2797. Partial Function with Placeholders

The challenge is to create a function called partialFn which modifies the behavior of an existing function fn. This modification

Problem Description

Easy

actually called. This concept is known as partial application or currying. The args array may contain placeholders, represented by the underscore character "_". When partialFn is later called with

involves pre-filling certain arguments of the fn function with values provided in an array called args, before the partialFn function is

Leetcode Link

additional arguments (restArgs), these placeholders should be replaced in order with the provided restArgs. Moreover, if there are more values left in restArgs after filling all placeholders, these values should be appended to the args array.

with any specific values from args, with placeholders substituted by restArgs, and ends with any additional restArgs values

in which it was defined so that it can access variables from that scope even after the scope execution has finished.

In simple terms, when partialFn is eventually called, it should call the original fn function with a new list of arguments. This list starts

appended. Intuition

The solution to this problem centers around the concept of closures in programming. A closure is a function that captures the scope

With that in mind, the partial function does two key things:

point in time.

1. It returns a new function, partialFn, that 'remembers' the original function fn and the pre-filled args array via a closure. 2. When partialFn is called, it replaces each placeholder ("_") in args with the corresponding element from restArgs. Then, if

The intuition behind replacing placeholders and appending additional restArgs is similar to merging two lists with the twist that some

elements in the first list are explicitly marked for replacement.

there are additional restArgs left, it appends them to the end of args.

- When all arguments are in place, the final step is to call the original function fn using the spread operator ..., which allows us to
- pass the arguments as separate values even though they are contained in an array. This implementation provides a flexible way to re-use functions by pre-filling some of their arguments and deciding others at a later

Solution Approach The reference solution provided is an implementation of the partial function application using JavaScript (or TypeScript in this case)

closures and the spread operator. Here's a step-by-step breakdown of the implementation:

1. Creating the Closure: The function partial takes a function fn and an array args as parameters. It returns a new function

2. Using the Spread Operator: Within the body of partialFn, we use the rest parameter syntax (...restArgs) to collect all

additional arguments supplied to partialFn into an array called restArgs.

3. Replacing Placeholders: We then iterate over args and replace any placeholders ("_") with the corresponding values from restArgs. We use a variable i to keep track of the current index of restArgs from which we need to take the next value.

args[j] = restArgs[i++]; 5 }

restArgs. If there are, they are appended to the end of args.

We have a single pass over the args array to replace placeholders.

treat the elements of an array as individual parameters to another function.

• We have a single pass over the remaining elements of restArgs (if any) to append them.

1 for (let j = 0; j < args.length; ++j) {</pre>

if (args[j] === '_') {

partialFn that creates a closure over fn and args.

1 while (i < restArgs.length) {</pre> args.push(restArgs[i++]);

4. Appending Additional Arguments: After all placeholders are replaced, we check if there are still any unused arguments left in

5. Invoking the Original Function: Finally, we use the spread operator again to call the original function fn with the modified args array as individual arguments. 1 return fn(...args);

This solution approach is efficient because:

replacement requirements.

This implementation leverages common JavaScript features such as closures to ensure that the partial function retains access to the original fn and args variables when partialFn is later invoked. The spread operator simplifies argument handling, allowing us to

Let's illustrate the solution approach with a simple example. Suppose we have a function multiply that takes two numbers and

• We ensure that we call the original function with precisely the right set of arguments, respecting the order and placeholders'

Example Walkthrough

returns their product:

function multiply(a, b) {

Here is how we use partialFn:

1 const result = double(3);

return a * b;

Now, we want to use the partialFn to create a new function where the first argument of multiply is pre-filled with 2. We also want to use _ as a placeholder for the second argument that we will provide later.

1 const double = partialFn(multiply, [2, '_']);

Now let's call double with an additional argument, 3:

4. There are no additional elements in restArgs to append to args.

:param fn: The original function to be partially applied.

if add_args_index < len(additional_args):</pre>

Here's what happens step-by-step: 1. We call double which is our partialFn with 3 as restArgs.

5. We then call the original multiply function with the elements of args spread into separate arguments, effectively calling

The result is 6, which is the product of 2 and 3. Our partialFn successfully created a new function that doubled its input. This

:param preset_args: List of arguments to apply, where None represents placeholders for additional arguments.

demonstrates how the partial function application uses closures and the spread operator to pre-fill and replace function arguments

2. Inside partialFn, we iterate over args ([2, '_']), and when we find a placeholder ('_'), we replace it with the first element from

multiply(2, 3). 6. multiply returns 6, which is then returned by double.

