## 2821. Delay the Resolution of Each Promise

**Easy** 

## **Problem Description**

The problem involves an array called functions that contains several functions. Each function, when called, will return a promise. A promise in JavaScript is an object representing the completion (or failure) of an asynchronous operation. Along with the functions array, you are also given a number ms, which represents the number of milliseconds of delay to introduce before each promise resolves. Your task is to return a new array of functions so that when each function is called, it will return a promise that only resolves after

waiting for the duration specified by ms. This delay should be applied individually to each of the promises returned by the functions in the new array. Each promise must preserve the order in which the original functions from the functions array were called. The goal is to create a new array of functions in such a way that if you call the functions in sequence, each function will delay its

execution by the specified ms milliseconds before resolving its promise, ensuring that the asynchronous operations are executed with a consistent delay between them. Intuition

#### To solve this problem, we leverage the map function in JavaScript, which allows us to transform each element in an array using a transformation function. In this context, each element in the functions array is a function that returns a promise.

By using map, we can iterate over each function in functions, and create a corresponding function in the new array that, when

called, initiates a delay before calling the original function (fn). After the delay, the original function should be called, and the

Here's how we arrive at the solution approach: 1. Use map to iterate over each function in the functions array. 2. For each function (fn), create a new async function that:

 Waits for the specified ms milliseconds using setTimeout wrapped in a promise. This is accomplished with await new Promise(resolve => setTimeout(resolve, ms));. The await keyword pauses the function execution until the timeout completes.

Calls the original function fn() to get the promise it was supposed to return.

meaning it will return a Promise and allow the use of await inside its body.

scheduled to call resolve() after ms milliseconds, resolving the promise.

a new array. This new array is then returned by the delayAll function.

Let's consider an example to illustrate the solution approach with a simple scenario:

promise it returns should be the result of the new function.

Here is a step-by-step explanation of the code:

which implicitly returns a promise.

- Returns the promise from calling fn() so that the returned value matches the intended asynchronous action of the original function. By defining the new function as async, we are indicating that it is an asynchronous function that implicitly returns a promise,
- **Solution Approach**

which can be awaited. This aligns with the requirement that each function in the new array should return a promise as well.

The implementation of the solution involves using a higher-order function, map, and leveraging the async/await syntax of JavaScript to handle the timing of the promise resolution.

The map function is a method available on arrays in JavaScript, which creates a new array with the results of calling a provided

#### function on every element in the calling array. The provided function should take each element from the original array (in this case, functions), apply a transformation, and return the new value that will be included in the new array.

In our case, we're using it to iterate over each function in the functions array and create a new function that introduces a delay before execution. This is our transformation. The async keyword is used to define the returning function as asynchronous,

We call map on the original functions array. map takes a function as its argument, which will be executed on each element (fn) of functions. Inside this function, we return a new async function. This ensures that the returned function is an asynchronous function,

Within the new async function, the first operation is await new Promise(resolve => setTimeout(resolve, ms));. This creates a promise that resolves after a delay of ms milliseconds. • new Promise(resolve => setTimeout(resolve, ms)) creates a new promise that will execute the setTimeout function. setTimeout is

Once the delay has finished, the original function fn is called. We return fn() which calls the original function and returns its

function's promise. Each new function created by the map loop now includes the delay as described, and the resulting functions are collected into

promise. Since we're in an async function, the result of fn() is returned as a resolved value of the outer asynchronous

By using await, we're telling JavaScript to wait until the promise is resolved (after the delay) before moving on to the next line.

The result is an array of functions that match the behavior of the original functions but with each of their promises resolving after a delayed interval specified by ms. This solution is elegant and cleanly utilizes JavaScript's handling of asynchronicity to enforce a simple yet crucial requirement—the delay between promise resolutions.

the array. We are given a delay ms of 1000 milliseconds, which means we want to introduce a 1-second delay before each promise resolves. // Original functions array let functions = [

Suppose we have an array of functions functions, where each function, when called, simply resolves a promise with its index in

### To apply the solution approach, we will follow these steps:

return async () => {

// Step 4: Introduce delay

Solution Implementation

**Python** 

() => Promise.resolve('Function 1'),

() => Promise resolve('Function 2'),

// Apply transformation to introduce delay

let ms = 1000; // Delay of 1000 milliseconds

() => Promise.resolve('Function 3')

**Example Walkthrough** 

let delayedFunctions = functions.map((fn, index) => { // Step 2: Return the new async function

delayedFunctions[1]().then(console.log); // After 1 more second, logs: "Function 2"

delayedFunctions[2]().then(console.log); // After 1 more second, logs: "Function 3"

functions, but with the additional specified delay before resolution.

Wraps each function in a given list with a delay.

:param functions\_array: The list of functions to be delayed.

:param original\_function: The function to be delayed.

# Map each original function to its delayed counterpart

return [delayed\_function(func) for func in functions\_array]

The delayed functions, when invoked,

:return: A list of delayed functions.

:return: The delayed function.

async def wrapper(\*args, \*\*kwargs):

def delay\_all(functions\_array: List[Callable], delay\_ms: int) -> List[Callable]:

will wait for the specified milliseconds before executing the original function.

:param delay\_ms: The number of milliseconds to delay the function execution.

Returns an async function that delays the given original function.

