

## **Problem Description**

In this problem, you are given a function fn. The task is to create a curried version of this function. The concept of currying comes from functional programming and involves transforming a function that takes multiple arguments into a sequence of functions, each taking a single argument.

For instance, if you have a function sum that adds three numbers like sum(1,2,3), a curried version csum should allow you to call csum(1)(2)(3), csum(1)(2,3), csum(1,2)(3), or csum(1,2,3). Each of these calls should produce the same result as the original sum.

The point of currying is that you can hold ('freeze') some parameters while passing the remaining ones later. This process is useful for creating more specific functions from general-purpose ones and can be a powerful technique in many programming scenarios.

## Intuition

The intuition behind creating a curried function starts with understanding that we need to return a function that can take any number of arguments and hold them until we have enough arguments to call the original function fn. Our curried function should return another function if it doesn't have all the necessary arguments, or call the original function with all the available arguments if it does.

1. Define a function curry that accepts a function fn, which is the one we want to transform.

To implement this, we:

fn needs (fn.length).

- 2. The curry function returns a new function named curried, utilizing closures to keep track of the arguments given so far.
- 3. The curried function checks if the number of arguments it has received (args.length) is at least the number of arguments that
- 4. If we have enough arguments, we call fn directly with those arguments. 5. If we do not yet have all the arguments, curried returns a new function that accepts the next set of arguments (nextArgs). This
- new function, when called, will concatenate the new arguments to the existing ones and call curried again.
- 6. This process continues recursively until curried receives enough arguments to call the original function fn.
- This approach allows us to partially apply arguments to the function and call it multiple times with different arguments until all the necessary arguments are provided.

Solution Approach The implementation of the curry function uses the concepts of closures and recursion in JavaScript, allowing us to progressively

### accumulate arguments until we are ready to apply them all to the original function. The critical aspects of the solution are as follows:

1. Closure: A closure is a function that remembers the environment in which it was created. In the curry implementation, the curried function forms a closure over args, which allows it to remember the arguments passed to it across multiple calls.

- 2. Recursion: The curried function is recursive, as it can return itself with partially applied arguments until it has enough arguments to apply to fn.
- Here's a step-by-step breakdown of the algorithm used in the solution: • The curry function takes a function fn as its argument and returns another function curried.

• The curried function can receive multiple arguments each time it is called (...args in the parameter list is a rest parameter that

collects all arguments into an array). It checks if the received arguments are sufficient to call the original function by comparing

concatenated together (...args, ...nextArgs). This is where the recursion happens.

- args.length (the length of the accumulated arguments) with fn.length (the number of parameters expected by fn). • If enough arguments are provided (args.length >= fn.length), curried immediately calls fn with those arguments using the spread syntax ...args, which expands the array contents as separate arguments to the function call.
- This process continues until the call to curried has enough arguments to call the original function fn, at which point the final value is returned.

If there are not enough arguments yet, curried returns a new function. This new function takes additional arguments

(...nextArgs) and again calls curried, this time with both the previously accumulated arguments and the new ones

Example Walkthrough

To illustrate the solution approach for creating a curried function, let's use a straightforward sum function that adds two numbers:

By using these techniques, the implementation supports creating a curried function that is flexible in its invocation style.

1 function curry(fn) {

**}**;

**}**;

12

13

14

**15** }

// Returns the curried function.

return function curried(...args) {

1 function sum(a, b) {

return a + b;

mentioned in the solution approach.

Now, let's go through the process of using the provided solution approach to create a curried version of this function:

```
// If the arguments are sufficient, call the original function.
if (args.length >= fn.length) {
  return fn.apply(this, args);
} else {
  // If not, return a new function waiting for the rest of the arguments.
  return function (...nextArgs)
    // Combine the already given args with the new ones and retry.
    return curried.apply(this, args.concat(nextArgs));
```

1. First, we define the curry function that converts any given function into a curried function. Assume that it does all the things

1 console.log(curriedSum(1)(2)); // Outputs: 3 - as it's a curried function. 3 console.log(curriedSum(1, 2)); // Also outputs: 3 - it works even if we pass all arguments in one go.

those to sum immediately and returns the result.

1 # Function that curries another function

return fn(\*args)

return curried\_more

if len(args) >= fn.\_\_code\_\_.co\_argcount:

def curried\_more(\*more\_args):

return curried(\*(args + more\_args))

BiFunction<Integer, Integer, Integer> sum = (a, b) -> a + b;

CurriedFunction<Integer, Integer> curriedSum = curry(sum);

Function<Integer, Integer> addToOne = curriedSum.apply(1);

Integer result = addToOne.apply(2); // Returns 3

System.out.println(result); // Prints: 3

// Call the curried 'sum' function with one argument at a time

// Curry the 'sum' function

// Output the result

2. We transform the sum function into its curried equivalent:

In the first instance, curriedSum(1) doesn't have enough arguments to call sum, so it returns a new function that takes the second

