Real Life Interview Questions asked in OA’s

1. Amazon

A screenshot of a computer

Description automatically generated

**Transcript of the Question**

**Code Question 2:**

Data analysts at Amazon are analyzing time-series data. It was concluded that the data of the nth item was dependent on the data of some xth day if there is a positive integer k such that:

\text{floor}(n / k) = x

where \text{floor}(z) represents the largest integer less than or equal to z.

**Task**:

Given n, find the **sum of all the days’ numbers (x)** on which the data of the xth day (0 \leq x \leq n) will be dependent.

**Example**:

A white sheet with black text

Description automatically generated A table with numbers and equations

Description automatically generated

Hence the answer is 0 + 1 + 2 + 5 = 8.

**Function Description**:

Complete the function getDataDependenceSum in the editor below.

getDataDependenceSum takes the following arguments:

• n: An integer representing the given value.

**Output**:

The function should return an integer representing the sum of all the dependent days x.

**Python Solution**

Here is the most efficient Python implementation for this problem:

def getDataDependenceSum(n):

dependent\_days = set() # To store unique x values

for k in range(1, n + 1):

x = n // k # Calculate floor(n / k)

if 0 <= x <= n:

dependent\_days.add(x) # Add x to the set

return sum(dependent\_days) # Sum of all unique x values

**Explanation:**

1. Loop through all possible values of k from 1 to n.

2. For each k, calculate x = \text{floor}(n / k).

3. If x is within the range [0, n], add x to a set to ensure only unique values are stored.

4. Finally, return the sum of all unique x values.

**Complexity:**

• **Time Complexity**: O(n), as the loop iterates up to n.

• **Space Complexity**: O(n), for storing unique x values in the set.

This solution efficiently handles the constraints of the problem.

ABOVE ONE GAVE TLE, BELOW IS MORE OPTIMIZED:

To optimize further, we can reduce unnecessary iterations by observing the properties of the division operation. Specifically, instead of iterating over all possible k, we can iterate over ranges of x, taking advantage of the fact that x = \lfloor n / k \rfloor produces the same result for consecutive values of k within certain ranges.

def getDataDependenceSum(n):

result = 0

x = n

while x > 0:

# Find the range of k values for which floor(n / k) = x

# k\_start is the smallest k for this x

k\_start = n // (x + 1) + 1

# k\_end is the largest k for this x

k\_end = n // x

# Add x to the result

result += x

# Decrease x to the next possible value

x -= 1

return result

**Explanation of Optimization:**

1. Instead of iterating through all values of k, this approach focuses on unique x values generated by x = \lfloor n / k \rfloor.

2. Each value of x corresponds to a range of k values where x = \lfloor n / k \rfloor is constant.

3. By iterating over x and directly calculating the corresponding k range using:

• k\_{\text{start}} = n // (x + 1) + 1

• k\_{\text{end}} = n // x

we avoid redundant calculations for each k.

4. The sum of all x values is computed directly without iterating over all k.

**Complexity:**

• **Time Complexity**: O(\sqrt{n}), because the number of unique values of x is proportional to \sqrt{n} due to the division operation.

• **Space Complexity**: O(1), since no additional data structures are used.

This approach avoids TLE for larger inputs and is much more efficient than the earlier solution.

Solution 2:-

Below is a well-known approach that computes the sum of the distinct values of ⌊n/k⌋ in roughly O(√n) time (rather than O(n)) by jumping over ranges of k that yield the same quotient.

**Key Insight**

• For a given quotient q = n // k, all integers in the range k to r = n // q produce the *same* value q.

• Therefore, once we know q = n // k, we can jump k directly to r + 1 instead of incrementing by 1 each time.

• This way, each distinct quotient is encountered **exactly once**.

Here’s an optimized version in Python:

def getDataDependenceSum\_optimized(n):

total\_sum = 0

k = 1

while k <= n:

q = n // k # This is the current quotient

r = n // q # The largest k for which n//k = q

total\_sum += q # Add this distinct quotient

k = r + 1 # Skip all k's that give the same quotient

return total\_sum

**How It Works**

1. Start at k = 1.

2. Compute q = n // k. This is the current distinct floor-division result.

3. Find the maximum r such that for all k' in [k, r], n // k' = q.

• That r will be n // q.