Python Solution

on-demand.

8

9

10

11

12

13

14

15

16

17

18

26

27

28

29

35

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

};

restArgs (which is 3).

3. Now args becomes [2, 3].

def partial(fn, preset_args):

def partially_applied(*additional_args):

for index in range(len(final_args)):

if final_args[index] is None:

add_args_index += 1

final_args = list(preset_args)

add_args_index = 0

return fn(*final_args)

add_partially = partial(add, [1, None, 3])

return (additionalArgs) -> {

int addArgsIndex = 0;

} else {

print(add_partially(2)) # Outputs 6 (1 + 2 + 3)

return partially_applied

1 import java.util.function.Function;

return a + b + c

Creates a partially applied function with the possibility to have placeholders.

:return: A new function with some arguments preset and remaining arguments to be provided later.

Iterate over the preset arguments and replace placeholders with additional arguments.

Create a copy of the preset arguments to avoid modifying the original list.

final_args[index] = additional_args[add_args_index]

* Creates a partially applied function with the possibility to have placeholders.

Object[] finalArgs = new Object[presetArgs.length];

if (addArgsIndex < additionalArgs.length) {</pre>

finalArgs[i] = additionalArgs[addArgsIndex++];

System.arraycopy(finalArgs, 0, combinedArgs, 0, finalArgs.length);

// Call the original function with the combined set of arguments.

// Append any remaining additional arguments that were not used as placeholders.

Object[] combinedArgs = new Object[finalArgs.length + additionalArgs.length - addArgsIndex];

System.arraycopy(additionalArgs, addArgsIndex, combinedArgs, finalArgs.length, additionalArgs.length - addArgsIndex);

for (int i = 0; i < presetArgs.length; i++) {</pre>

finalArgs[i] = presetArgs[i];

if (addArgsIndex < additionalArgs.length) {</pre>

return fn.apply(combinedArgs);

if (presetArgs[i] == null) {

The original function to be partially applied.

* @return A new function with some arguments preset and remaining arguments to be provided later.

public static Function<Object[], Object> partial(Function<Object[], Object> fn, Object[] presetArgs) {

// Iterate over the preset arguments and replace placeholders with additional arguments.

* @param presetArgs Array of arguments to apply, where null represents placeholders for additional arguments.

In the above code, double is a new function that, when called with a number, will multiply that number by 2.

19 20 # Append any remaining additional arguments that were not used as placeholders. 21 while add_args_index < len(additional_args):</pre> 22 final_args.append(additional_args[add_args_index]) 23 add_args_index += 1 24 # Call the original function with the combined set of arguments.

```
/**
* This class provides a way to create a partially applied function with placeholders.
public class PartialFunction {
```

* @param fn

/**

*/

Java Solution

30 # Example usage:

31 # def add(a, b, c):

```
// Example usage:
43
       public static void main(String[] args) {
44
           Function<Object[], Object> add = (Object[] params) -> {
45
46
               int sum = 0;
47
               for (Object param : params) {
                   sum += (int) param;
48
49
50
               return sum;
51
           };
52
53
           Function<Object[], Object> addPartially = partial(add, new Object[]{1, null, 3});
           System.out.println(addPartially.apply(new Object[]\{2\})); // Outputs 6 (1 + 2 + 3)
54
55
56 }
57
C++ Solution
  1 #include <functional>
  2 #include <vector>
     #include <iostream>
     /**
      * Represents a placeholder for future arguments.
      * This could be replaced with std::optional or a dedicated placeholder type for more complex scenarios.
      */
     const auto _ = nullptr;
 10
 11 /**
     * Creates a partially applied function with the possibility to have placeholders.
      * @param fn The original function to be partially applied.
      * @param presetArgs Vector of arguments to apply, where nullptr represents placeholders for additional arguments.
 14
      * @returns A new function with some arguments preset and remaining arguments to be provided later.
 16
     */
    template<typename ReturnType, typename... Args>
 18 auto partial(std::function<ReturnType(Args...)> fn, std::vector<std::variant<Args..., std::nullptr_t>> presetArgs)
 19
         // The returned lambda captures fn and presetArgs by value.
 20
         return [fn, presetArgs](Args... additionalArgs) mutable -> ReturnType {
 21
 22
             std::vector<std::variant<Args..., std::nullptr_t>> finalArgs = presetArgs;
 23
             std::queue<std::variant<Args...>> addArgsQueue({additionalArgs...});
 24
 25
             // Iterates over the preset arguments and replaces placeholders with additional arguments.
 26
             for (auto& arg : finalArgs) {
                 if (std::holds_alternative<std::nullptr_t>(arg)) {
 27
 28
                     if (!addArgsQueue.empty()) {
 29
                         arg = addArgsQueue.front();
 30
                         addArgsQueue.pop();
 31
 32
 33
 34
 35
             // Appends any remaining additional arguments that were not used as placeholders.
 36
             while (!addArgsQueue.empty()) {
 37
                 finalArgs.push_back(addArgsQueue.front());
 38
                 addArgsQueue.pop();
 39
 40
 41
             // Calls the original function with the combined set of arguments using std::apply.
 42
             return std::apply(fn, finalArgs);
 43
         };
 44
 45
     // Example usage
     int add(int a, int b, int c) {
 48
         return a + b + c;
 49
 50
    int main() {
```

