level_match.group(1)

        # Extract message and metadata
        parts = line.split(':', 1)
        if len(parts) < 2:
            continue

        remaining = parts[1].strip()
        if '|' in remaining:
            message, metadata_str = remaining.split('|', 1)
            message = message.strip()
            metadata = {}

            # Parse metadata
            metadata_pairs = metadata_str.strip().split()
            for pair in metadata_pairs:
                if '=' in pair:
                    key, value = pair.split('=', 1)
                    metadata[key] = value
        else:
            message = remaining.strip()
            metadata = {}

        entries.append({
            'timestamp': timestamp,
            'level': level,
            'message': message,
            'metadata': metadata
        })

    return entries

# Alternative: Manual parsing without regex
def parse_logs_manual(log_string):
    """Parse without regex"""
    entries = []
```

```
    for line in log_string.strip().split('\n'):
        if not line.strip():
            continue

        # Find timestamp
        start = line.find('[')
        end = line.find(']')
        if start == -1 or end == -1:
            continue
        timestamp = line[start+1:end]

        # Find level
        colon_idx = line.find(':', end)
        if colon_idx == -1:
            continue
        level = line[end+1:colon_idx].strip()

        # Find message and metadata
        remaining = line[colon_idx+1:].strip()
        if '|' in remaining:
            message, meta_str = remaining.split('|', 1)
            metadata = {}
            for pair in meta_str.strip().split():
                if '=' in pair:
                    k, v = pair.split('=', 1)
                    metadata[k] = v
        else:
            message = remaining
            metadata = {}

        entries.append({
            'timestamp': timestamp,
            'level': level,
            'message': message.strip(),
            'metadata': metadata
        })

    return entries

# Test
log = """[2024-01-15 10:30:00] INFO: User login | user_id=123 ip=192.168.1.1
[2024-01-15 10:31:00] ERROR: Failed query | query_id=456"""

result = parse_logs(log)
print(result[0]['timestamp'])  # '2024-01-15 10:30:00'
print(result[0]['metadata']['user_id'])  # '123'
```

TEST CASES & EDGE CASES

Edge Cases & Testing:

- Empty string: return []
- Entry without metadata: only timestamp, level, message
- Multiple metadata fields: parse all key-value pairs
- Malformed entries: skip or handle gracefully
- Special characters in message: preserve them
- Empty metadata after pipe: handle empty dict

Test Cases:

```
def test_log_parsing():
    # Test 1: Standard parsing
    log1 = "[2024-01-15 10:30:00] INFO: Test | key=value"
    result = parse_logs(log1)
    assert result[0]['level'] == 'INFO'
    assert result[0]['metadata']['key'] == 'value'

    # Test 2: No metadata
    log2 = "[2024-01-15 10:30:00] ERROR: Test message"
    result = parse_logs(log2)
    assert result[0]['message'] == 'Test message'
    assert result[0]['metadata'] == {}

    # Test 3: Multiple metadata
    log3 = "[2024-01-15 10:30:00] DEBUG: Test | a=1 b=2 c=3"
    result = parse_logs(log3)
    assert len(result[0]['metadata']) == 3

    # Test 4: Empty input
    assert parse_logs("") == []

test_log_parsing()
```

Interview Tips:

- Discuss regex vs manual parsing trade-offs
- Ask about error handling for malformed entries
- Clarify if metadata keys/values can have special chars
- Mention performance for large log files
- For Faire: Test edge cases thoroughly

Source: Backend Engineer coding round (2024)

2 Online Assessment (OA) - CodeSignal

2.1 OA Format and Structure

- **Platform:** CodeSignal
- **Number of Questions:** 4 coding questions
- **Time Limit:** 70 minutes total
- **Scoring:**
  - Maximum score: 850 points
  - Passing threshold: 750+ points
  - Each question has multiple test cases
  - Partial credit given for passing some test cases
- **Difficulty:** Mix of Easy to Medium-Hard LeetCode-style problems
- **Note:** High passing bar - need to solve most questions completely or nearly completely

2.2 OA Question Examples

2.2.1 Array Divisibility Check

PROBLEM

Problem Statement:
Given an array of integers `arr` and an integer `k`, determine if you can partition the array into contiguous subarrays such that the sum of each subarray is divisible by `k`.
Return `true` if such a partition exists, `false` otherwise.
Example 1:

```
Input: arr = [3, 1, 2, 6, 4, 2], k = 3
Output: true
Explanation: [3] (sum=3, 3%3=0), [1,2] (sum=3, 3%3=0),
             [6] (sum=6, 6%3=0), [4,2] (sum=6, 6%3=0)
```

Example 2:

```
Input: arr = [1, 2, 3], k = 5
Output: false
Explanation: Total sum = 6, cannot partition into sums
             divisible by 5
```

Example 3:

```
Input: arr = [5, 10, 15], k = 5
Output: true
Explanation: Each element is divisible by 5:
             [5], [10], [15]
```

Constraints:

- $1 \leq \text{arr.length} \leq 10^5$
- $-10^9 \leq \text{arr[i]} \leq 10^9$
- $1 \leq k \leq 10^3$

SOLUTION

Solution Approach:
Key insight: If total sum is not divisible by k, impossible. Otherwise, use prefix sum modulo tracking:

- Calculate prefix sums modulo k
- A valid partition exists if we can split where each segment has sum $\equiv 0 \pmod{k}$
- Use hash map to track seen remainders
- Time: $O(N)$ where N = array length
- Space: $O(K)$ for remainder tracking

Python Solution:

```
def can_partition_divisible(arr, k):
    # First check: total sum must be divisible by k
    total = sum(arr)
    if total % k != 0:
        return False

    # Try to greedily partition
    current_sum = 0
    for num in arr:
        current_sum += num
        if current_sum % k == 0:
            current_sum = 0  # Start new partition

    # If we end with current_sum == 0, valid partition exists
    return current_sum == 0

# Alternative: Count remainders
def can_partition_divisible_v2(arr, k):
    if sum(arr) % k != 0:
        return False

    # Count prefix sum remainders
    prefix_sum = 0
    remainder_count = {0: 1}
```

```
    for num in arr:
        prefix_sum = (prefix_sum + num) % k
        # Handle negative modulo
        if prefix_sum < 0:
            prefix_sum += k

        if prefix_sum == 0:
            # Can partition here
            remainder_count = {0: 1}
        else:
            remainder_count[prefix_sum] = \
                remainder_count.get(prefix_sum, 0) + 1

    return True  # If we got here, partition exists

# Tests
print(can_partition_divisible([3,1,2,6,4,2], 3))  # True
print(can_partition_divisible([1,2,3], 5))        # False
print(can_partition_divisible([5,10,15], 5))      # True
```

TEST CASES & EDGE CASES

Edge Cases & Testing:

- Single element divisible by k: `[6], k=3` → `true`
- Single element not divisible: `[5], k=3` → `false`
- All elements divisible: each forms own partition
- Negative numbers: handle modulo correctly (`(-5) % 3 = 1` in Python)
- Zero in array: `[0, k]` always works
- Large k: k ¿ sum(arr)
- Total sum not divisible: early return false

Test Cases:

```
def test_array_divisibility():
    assert can_partition_divisible([3,1,2,6,4,2], 3) == True
    assert can_partition_divisible([1,2,3], 5) == False
    assert can_partition_divisible([5,10,15], 5) == True
    assert can_partition_divisible([1], 1) == True
    assert can_partition_divisible([2], 3) == False
    assert can_partition_divisible([-3,3], 3) == True
    assert can_partition_divisible([0,0,0], 5) == True

test_array_divisibility()
```

Interview Tips:

- Clarify if empty partitions are allowed
- Ask about negative number handling
- Discuss greedy vs DP approaches
- Mention total sum check optimization
- For Faire: Test with negative numbers and zeros

Source: CodeSignal OA (2024)

2.2.2 Digit Frequency Counter

PROBLEM

Problem Statement:
Given two integers `L` and `R` representing a range $[L, R]$, count the frequency of each digit (0-9) that appears in all numbers within this range (inclusive).
Return an array of length 10 where `result[i]` represents the count of digit `i`.
Example 1:

```
Input: L = 10, R = 12
Output: [1, 3, 1, 0, 0, 0, 0, 0, 0, 0]

