2637. Promise Time Limit

Medium **Leetcode Link**

The task is to create a wrapper function that controls how long an asynchronous function is allowed to run before it's considered to

Problem Description

have taken too long. An asynchronous function, which we'll call fn, is given to us along with a timeout threshold t in milliseconds. We need to return a new function that behaves in the following way: When called, it lets fn run with any arguments passed to it.

- If fn finishes its task within t milliseconds, the new function should complete with the same result as fn.
- If fn does not finish within the allotted t milliseconds, the new function should stop waiting for fn and instead return a rejection with the message "Time Limit Exceeded".
- This problem combines asynchronous programming with timing control, requiring knowledge of promises and the race condition in asynchronous flows.

Intuition

The intuition behind the solution is to use concurrency in JavaScript Promise operations. We can race two promises against each

other: one promise represents the completion of the input function fn, and the other represents the time limit as a timeout. Here's

the thinking process: 1. Start by invoking the fn with its arguments, wrapped in a promise. 2. Create a timeout promise that will reject with "Time Limit Exceeded" after t milliseconds. 3. Use Promise race to run both promises (the function promise and the timeout promise) against each other.

- 5. If fn takes too long and the timeout elapses first, the race is won by the timeout promise, and "Time Limit Exceeded" is returned.

4. If fn completes first, the race is won by the function promise, and its result is returned.

- This approach ensures that regardless of what fn is doing, we're not waiting for it longer than t milliseconds, enforcing a strict time limit on its execution.
- Solution Approach The solution makes use of Promises and the race method provided by the Promise API. The idea is to have two promises: one that

represents the asynchronous operation fn and another that acts as a timer. Whichever promise settles first determines the outcome. To implement this:

2. timeLimit returns a new function that accepts any number of arguments (...args).

3. Inside this new function, we set up a race condition between two promises: • The first promise is the result of calling fn(...args). Since fn is asynchronous and returns a promise, it will either resolve with the result of fn or reject if fn fails.

• The second promise is created with a call to new Promise((_, reject) => setTimeout(() => reject('Time Limit

Exceeded'), t)). This promise waits for t milliseconds and then rejects with "Time Limit Exceeded".

4. We use Promise race to run these two promises. This method returns a new promise that resolves or rejects as soon as one of

1. We define a function timeLimit that takes an asynchronous function fn and a timeout value t.

the promises in the array resolves or rejects, with the value or reason from that promise.

system remains responsive and does not wait indefinitely for an action to complete.

1 // Simulate an asynchronous operation that takes a random amount of time to complete,

new Promise((_, reject) => setTimeout(() => reject('Time Limit Exceeded'), t))

a timeout behavior for any asynchronous operation.

Promise race to reject with "Time Limit Exceeded".

- 5. When calling the race, if fn completes before the timeout, the resulting promise will resolve with the value provided by fn. 6. If fn does not complete within t milliseconds, the timer promise will reject first, causing the race to end with a rejection. By using Promise race, we ensure that our wrapped function never waits longer than t milliseconds to settle. This effectively creates
- Here's a detailed handling scenario: • Assume the function call fn(...args) finishes in less time than t. In this case, Promise race will resolve to whatever fn(...args)

• On the other hand, if fn(...args) takes longer than t milliseconds, the promise from setTimeout will reject first, causing

This pattern is a common solution in scenarios where a timeout needs to be enforced on asynchronous operations, ensuring that a

Let's say we have an asynchronous function asyncoperation that resolves after a random amount of time, which sometimes might exceed our threshold. We want to ensure that if asyncOperation takes more than 2000 milliseconds (2 seconds), it should be considered as failed due to taking too long. We will use the timeLimit function to enforce this rule.

function asyncOperation(resolveValue) { return new Promise((resolve) => { const delay = Math.floor(Math.random() * 3000); // Random delay from 0 to 2999 ms

Promise.race([

fn(...args),

});

8

13

2 // sometimes more than 2000 milliseconds.

11 const timeLimit = (fn, t) => (...args) =>

setTimeout(() => resolve(resolveValue), delay);

// The `timeLimit` function wraps around our `asyncOperation`.

.then(result => console.log(`Operation successful: \${result}`))

Example Walkthrough

resolves to.

15 16 // Let's create a time-limited version of our `asyncOperation` with a 2000 ms limit. const limitedAsyncOperation = timeLimit(asyncOperation, 2000); 19 // Call the time-limited asynchronous function with a sample resolve value. limitedAsyncOperation('Sample resolve value')

```
.catch(error => console.log(`Operation failed: ${error}`));
In this walkthrough:
 1. We define an asynchronous function asyncoperation that would typically represent a more complex async process, such as an
   API call or a database transaction.
 2. We set up a timeLimit function according to the solution approach.
 3. We create limitedAsyncOperation by passing asyncOperation and the desired timeout of 2000 milliseconds to timeLimit.
 4. When we call limitedAsyncOperation, it initiates two parallel promises:
      • The original asyncOperation promise that resolves after a random delay.

    A new promise that will reject with "Time Limit Exceeded" after 2000 milliseconds.

 5. Promise race is used to return the outcome of whichever promise settles first.

    If asyncOperation completes in less than 2 seconds, its resolve value will be logged to the console.

      • If asyncOperation takes more than 2 seconds, the console will log "Operation failed: Time Limit Exceeded".
The above code demonstrates how the timeLimit function effectively imposes a timeout constraint on an asynchronous operation.
```

