Problem 2650: Design Cancellable Function

Problem Information

Difficulty: Hard
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

Sometimes you have a long running task, and you may wish to cancel it before it completes. To help with this goal, write a function

cancellable

that accepts a generator object and returns an array of two values: a

cancel function

and a

promise

.

You may assume the generator function will only yield promises. It is your function's responsibility to pass the values resolved by the promise back to the generator. If the promise rejects, your function should throw that error back to the generator.

If the cancel callback is called before the generator is done, your function should throw an error back to the generator. That error should be the string

"Cancelled"

(Not an

Error

object). If the error was caught, the returned promise should resolve with the next value that was yielded or returned. Otherwise, the promise should reject with the thrown error. No more code should be executed.

When the generator is done, the promise your function returned should resolve the value the generator returned. If, however, the generator throws an error, the returned promise should reject with the error.

An example of how your code would be used:

function* tasks() { const val = yield new Promise(resolve => resolve(2 + 2)); yield new Promise(resolve => setTimeout(resolve, 100)); return val + 1; // calculation shouldn't be done. } const [cancel, promise] = cancellable(tasks()); setTimeout(cancel, 50); promise.catch(console.log); // logs "Cancelled" at t=50ms

If instead

cancel()

was not called or was called after

t=100ms

, the promise would have resolved

5

.

Example 1:

Input:

generatorFunction = function*() {   return 42; } cancelledAt = 100

Output:

{"resolved": 42}

Explanation:

const generator = generatorFunction(); const [cancel, promise] = cancellable(generator); setTimeout(cancel, 100); promise.then(console.log); // resolves 42 at t=0ms

The generator immediately yields 42 and finishes. Because of that, the returned promise immediately resolves 42. Note that cancelling a finished generator does nothing.

Example 2:

Input:

generatorFunction = function*() {   const msg = yield new Promise(res => res("Hello"));   throw `Error: ${msg}`; } cancelledAt = null

Output:

{"rejected": "Error: Hello"}

Explanation:

A promise is yielded. The function handles this by waiting for it to resolve and then passes the resolved value back to the generator. Then an error is thrown which has the effect of causing the promise to reject with the same thrown error.

Example 3:

Input:

generatorFunction = function*() {   yield new Promise(res => setTimeout(res, 200));   return "Success"; } cancelledAt = 100

Output:

{"rejected": "Cancelled"}

Explanation:

While the function is waiting for the yielded promise to resolve, cancel() is called. This causes an error message to be sent back to the generator. Since this error is uncaught, the returned promise rejected with this error.

Example 4:

Input:

generatorFunction = function*() {   let result = 0;   yield new Promise(res => setTimeout(res, 100));   result += yield new Promise(res => res(1));   yield new Promise(res => setTimeout(res, 100));   result += yield new Promise(res => res(1));   return result; }
cancelledAt = null

Output:

{"resolved": 2}

Explanation:

4 promises are yielded. Two of those promises have their values added to the result. After 200ms, the generator finishes with a value of 2, and that value is resolved by the returned promise.

Example 5:

Input:

generatorFunction = function*() {   let result = 0;   try {   yield new Promise(res => setTimeout(res, 100));   result += yield new Promise(res => res(1));   yield new Promise(res => setTimeout(res, 100));   result += yield new Promise(res => res(1));   } catch(e) {   return result;   }   return result; } cancelledAt = 150

Output:

{"resolved": 1}

Explanation:

The first two yielded promises resolve and cause the result to increment. However, at t=150ms, the generator is cancelled. The error sent to the generator is caught and the result is returned and finally resolved by the returned promise.

Example 6:

Input:

generatorFunction = function*() {   try {   yield new Promise((resolve, reject) => reject("Promise Rejected"));   } catch(e) {   let a = yield new Promise(resolve => resolve(2)); let b = yield new Promise(resolve => resolve(2));   return a + b;   }; } cancelledAt = null

Output:

{"resolved": 4}

Explanation:

The first yielded promise immediately rejects. This error is caught. Because the generator hasn't been cancelled, execution continues as usual. It ends up resolving 2 + 2 = 4.

Constraints:

cancelledAt == null or 0 <= cancelledAt <= 1000

generatorFunction

returns a generator object

Code Snippets

JavaScript:

```
/**
* @param {Generator} generator
* @return {[Function, Promise]}
*/
var cancellable = function(generator) {
};

/**
* function* tasks() {
* const val = yield new Promise(resolve => resolve(2 + 2));
* yield new Promise(resolve => setTimeout(resolve, 100));
* return val + 1;
* }
* const [cancel, promise] = cancellable(tasks());
* setTimeout(cancel, 50);
* promise.catch(console.log); // logs "Cancelled" at t=50ms
*/
```

TypeScript:

```
function cancellable<T>(generator: Generator<Promise<any>, T, unknown>): [()
=> void, Promise<T>] {

};

/**
* function* tasks() {
* const val = yield new Promise(resolve => resolve(2 + 2));
* yield new Promise(resolve => setTimeout(resolve, 100));
* return val + 1;
* }
* const [cancel, promise] = cancellable(tasks());
* setTimeout(cancel, 50);
* promise.catch(console.log); // logs "Cancelled" at t=50ms
*/
```

Solutions

JavaScript Solution:

```
/**
* Problem: Design Cancellable Function
* Difficulty: Hard
* Tags: array, string
*
```

```
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {Generator} generator
* @return {[Function, Promise]}
*/
var cancellable = function(generator) {

};

/**
* function* tasks() {
* const val = yield new Promise(resolve => resolve(2 + 2));
* yield new Promise(resolve => setTimeout(resolve, 100));
* return val + 1;
* }
* const [cancel, promise] = cancellable(tasks());
* setTimeout(cancel, 50);
* promise.catch(console.log); // logs "Cancelled" at t=50ms
*/
```

TypeScript Solution:

```
/**
* Problem: Design Cancellable Function
* Difficulty: Hard
* Tags: array, string
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function cancellable<T>(generator: Generator<Promise<any>, T, unknown>): [()
=> void, Promise<T>] {

};
```

```
/**
* function* tasks() {
* const val = yield new Promise(resolve => resolve(2 + 2));
* yield new Promise(resolve => setTimeout(resolve, 100));
* return val + 1;
* }
* const [cancel, promise] = cancellable(tasks());
* setTimeout(cancel, 50);
* promise.catch(console.log); // logs "Cancelled" at t=50ms
*/
```