25 26 27 // Call the original function with the combined set of arguments. return fn(...finalArgs); 28 29 **}**; 30

33 // const add = (a, b, c) => a + b + c;

Time and Space Complexity

an analysis of its computational complexity:

// Example usage:

52

53

54

55

56

57

58

59

60

62

61 }

/**

*/

6

10

12

13

14

15

16

17

18

19

20

21

22

23

24

31

std::function<int(int, int, int)> addFn = add;

auto addPartially = partial(addFn, {1, _, 3});

* @param fn The original function to be partially applied.

return function(...additionalArgs): any {

const finalArgs = [...presetArgs];

if (finalArgs[index] === '_') {

while (addArgsIndex < additionalArgs.length) {</pre>

// const addPartially = partial(add, [1, '_', 3]);
// console.log(addPartially(2)); // Outputs 6 (1 + 2 + 3)

let addArgsIndex = 0;

std::cout << addPartially(2) << std::endl;</pre>

// Outputs 6 (1 + 2 + 3)

return 0;

Typescript Solution

// Create a partially applied function with a placeholder for the second argument

* Creates a partially applied function with the possibility to have placeholders.

for (let index = 0; index < finalArgs.length; ++index) {</pre>

finalArgs.push(additionalArgs[addArgsIndex++]);

if(addArgsIndex < additionalArgs.length) {</pre>

* @param presetArgs Array of arguments to apply, where '_' represents placeholders for additional arguments.

* @returns A new function with some arguments preset and remaining arguments to be provided later.

// Iterate over the preset arguments and replace placeholders with additional arguments.

function partial(fn: (...args: any[]) => any, presetArgs: any[]): (...args: any[]) => any {

finalArgs[index] = additionalArgs[addArgsIndex++];

// Append any remaining additional arguments that were not used as placeholders.

// Create a copy of the preset arguments to avoid modifying the original array.

Time Complexity The time complexity of the partial function itself, during its creation, is O(n) where n is the length of the args array. This is due to the initial for-loop which iterates over the entire args array to replace the placeholders with actual values when the returned function is eventually called.

When the returned function is called (...restArgs), it iterates over the args array up to a maximum of n times again (to replace

placeholders), and then it iterates over restArgs which can be denoted as m, pushing any remaining arguments into the args array.

Thus, the time complexity for the returned function invocation is O(n + m), where m is the number of arguments passed when the

The given TypeScript code defines a function named partial, which takes a function fn and an array args containing any number of

arguments, where some arguments can be placeholders (denoted by '_'). It returns a new function that, when called, will invoke fn

with a combination of the args provided during partial application and the arguments passed to the newly created function. Here is

returned function is called. Combining these, invoking the returned function has a time complexity of O(n + m).

Space Complexity

The space complexity of the partial function is also 0(n). Additional space is required for storing the partially applied arguments (args). When the returned function is called and args is updated, no additional space proportional to args length is required because the changes happen in place.

However, it should be noted that this analysis assumes the fn function that is eventually called does not allocate additional space, as the space complexity of fn might have implications on the overall space complexity of the returned function. But with the given code, ignoring fn's own complexity, the space complexity remains O(n).