1 let curriedSum = curry(sum);

In the second instance, even though curriedSum is a curried function, we provide all required arguments at once, so it just applies

# Otherwise, return a function that awaits the rest of the arguments

3. Let's use our new curriedSum in different ways, which shows the flexibility of currying:

and recursion to continue gathering arguments until the original function can be called with all of them. Python Solution

This example demonstrates the concept of creating a curried function using closures to maintain state across multiple function calls

argument. Then, when we call this new function with (2), we fulfill the number of arguments, and sum is called with both values.

def curry(fn): # The curried function def curried(\*args): # If the number of provided arguments is sufficient, call the original function

```
13
       return curried
15 # Example of usage:
16 # A simple function that adds two numbers
```

return a + b

20 # Curry the 'sum' function

17 **def** sum(a, b):

10

12

else:

```
21 curried_sum = curry(sum)
22
23 # Call the curried 'sum' function with one argument at a time
  result = curried_sum(1)(2) # Returns 3
25
26 # Now 'result' variable holds the result of calling 'curried_sum'
  print(result) # Output will be 3
28
Java Solution
1 import java.util.function.BiFunction;
   import java.util.function.Function;
   // Interface for a curried function that can accept one argument
   interface CurriedFunction<T, U> {
       Function<U, T> apply(T t);
   public class CurryExample {
       // Method that curries another BiFunction
       public static <T, U, R> CurriedFunction<R, U> curry(BiFunction<T, U, R> biFunction) {
           // The curried function
12
           return (T t) -> (U u) -> biFunction.apply(t, u);
14
15
       // Example of usage:
16
       public static void main(String[] args) {
17
           // A simple function that adds two numbers
18
```

### 30 31 } 32

19

20

21

22

25

26

27

28

29

```
C++ Solution
1 #include <functional>
2 #include <iostream>
   // Curries another function
  template <typename Function, typename... Args>
  auto curry(Function&& fn, Args&&... args) {
       // Check if the number of arguments is sufficient to call the function
       if constexpr (sizeof...(args) >= fn.length) {
           // If enough arguments are provided, call the original function
           return std::bind(std::forward<Function>(fn), std::forward<Args>(args)...);
       } else {
11
12
           // If not enough arguments, return a lambda that takes the rest of the arguments
13
           return [=](auto&&... rest) {
               // Capture current arguments and call `curry` with all of them
14
               return curry(fn, args..., std::forward<decltype(rest)>(rest)...);
15
           };
16
17
18 }
19
  // Example of usage:
   // A simple function that adds two numbers
   int sum(int a, int b) {
23
       return a + b;
24 }
26 // Deduction guide for the curry function, to help the compiler deduce the return type
27 template <typename Function, typename... Args>
  auto curry(Function&& fn, Args&&... args) -> decltype(auto);
29
   int main() {
       // Curry the 'sum' function
       auto curriedSum = curry(sum);
33
34
       // Call the curried 'sum' function with one argument at a time
35
       auto addOne = curriedSum(1);
       int result = add0ne(2);
36
37
       std::cout << result; // Outputs 3</pre>
38
39
       return 0;
```

40 }

```
41
Typescript Solution
  // Function that curries another function
   function curry(fn: (...args: any[]) => any): (...args: any[]) => any {
       // The curried function
       return function curried(...args: any[]): any {
           // If the number of provided arguments is sufficient, call the original function
           if (args.length >= fn.length) {
               return fn(...args);
           // Otherwise, return a function that awaits the rest of the arguments
           return (...nextArgs: any[]) => curried(...args, ...nextArgs);
10
11
       };
12 }
13
  // Example of usage:
  // A simple function that adds two numbers
   function sum(a: number, b: number): number {
     return a + b;
  // Curry the 'sum' function
   // Call the curried 'sum' function with one argument at a time
   curriedSum(1)(2); // Returns 3
Time and Space Complexity
```

## The time complexity of the curry function itself is 0(1), as it's simply returning another function. However, the returned curried function can be called multiple times, depending on how many arguments it receives each time. When the final curried function is

executed with sufficient arguments, it calls the original function fn.

**Time Complexity** 

21 const curriedSum = curry(sum);

Assuming fn takes n arguments and has a time complexity of O(f(n)), where f(n) is the complexity of function fn, each partial application of curried is a constant time operation 0(1). If a curried function is invoked sequentially for each argument, the overall time complexity for all the calls until fn is invoked with all n arguments would be O(n). Therefore, provided that the function fn has a linear number of arguments, and ignoring the complexity of fn itself, the time complexity of making the series of calls until fn's execution completes would be O(n).

# **Space Complexity**

The space complexity is associated with the closures created for each partial application. Since each call to curried potentially returns a new closure that holds the given arguments, up to n closures could be created where n is the number of arguments fn requires.

Therefore, if there are n arguments, the space complexity would be O(n) due to the n closures that store the accumulated arguments until all are provided.