Python Solution 1 from typing import Callable, Any

9

10

11

16

17

24

30

- 2 import asyncio # Define a generic asynchronous function type that returns a Future. AsyncFunction = Callable[..., Any]
- :param async_function: The asynchronous function to wrap. :param time_limit_millis: The maximum amount of time (in milliseconds) to wait before cancelling. 13 :returns: A function that behaves like the original async function but with a time limit. 14 15

return await asyncio.wait_for(async_function(*args, **kwargs), time_limit_millis / 1000.0)

async def time_limit(async_function: AsyncFunction, time_limit_millis: int) -> AsyncFunction:

Wraps an asynchronous function with a time limit, enforcing it to either

Use asyncio.wait_for to apply a timeout to the async function

27 # Define an asynchronous operation that may take longer than the allocated time limit.

resolve or reject within a set timeframe.

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

async def example_async_function(duration):

31 # Wrap the async function with a time limit.

limited_function = time_limit(example_async_function, 100)

return asyncFunction.apply(args).get();

throw new RuntimeException(e);

Future<T> future = executor.submit(task);

future.cancel(true);

1 #include <iostream> // For std::cout and std::endl

#include <thread> // For std::this_thread::sleep_for

* @param asyncFunction The asynchronous function to wrap.

#include <functional> // For std::function

#include <chrono> // For std::chrono

4 #include <stdexcept> // For std::runtime_error

* resolve or fail within a set timeframe.

auto result = asyncFunction(args);

2 #include <future> // For std::async, std::future, and std::chrono

8 // Define a type alias for a generic function that returns a std::future.

9 using AsyncFunction = std::function<std::future<void>(std::vector<int>)>;

// Start the async function with the given arguments

// Wait for the result for the given time limit

// If it times out, throw a runtime_error

// Otherwise, return the original function's result

39 // longer than the specified time limit and handles timeout errors.

limitedAsyncOperation({150}).get(); // Using 150ms for the operation

* Wraps an asynchronous function with a time limit, enforcing it to either

// Use Promise.race to compete the async function call against a timeout

// Create a new Promise that automatically rejects after timeLimitMillis

// limited(150).catch(error => console.log(error.message)); // Expected output: "Time Limit Exceeded" at t=100ms

std::cout << e.what() << std::endl; // Expected output: "Time Limit Exceeded" after t=100ms</pre>

throw std::runtime_error("Time Limit Exceeded");

* Wraps an asynchronous function with a time limit, enforcing it to either

* @param timeLimitMillis The maximum amount of time (in milliseconds) to wait before canceling.

* @return A function that behaves like the original async function but with a time limit.

return [asyncFunction, timeLimitMillis](std::vector<int> args) -> std::future<void> {

if (result.wait_for(std::chrono::milliseconds(timeLimitMillis)) == std::future_status::timeout) {

// Return a new function which will race the original function against a timeout

auto timeLimit(AsyncFunction asyncFunction, int timeLimitMillis) -> AsyncFunction {

Executors.newSingleThreadScheduledExecutor()

} catch (ExecutionException | InterruptedException e) {

// Schedule a task to cancel the future after the time limit

// Cancel the future if it times out or encounters an issue

throw new RuntimeException("Time Limit Exceeded", e);

await asyncio.sleep(duration)

20 except asyncio.TimeoutError: 21 # Raise a custom exception if the function times out 22 raise TimeoutError('Time Limit Exceeded')

```
33
  # Use the wrapped function with a time-out that exceeds the time limit and handle exceptions.
35 async def main():
       try
36
```

26 # Usage example:

try:

return wrapper

```
37
           # This should raise a TimeoutError after 100ms
           await limited_function(0.150)
       except TimeoutError as e:
39
           print(e)
40
42 # Run the main function to demonstrate usage.
   if __name__ == '__main__':
       asyncio.run(main())
45
Java Solution
   import java.util.concurrent.*;
   import java.util.function.*;
    /**
    * Represents a generic function that returns a Future.
    */
   @FunctionalInterface
   interface AsyncFunction<T> {
       Future<T> apply(Object... params);
10 }
11
   * Wraps an asynchronous function with a time limit, enforcing it to either
    * complete or cancel within a set timeframe.
15
    * @param asyncFunction The asynchronous function to wrap.
   * @param timeLimitMillis The maximum amount of time (in milliseconds) to wait before cancelling.
   * @param <T> The type of the result provided by the asynchronous function.
    * @return A function that behaves like the original async function but with a time limit.
20
    */
   public static <T> AsyncFunction<T> timeLimit(AsyncFunction<T> asyncFunction, long timeLimitMillis) {
22
       // Return a new function that upon invocation, submits the original task to an executor
23
       // and applies the timeout.
       return (Object... args) -> {
24
25
           // Create a new executor to run the asynchronous function
           ExecutorService executor = Executors.newSingleThreadExecutor();
26
27
28
           // Submit the original asynchronous function as a callable task to the executor
29
           Callable<T> task = () -> {
30
               try {
```