# Delay the execution by sleeping for the specified amount of time

await asyncio.sleep(delay\_ms / 1000) # asyncio.sleep uses seconds

// Create an empty vector of std::function objects to hold the delayed functions

std::vector<std::function<void()>> delayedFunctions;

async def delayed\_function(original\_function: Callable) -> Callable:

1. We create a new array, delayedFunctions, by mapping over each function in functions.

```
await new Promise(resolve => setTimeout(resolve, ms));
    // Step 5: Call original function
    return fn();
 2. Each element in delayedFunctions is now an async function that will incorporate the delay before calling the original function:
// Call each function in sequence with the delay
delayedFunctions[0]().then(console.log); // After 1 second, logs: "Function 1"
```

By calling the functions in sequence, we observe that each function waits for 1 second (the value of ms) before resolving its

solution approach works as intended, with each new function in delayedFunctions behaving exactly like its counterpart in

promise. The output is logged to the console with a consistent 1-second delay between each message. This confirms that the

from typing import Callable, List import asyncio

```
# Invoke the original function after the delay
    return original_function(*args, **kwargs)
return wrapper
```

import java.util.concurrent.\*;

import java.util.function.\*;

int delayMs) {

Java

```
import java.util.List;
import java.util.stream.Collectors;
public class DelayFunctions {
    /**
    * Wraps each function in a given list with a delay.
     * The delayed functions, when invoked, will wait for the specified milliseconds before executing the original function.
     * @param functionsList The list of functions to be delayed.
                            The number of milliseconds to delay the function execution.
     * @param delayMs
    * @return A list of delayed functions.
    public static List<Supplier<Future<Object>>> delayAll(List<Supplier<Object>>> functionsList, int delayMs) {
        ExecutorService executor = Executors.newFixedThreadPool(functionsList.size());
       // Map each original function to a new function that has a delay
       return functionsList.stream().map(originalFunction -> (Supplier<Future<Object>>) () -> {
           // Return a Future task
            return executor.submit(() -> {
                // Delay the execution by waiting for the specified amount of time
                try {
                    TimeUnit.MILLISECONDS.sleep(delayMs);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt(); // Restore the interrupted status
                    // Handle the interrupted exception
                // Invoke the original function after the delay
                return originalFunction.get();
            });
        }).collect(Collectors.toList());
C++
#include <functional>
#include <vector>
#include <chrono>
#include <thread>
/**
* Wraps each function in a given vector with a delay.
* The delayed functions, when invoked,
 * will wait for the specified duration in milliseconds before executing
 * the original function.
 * @param functionsVector The vector of functions to be delayed.
 * @param delayMs The number of milliseconds to delay the function execution.
 * @return A vector of delayed functions.
*/
std::vector<std::function<void()>> delayAll(
    const std::vector<std::function<void()>>& functionsVector,
```

```
// Iterate over the functions in the input vector
      for (const auto& originalFunction : functionsVector) {
          // Create a new function with a delay
          std::function<void()> delayedFunction = [originalFunction, delayMs]() {
              // Create a duration object for the delay
              std::chrono::milliseconds duration(delayMs);
              // Put the current thread to sleep for the time specified by duration
              std::this_thread::sleep_for(duration);
              // After the delay, call the original function
              originalFunction();
          };
          // Add the delayed function to the vector of delayed functions
          delayedFunctions.push_back(delayedFunction);
      // Return the vector containing the delayed functions
      return delayedFunctions;
  TypeScript
  /**
   * Wraps each function in a given array with a delay.
   * The delayed functions, when invoked,
   * will wait for the specified milliseconds before executing the original function.
   * @param {Function[]} functionsArray - The array of functions to be delayed.
   * @param {number} delayMs - The number of milliseconds to delay the function execution.
   * @returns {Function[]} An array of delayed functions.
  function delayAll(functionsArray: Function[], delayMs: number): Function[] {
      return functionsArray.map(originalFunction => {
          // Return a new async function
          return async function delayedFunction(): Promise<any> {
              // Delay the execution by waiting for the specified amount of time
              await new Promise<void>(resolve => setTimeout(resolve, delayMs));
              // Invoke the original function after the delay
              return originalFunction();
      });
from typing import Callable, List
import asyncio
def delay_all(functions_array: List[Callable], delay_ms: int) -> List[Callable]:
   Wraps each function in a given list with a delay.
   The delayed functions, when invoked,
   will wait for the specified milliseconds before executing the original function.
    :param functions_array: The list of functions to be delayed.
    :param delay_ms: The number of milliseconds to delay the function execution.
    :return: A list of delayed functions.
```

## Time and Space Complexity **Time Complexity**

return wrapper

:return: The delayed function.

async def wrapper(\*args, \*\*kwargs):

async def delayed\_function(original\_function: Callable) -> Callable:

:param original\_function: The function to be delayed.

# Invoke the original function after the delay

return original\_function(\*args, \*\*kwargs)

# Map each original function to its delayed counterpart

return [delayed\_function(func) for func in functions\_array]

Returns an async function that delays the given original function.

# Delay the execution by sleeping for the specified amount of time

await asyncio.sleep(delay\_ms / 1000) # asyncio.sleep uses seconds

The time complexity of the delayAll function is O(n), where n is the number of functions in the functions array. The map function iterates over each function and wraps it with an asynchronous function that introduces a delay before invoking the original function. The delay itself (setTimeout(resolve, ms)) does not add to the computational complexity, as it's merely setting up a timer and not performing any computation during the waiting time.

However, invoking each delayed function will incur the ms delay plus the time complexity of the original function. If the time

complexities of the provided functions are not uniform, the overall time complexity when executing all the delayed functions would be 0(m + f(n)), where m is the constant delay (ms) multiplied by n and f(n) represents the time complexity of the original functions which is executed after the delay.

# **Space Complexity**

The space complexity of the delayAll function is also 0(n). This is due to the creation of a new array of asynchronous functions based on the length of the input functions array. Each created function is essentially a closure that captures the fn and ms variables.