Explanation:
Numbers: 10, 11, 12
10: digits 1, 0
11: digits 1, 1
12: digits 1, 2

Digit frequencies:
0: 1 time  (from 10)
1: 3 times (from 10, 11, 11, 12)
2: 1 time  (from 12)
```

Example 2:

```
Input: L = 1, R = 13
Output: [1, 6, 1, 1, 0, 0, 0, 0, 0, 0]

Numbers: 1,2,3,4,5,6,7,8,9,10,11,12,13
0: 1 (from 10)
1: 6 (from 1, 10, 11, 11, 12, 13)
2: 1 (from 2, 12)
3: 1 (from 3, 13)
...
```

Constraints:

- $0 \leq L \leq R \leq 10^9$

SOLUTION

Solution Approach:
Method 1: Brute Force (Small ranges)

- Iterate through each number in [L, R]
- Convert to string and count each digit
- Time: $O((R - L) \times \log R)$
- Space: $O(1)$
- Works well for small ranges

Method 2: Digit DP (Large ranges)

- Use digit dynamic programming
- count(R) - count(L-1) where count(n) = digits from 0 to n
- Time: $O(\log R)$
- More complex but handles large ranges

Python Solution:

```
def count_digit_frequency(L, R):
    # Brute force approach for reasonable ranges
    freq = [0] * 10

    for num in range(L, R + 1):
        # Count digits in current number
        for digit_char in str(num):
            digit = int(digit_char)
            freq[digit] += 1

    return freq

# Optimized for large ranges using helper function
def count_digits_upto(n):
    """Count digit frequencies from 0 to n"""
    if n < 0:
        return [0] * 10

    freq = [0] * 10
    for num in range(n + 1):
        for digit_char in str(num):
            freq[int(digit_char)] += 1
```

```
    return freq

def count_digit_frequency_optimized(L, R):
    # Count from 0 to R
    freq_R = count_digits_upto(R)
    # Count from 0 to L-1
    freq_L = count_digits_upto(L - 1)

    # Subtract to get range [L, R]
    result = [freq_R[i] - freq_L[i] for i in range(10)]
    return result

# Tests
print(count_digit_frequency(10, 12))
# [1, 3, 1, 0, 0, 0, 0, 0, 0, 0]

print(count_digit_frequency(1, 13))
# [1, 6, 1, 1, 0, 0, 0, 0, 0, 0] (approx)

print(count_digit_frequency(0, 10))
# Should count 0,1,2,...,10
```

TEST CASES & EDGE CASES

Edge Cases & Testing:

- Single number: L = R = 5 → only digit 5 appears once
- Range with 0: L = 0, R = 5
- Leading zeros: don't count (10 has digits 1 and 0, not 0, 1, 0)
- Large numbers: 10^9 range
- Consecutive numbers: [99, 101] crosses 100
- All same digit: [111, 111] → three 1's

Test Cases:

```
def test_digit_frequency():
    # Test 1: Example from problem
    assert count_digit_frequency(10, 12) == \
        [1, 3, 1, 0, 0, 0, 0, 0, 0, 0]

    # Test 2: Single number
    assert count_digit_frequency(5, 5) == \
        [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]

    # Test 3: Include zero
```

```
    assert count_digit_frequency(0, 1)[0] == 1
    assert count_digit_frequency(0, 1)[1] == 1

    # Test 4: Range crossing hundreds
    result = count_digit_frequency(99, 101)
    assert result[0] == 2  # From 100, 100
    assert result[1] == 4  # From 99, 100, 100, 101
    assert result[9] == 2  # From 99, 99

test_digit_frequency()
```

Interview Tips:

- Clarify if leading zeros count (they don't for numbers)
- Discuss time complexity for different range sizes
- Mention digit DP for very large ranges
- Ask about memory constraints
- For Faire: Test boundary cases like 0, 999, 1000

Source: CodeSignal OA (2024)

2.2.3 Ads Assortment Problem

PROBLEM

Problem Statement:
You are given `n` advertisements, where each ad has a `value` (revenue generated) and a `cost` (budget required). You have a total budget of `B`.
Select a subset of ads to maximize total value while staying within budget.
Additionally, some ads may have **category constraints**: You can only select at most `k` ads from each category.
Example 1:

```
Input:
ads = [
  {'value': 60, 'cost': 10, 'category': 'tech'},
  {'value': 100, 'cost': 20, 'category': 'fashion'},
  {'value': 120, 'cost': 30, 'category': 'tech'}
]
budget = 50
category_limit = 2  # Max 2 ads per category

Output: 220
Explanation: Select ads 2 and 3 (fashion + tech)
             Cost: 20 + 30 = 50, Value: 100 + 120 = 220
```

Example 2:

```
Input:
ads = [
  {'value': 60, 'cost': 10},
  {'value': 100, 'cost': 20},
  {'value': 120, 'cost': 30}
]
budget = 35

Output: 160
Explanation: Select ads 1 and 2.
             Cost: 10 + 20 = 30, Value: 60 + 100 = 160
```

Constraints:

- $1 \leq n \leq 100$
- $1 \leq \text{cost[i]}, \text{value[i]} \leq 1000$
- $1 \leq B \leq 10000$

SOLUTION

Solution Approach:
Method 1: Dynamic Programming (0/1 Knapsack)

- Classic 0/1 knapsack problem
- dp[i][j] = max value using first i ads with budget j
- Time: $O(N \times B)$
- Space: $O(N \times B)$, optimizable to $O(B)$

Method 2: Greedy (with category constraints)

- Sort ads by value/cost ratio (efficiency)
- Greedily select highest efficiency ads
- Track category counts
- Time: $O(N \log N)$
- Note: May not give optimal solution, but good approximation

Python Solution:

```
def max_ad_value_knapsack(ads, budget):
    """0/1 Knapsack DP solution"""
    n = len(ads)
    # dp[i][b] = max value using first i ads with budget b
```

```
    dp = [[0] * (budget + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        cost = ads[i-1]['cost']
        value = ads[i-1]['value']

        for b in range(budget + 1):
            # Don't take current ad
            dp[i][b] = dp[i-1][b]

            # Take current ad if possible
            if b >= cost:
                dp[i][b] = max(dp[i][b], dp[i-1][b-cost] + value)

    return dp[n][budget]

# Space-optimized version
def max_ad_value_optimized(ads, budget):
    """Space-optimized O(B) space"""
    dp = [0] * (budget + 1)

    for ad in ads:
        cost = ad['cost']
        value = ad['value']

        # Traverse backwards to avoid using same item twice
        for b in range(budget, cost - 1, -1):
            dp[b] = max(dp[b], dp[b - cost] + value)

    return dp[budget]

# With category constraints
def max_ad_value_with_categories(ads, budget, category_limit):
    """DP with category constraints"""
    from collections import defaultdict

    # Group ads by category
    categories = defaultdict(list)
    for i, ad in enumerate(ads):
        cat = ad.get('category', 'default')
        categories[cat].append(i)

    # Generate valid ad combinations respecting category limits
    # For simplicity, use greedy approach
    ads_sorted = sorted(enumerate(ads),
                        key=lambda x: x[1]['value'] / x[1]['cost'],
```

```
                        reverse=True)

    category_count = defaultdict(int)
    selected = []
    total_cost = 0
    total_value = 0

    for idx, ad in ads_sorted:
        cat = ad.get('category', 'default')
        if (category_count[cat] < category_limit and
            total_cost + ad['cost'] <= budget):
            selected.append(idx)
            category_count[cat] += 1
            total_cost += ad['cost']
            total_value += ad['value']

    return total_value

# Tests
ads1 = [
    {'value': 60, 'cost': 10},
    {'value': 100, 'cost': 20},
    {'value': 120, 'cost': 30}
]
print(max_ad_value_knapsack(ads1, 35))  # 160

ads2 = [
    {'value': 60, 'cost': 10, 'category': 'tech'},
    {'value': 100, 'cost': 20, 'category': 'fashion'},
    {'value': 120, 'cost': 30, 'category': 'tech'}
]
print(max_ad_value_with_categories(ads2, 50, 2))  # 220
```

TEST CASES & EDGE CASES

Edge Cases & Testing:

- Budget = 0: return 0
- Single ad within budget: return its value
- Single ad exceeds budget: return 0
- All ads exceed budget: return 0
- Category limit = 0: cannot select any ads
- Multiple ads same efficiency: test tie-breaking

- Exact budget match: select ads that sum to exactly B

Test Cases:

```
def test_ads_assortment():
    # Test 1: Basic knapsack
    ads = [{'value': 60, 'cost': 10}, {'value': 100, 'cost': 20}]
    assert max_ad_value_knapsack(ads, 25) == 100

    # Test 2: All ads fit
    ads = [{'value': 10, 'cost': 5}, {'value': 20, 'cost': 10}]
    assert max_ad_value_knapsack(ads, 15) == 30

    # Test 3: No ads fit
    ads = [{'value': 100, 'cost': 50}]
    assert max_ad_value_knapsack(ads, 40) == 0

    # Test 4: Empty ads
    assert max_ad_value_knapsack([], 100) == 0

    # Test 5: Zero budget
    ads = [{'value': 100, 'cost': 10}]
    assert max_ad_value_knapsack(ads, 0) == 0

test_ads_assortment()
```

Interview Tips:

- Recognize as 0/1 knapsack variant
- Discuss trade-offs: DP vs greedy
- Ask about fractional ads (0/1 vs fractional knapsack)
- Clarify category constraint details
- For Faire: Test boundary conditions

Source: CodeSignal OA (2024)

2.2.4 General OA Topics

- Array manipulation and subarrays
- String processing and pattern matching
- Hash maps and frequency counting
- Greedy algorithms
- Dynamic programming (basic)
- Math and number theory (modular arithmetic)

3 System Design Questions

3.1 Design a Post "Like" Functionality

- **Description:** Design a system for liking posts (similar to social media)
- **Focus:** Communication and thought process
- **Key aspects to consider:**
  - How to think about the problem
  - Different solution approaches
  - Scalability considerations
- **Note:** More emphasis on communication than specific implementation
- **Position:** Backend Engineer
- **Source:** System Design round (2024-2025)

4 Data Engineering Questions

4.1 Senior Data Engineer Interview

- **Interview Structure:**
  - 1 round phone screen: Coding
  - 3 rounds virtual onsite:
    1. Coding
    2. Behavioral Questions (BQ)
    3. Pipeline Design
- **Timeline:** August application, late August recruiter reach out
- **Result:** Rejection
- **Feedback:** Very difficult for job-hopping while employed; low error tolerance
- **Source:** Senior Data Engineer interview (2025)

5 Frontend/Full Stack Specific

5.1 Object Manipulation (Frontend Intern)

PROBLEM

Problem Statement:
You are building a product catalog system for an e-commerce platform. Implement a `ProductCatalog` class that stores products and supports various filtering operations.
Each product has the following attributes:

- `id`: unique identifier (string)

- `name`: product name (string)
- `price`: product price (float)
- `category`: product category (string)
- `attributes`: dictionary of additional attributes (e.g., {"size": "M", "color": "blue"})
- `in_stock`: whether product is in stock (boolean)

Implement the following methods:

1. `add_product(product)`: Add a product to the catalog
2. `get_product(product_id)`: Retrieve a product by ID
3. `filter_by_category(category)`: Return all products in a category
4. `filter_by_price_range(min_price, max_price)`: Return products in price range
5. `filter_by_attributes(attr_dict)`: Return products matching all given attributes
6. `search_by_name(query)`: Return products with name containing query (case-insensitive)
7. `get_available_products()`: Return all in-stock products

Example:

```
catalog = ProductCatalog()

catalog.add_product({
    'id': 'p1', 'name': 'Blue T-Shirt', 'price': 19.99,
    'category': 'clothing', 'attributes': {'size': 'M', 'color': 'blue'},
    'in_stock': True
})

catalog.add_product({
    'id': 'p2', 'name': 'Red T-Shirt', 'price': 24.99,
    'category': 'clothing', 'attributes': {'size': 'L', 'color': 'red'},
    'in_stock': False
})