42 try { 43 // Return the result of the future, awaiting termination with the given time limit 44 return CompletableFuture.completedFuture(future.get(timeLimitMillis, TimeUnit.MILLISECONDS)); 45 } catch (TimeoutException | InterruptedException | ExecutionException e) { 46

} finally {

};

31

33

34

35

36

37

38

39

40

41

47

48

49

50

52

53

56

62

65

66

68

79

10

14

15

18

20

21

22

23

24

25

26

27

28

29

30

31

32

33

52

54

55

57

58

60

59 }

/**

15

16

17

19

20

21

try {

*/

11 /**

C++ Solution

51 // Shutdown the executor service to prevent lingering threads executor.shutdown(); 55 } // Usage example (uncomment to use within a main method or other appropriate context): AsyncFunction<Void> limited = timeLimit(duration -> { CompletableFuture<Void> future = new CompletableFuture<>(); new Thread(() -> { try { Thread.sleep((Long) duration); future.complete(null); // Resolve the future upon successful completion } catch (InterruptedException e) { future.completeExceptionally(e); }).start(); return future; 70 }, 100); // Call the wrapped function with a duration that exceeds the limit, catching any exceptions try { limited.apply(150).get(); } catch (Exception e) { System.out.println(e.getMessage()); // Expected output: "Time Limit Exceeded" 77 } 78 */

.schedule(() -> future.cancel(true), timeLimitMillis, TimeUnit.MILLISECONDS);

34 } 35 36 // Usage example: 37 // The following lines of code provide a basic example of how the timeLimit function 38 // could be used in practice. It defines an asynchronous operation that could take

};

return result;

53 // Run and catch any timeout errors

} catch (std::runtime_error &e) {

40 41 // Define the actual async operation function 42 AsyncFunction myAsyncOperation = [](std::vector<int> duration) -> std::future<void> { return std::async(std::launch::async, [duration]() { 43 // Simulate a long running operation 44 45 std::this_thread::sleep_for(std::chrono::milliseconds(duration[0])); std::cout << "Operation finished" << std::endl;</pre> 46 47 }); }; 48 49 50 // Use the timeLimit wrapper with the async operation with a timeout of 100ms 51 AsyncFunction limitedAsyncOperation = timeLimit(myAsyncOperation, 100);

// Print out the error message if there's a timeout

// Define a generic function type that returns a Promise.

type AsyncFunction = (...params: any[]) => Promise<any>;

* @param asyncFunction The asynchronous function to wrap.

* resolve or reject within a set timeframe.

asyncFunction(...args),

return Promise.race([

Time and Space Complexity

* @param timeLimitMillis The maximum amount of time (in milliseconds) to wait before rejecting. * @returns A function that behaves like the original async function but with a time limit. */ function timeLimit(asyncFunction: AsyncFunction, timeLimitMillis: number): AsyncFunction { // Return a new function which will race the original function against a timeout 13 return async function (...args) { 14

Typescript Solution

}; 22 } 23 24 // Usage example: 25 // Define an asynchronous operation that may take longer than the allocated time limit. 26 // const limited = timeLimit((duration) => new Promise(resolve => setTimeout(resolve, duration)), 100); // Use the wrapped operation with a timeout that exceeds the time limit and catch any errors.

1);

the promise if it doesn't resolve within time t. The computational complexities are as follow: Time Complexity

any loops or recursive calls.

Since fn could be any function, its complexity can vary. When this resulting function is called, it will execute fn(...args) and setTimeout(..., t) concurrently, and the Promise.race() will settle as soon as the first promise settles.

Thus, the overall time complexity of the resulting function is O(f(n)) where O(f(n)) represents the time complexity of the function for

However, the time complexity of the resulting function when called is determined by fn, which is an input parameter to timeLimit.

The time complexity of the timeLimit function itself is 0(1) (constant time), as it simply sets up a Promise.race() construct without

The code defines a function timeLimit that takes another function fn and a time limit t, and returns a new function that will reject

new Promise((_, reject) => setTimeout(() => reject(new Error('Time Limit Exceeded')), timeLimitMillis)),

Space Complexity The space complexity of the timeLimit function is 0(1). It does not utilize any additional space that grows with the input size, so it

uses constant space.

The space complexity of the resulting function when it is called with a specific fn is determined by the space that fn uses. If fn uses space that grows with the input, then the resulting function will also have a space complexity that reflects that growth. However, since we do not have specifics on what fn does, we denote the space complexity of the function as O(g(n)), where O(g(n))

that is passed to timeLimit.

represents the space complexity of fn.

In addition to the space used by fn, the resulting function uses space for the Promise race() and the setTimeout. However, this additional space does not grow with the input and is thus considered constant, not affecting the overall space complexity which remains O(g(n)).