Faire Interview Questions
Backend Software Engineer

Compiled from 1point3acres.com

November 8, 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 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: $O(N \cdot K)$

Python 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())
```

Edge Cases & Testing:

- 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 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

Python 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)
```

Edge Cases & Testing:

- 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 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

Python 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
```

Edge Cases & Testing:

- 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

- **Description:** Funnel-shaped problem (specific details not publicly available)
- **Key Point:** Interviewer heavily focused on test cases and edge cases
- **Note:** Candidate mentioned using JUnit for testing
- **Feedback:** Even after comprehensive testing, interviewer kept asking for more edge cases
- **Source:** Coding interview (2024-2025)

1.1.5 Number to Word Conversion

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 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
```

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 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 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
```

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:** LeetCode 485 - Max Consecutive Ones
- **Duration:** 45 minutes
- **Source:** Coding interview (2024-2025)

1.4 Additional Coding Problems

1.4.1 Date Format Function

- **Description:** Implement a function to parse and format dates
- **Details:**
  - Convert dates between different formats (e.g., "MM/DD/YYYY" $\leftrightarrow$ "YYYY-MM-DD")
  - Handle edge cases: invalid dates, leap years, different delimiters
  - Validate date inputs
- **Position:** Frontend/Full Stack
- **Source:** Frontend interview (2024)

1.4.2 Maze with Portals

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 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))
```

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 $\rightarrow$ B, B $\rightarrow$ 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

- **Description:** Parse a string with specific format rules and extract information
- **Details:** Limited information available; involves pattern matching and string manipulation
- **Position:** Backend Engineer
- **Source:** 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 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 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
```

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 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 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
```

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

- **Description:** Optimize selection of advertisements to maximize value given constraints
- **Details:**
  - Multiple ads with associated values and constraints
  - Budget/capacity limitations
  - Need to maximize total value
- **Similar to:** Knapsack-variant or greedy selection problem
- **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)

- **Description:** Write operations on product objects using Python or JavaScript
- **Details:**
  - Objects represent products with attributes (size, color, etc.)
  - Operations: search/filter products by attributes
- **Duration:** 1 hour
- **Position:** Frontend Intern
- **Location:** Canada
- **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