# Filter by category
catalog.filter_by_category('clothing')  # Returns [p1, p2]

# Filter by attributes
catalog.filter_by_attributes({'color': 'blue'})  # Returns [p1]

# Get available products
catalog.get_available_products()  # Returns [p1]
```

Constraints:

- $1 \leq$ number of products $\leq 10^4$
- Product IDs are unique
- $0 \leq$ price $\leq 10^6$
- Attribute keys and values are non-empty strings

SOLUTION

Approach:
Use a dictionary to store products by ID for O(1) lookup. For filtering operations, iterate through products and check conditions. Can optimize with indexes for common queries.
Time Complexity:

- add_product: $O(1)$
- get_product: $O(1)$
- filter operations: $O(N)$ where N = number of products

Space Complexity: $O(N)$ for storing N products
Python Solution:

```
class ProductCatalog:
    def __init__(self):
        self.products = {}  # id -> product mapping

    def add_product(self, product):
        """Add a product to the catalog"""
        if 'id' not in product:
            raise ValueError("Product must have an id")

        product_id = product['id']
        self.products[product_id] = product

    def get_product(self, product_id):
        """Get product by ID"""
        return self.products.get(product_id, None)

    def filter_by_category(self, category):
        """Return all products in a category"""
        result = []
        for product in self.products.values():
            if product.get('category') == category:
                result.append(product)
        return result
```

```
    def filter_by_price_range(self, min_price, max_price):
        """Return products within price range [min_price, max_price]"""
        result = []
        for product in self.products.values():
            price = product.get('price', 0)
            if min_price <= price <= max_price:
                result.append(product)
        return result

    def filter_by_attributes(self, attr_dict):
        """Return products matching all given attributes"""
        result = []
        for product in self.products.values():
            product_attrs = product.get('attributes', {})

            # Check if all attributes match
            match = True
            for key, value in attr_dict.items():
                if product_attrs.get(key) != value:
                    match = False
                    break

            if match:
                result.append(product)

        return result

    def search_by_name(self, query):
        """Return products with name containing query (case-insensitive)"""
        result = []
        query_lower = query.lower()

        for product in self.products.values():
            name = product.get('name', '').lower()
            if query_lower in name:
                result.append(product)

        return result

    def get_available_products(self):
        """Return all in-stock products"""
        result = []
        for product in self.products.values():
            if product.get('in_stock', False):
                result.append(product)
        return result
```

```
    def remove_product(self, product_id):
        """Remove a product from catalog"""
        if product_id in self.products:
            del self.products[product_id]
            return True
        return False

    def update_stock(self, product_id, in_stock):
        """Update stock status"""
        if product_id in self.products:
            self.products[product_id]['in_stock'] = in_stock
            return True
        return False


# Optimized version with indexes
class OptimizedProductCatalog:
    def __init__(self):
        self.products = {}
        self.category_index = {}  # category -> [product_ids]
        self.stock_index = {'in_stock': set(), 'out_of_stock': set()}

    def add_product(self, product):
        product_id = product['id']
        self.products[product_id] = product

        # Update category index
        category = product.get('category')
        if category:
            if category not in self.category_index:
                self.category_index[category] = set()
            self.category_index[category].add(product_id)

        # Update stock index
        if product.get('in_stock', False):
            self.stock_index['in_stock'].add(product_id)
        else:
            self.stock_index['out_of_stock'].add(product_id)

    def filter_by_category(self, category):
        """O(K) where K is products in category"""
        product_ids = self.category_index.get(category, set())
        return [self.products[pid] for pid in product_ids]

    def get_available_products(self):
        """O(K) where K is in-stock products"""
```

```
        return [self.products[pid] for pid in self.stock_index['in_stock']]
```

TEST CASES & EDGE CASES

Edge Cases & Testing:

- Empty catalog: all filters return empty list
- Product with missing attributes: handle gracefully
- Duplicate product IDs: overwrite or reject
- Case sensitivity in search: ensure case-insensitive
- Price = 0 or negative: validate input
- Empty search query: return all or none
- Attribute value is None or empty string
- Multiple filters combined (AND vs OR logic)

Test Cases:

```
def test_product_catalog():
    catalog = ProductCatalog()

    # Test 1: Add and retrieve
    p1 = {'id': 'p1', 'name': 'Shirt', 'price': 20,
          'category': 'clothing', 'in_stock': True,
          'attributes': {'size': 'M', 'color': 'blue'}}
    catalog.add_product(p1)
    assert catalog.get_product('p1') == p1