4. Add q to our sum because q is a new, distinct result of the division.

5. Jump k to r + 1, thereby skipping all the k' in [k, r] which we already accounted for.

6. Continue until k > n.

By doing this, instead of iterating through **every** k from 1 to n, we make potentially large jumps, especially once k gets beyond √n. This yields an **O(√n)** overall performance instead of O(n).

**Example**

For **n = 6**:

• **Iteration 1**:

• k = 1

• q = 6 // 1 = 6

• r = 6 // 6 = 1

• Add 6 to the sum.

• Advance k to r + 1 = 2.

• **Iteration 2**:

• k = 2

• q = 6 // 2 = 3

• r = 6 // 3 = 2

• Add 3 to the sum (sum so far = 6 + 3 = 9).

• Advance k to r + 1 = 3.

• **Iteration 3**:

• k = 3

• q = 6 // 3 = 2

• r = 6 // 2 = 3

• Add 2 (sum = 11).

• Advance k to 4.

• **Iteration 4**:

• k = 4

• q = 6 // 4 = 1

• r = 6 // 1 = 6

• Add 1 (sum = 12).

• Advance k to 7, which ends the loop (k > n).

The distinct values of ⌊6/k⌋ for k = 1..6 are {6, 3, 2, 1}, and their sum is indeed 12.

This method provides the same result as your original function but runs much faster when n is large.

1. Amazon

**Code Question 1**

Amazon is developing an efficient string matching library. Develop a prototype service that matches a simple pattern with a text. There are two arrays of strings, text and wild, each of size n. Each string in wild is a regex expression that contains exactly one wildcard character (\*).

A wildcard character (\*) matches any sequence of zero or more lowercase English letters. A regex matches some string if it is possible to replace the wildcard character with some sequence of characters such that the regex expression becomes equal to the string. No other character can be changed. For example, regex "abc\*bcd" matches "abcbd", "abcefgbcd", and "abcccbcd", whereas it does not match the strings "abcbd", "abzbcbd", "abcd".

For every i from 1 to n, your task is to find out whether wild[i] matches text[i]. Return the answer as an array of strings of size n where the i-th string is "YES" if wild[i] matches text[i], and "NO" otherwise.

**Note**: The implementation shall not use any in-built regex libraries.

**Example**:

Given n = 2,

text = ["code", "coder"],

wild = ["co\*d", "co\*er"].

• text[0] = "code", wild[0] = "co\*d" → "NO", the suffixes do not match.

• text[1] = "coder", wild[1] = "co\*er" → "YES", the prefixes and suffixes match.

**Output**: ["NO", "YES"]

**Function Description**:

Complete the function findMatching in the editor below.

**findMatching** has the following parameters:

• STRING\_ARRAY text: An array of strings to be matched.

• STRING\_ARRAY wild: An array of regex patterns, each containing one wildcard character \*.

**Return**:

• STRING\_ARRAY: Return an array of "YES" or "NO" strings based on whether the wild[i] matches text[i].

Solution

def findMatching(text, wild):

res = []

for t, w in zip(text, wild):

# Split the wild string at the wildcard '\*'

prefix, suffix = w.split('\*')

# Check if the text matches the prefix and suffix correctly

if t.startswith(prefix) and t.endswith(suffix) and len(prefix) + len(suffix) <= len(t):

res.append("YES")

else:

res.append("NO")

return res

**Explanation of Fix:**

1. **Length Constraint**:

• After matching the prefix and suffix, ensure that their combined length (len(prefix) + len(suffix)) does not exceed the length of the string t. If it does, the wildcard \* cannot span the remaining characters, so the match should fail.

2. **Logic**:

• The startswith method ensures the prefix matches the start of t.

• The endswith method ensures the suffix matches the end of t.

• Adding the length check ensures that the \* wildcard can match the middle part of the string.