    # Test 2: Filter by category
    p2 = {'id': 'p2', 'name': 'Pants', 'price': 30,
          'category': 'clothing', 'in_stock': True,
          'attributes': {}}
    catalog.add_product(p2)
    assert len(catalog.filter_by_category('clothing')) == 2

    # Test 3: Price range filter
    results = catalog.filter_by_price_range(15, 25)
    assert len(results) == 1
    assert results[0]['id'] == 'p1'

    # Test 4: Attribute filter
    results = catalog.filter_by_attributes({'color': 'blue'})
    assert len(results) == 1
    assert results[0]['id'] == 'p1'
```

```
    # Test 5: Search by name
    results = catalog.search_by_name('shirt')
    assert len(results) == 1

    # Test 6: Available products
    p3 = {'id': 'p3', 'name': 'Hat', 'price': 15,
          'category': 'accessories', 'in_stock': False,
          'attributes': {}}
    catalog.add_product(p3)
    results = catalog.get_available_products()
    assert len(results) == 2  # p1 and p2

    # Test 7: Empty results
    assert catalog.filter_by_category('shoes') == []
    assert catalog.filter_by_attributes({'size': 'XL'}) == []

    print("All tests passed!")

test_product_catalog()
```

Interview Tips:

- Ask about performance requirements (optimize with indexes?)
- Clarify attribute matching logic (AND vs OR)
- Discuss trade-offs: simple iteration vs indexed lookups
- Consider using list comprehensions for cleaner code
- Validate input data (missing fields, invalid types)

Position: Frontend Intern — Duration: 1 hour — Source: Intern interview (2023)

6 Behavioral Interview Questions

Common behavioral questions asked at Faire (compiled from Prepfully and 1point3acres):

1. Tell me about yourself and your background
2. Why do you want to work at Faire? What interests you about the company?
3. Describe a time when you had to work with a difficult team member. How did you handle it?
4. Tell me about a challenging technical problem you solved. What was your approach?
5. How do you handle disagreements with your manager or team members?
6. Describe a project you're most proud of. What was your role?
7. Tell me about a time you failed. What did you learn?

8. How do you prioritize tasks when you have multiple deadlines?
9. Describe a situation where you had to learn a new technology quickly
10. Where do you see yourself in 3-5 years?

6.1 Tips for Behavioral Rounds

- Use STAR method (Situation, Task, Action, Result)
- Prepare stories that demonstrate:
  - Technical problem-solving
  - Teamwork and collaboration
  - Leadership and ownership
  - Adaptability and learning
  - Communication skills
- Research Faire's business model (wholesale marketplace connecting retailers and brands)
- Be prepared to discuss why you're interested in B2B marketplace space
- Demonstrate understanding of Faire's mission to empower small businesses

7 Key Interview Tips

7.1 Test Cases and Edge Cases

- Faire interviewers place **heavy emphasis** on test cases
- Be prepared to discuss many edge cases
- Consider using testing frameworks (e.g., JUnit) during the interview
- Even comprehensive testing may not be enough - be ready to think of more cases

7.2 Recent Trends (2024-2025)

- Low error tolerance - even mostly working code may result in rejection
- Difficult for candidates job-hopping while employed due to time constraints
- Senior level may not include BQ or system design in some cases (varies by position)

8 Additional Notes

- **Company:** Faire is a wholesale marketplace (Canadian company with US presence)
- **Interview Difficulty:** Generally rated as "Hard" by candidates
- **Response Time:** Fast - usually hear back same day or next day after each round
- **Process:** HR is responsive and provides feedback

Disclaimer

This information is compiled from multiple public sources including 1point3acres.com, Prepfully.com, and web search results. Actual interview content may vary. Some details on 1point3acres.com were hidden behind paywalls and may not be completely comprehensive. Use this as a reference for preparation, but be prepared for variations in actual interviews.

Sources:

- 1point3acres Faire tag (3 pages): `https://www.1point3acres.com/bbs/tag/faire-7100-1.html`
- Prepfully Faire Interviews: `https://prepfully.com/interview-guides/faire`
- Various web search results for CodeSignal OA format and interview structure

Data collected: November 2025
Document compiled from 10+ interview experiences across multiple sources