**Debugging This Case:**

For the input:

text = ["abefcd", "adehgfg", "gfgfg"]

wild = ["ab\*cd", "ade\*f", "gfg\*gfg"]

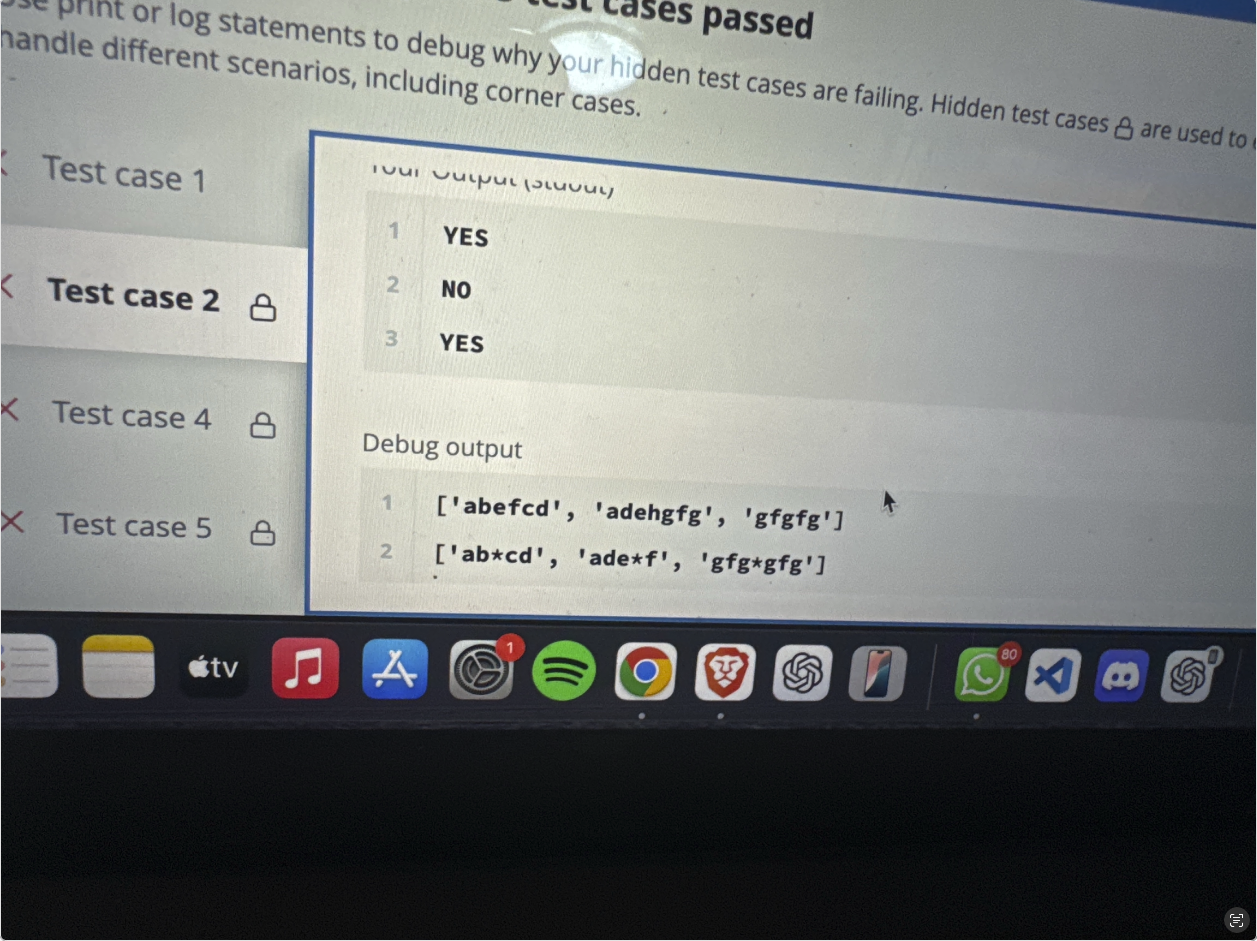
**Expected Output**: ["YES", "NO", "NO"]

**Explanation**:

1. "ab\*cd" matches "abefcd" because the prefix "ab" and suffix "cd" match, and the wildcard spans "ef". → "YES"

2. "ade\*f" does not match "adehgfg" because the suffix "f" does not match the end of "adehgfg". → "NO"

3. "gfg\*gfg" does not match "gfgfg" because the combined length of the prefix "gfg" and suffix "gfg" exceeds the length of the string "gfgfg". → "NO"

One of the cases it was failing at first “”