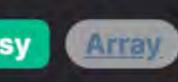
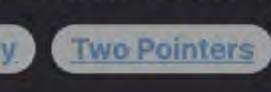
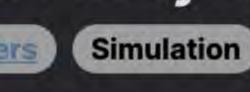
2562. Find the Array Concatenation Value







Problem Description





This LeetCode problem involves manipulating an array of integers to calculate what is referred to as a "concatenation value". Here's what you need to know:

Leetcode Link

- You have an array of integers called nums, with each element having a 0-based index. • The term "concatenation" of two numbers here refers to creating a new number by joining the digit sequences of both numbers
- end to end.
- For example, concatenating 15 and 49 yields 1549.
- Initially, the concatenation value is 0. To find the final concatenation value, you go through the following steps:
 - 1. If nums contains more than one number, take the first and last elements. 2. Concatenate those two elements and add their concatenation's numerical value to the concatenation value of nums.
 - 3. Remove the first and last elements from nums.
- 4. If only one number exists in nums, add its value to the concatenation value and remove it. This process repeats until nums is empty.
- Your goal is to return the final concatenation value after all elements are combined and removed according to the steps above.

Intuition

no more elements left to process.

Solution Approach

To approach the solution to this problem, you start by understanding that you'll be reducing the array from both ends until there are

Here's how you arrive at the solution step-by-step:

At each step where there are at least two elements, you consider the first and last element for the operation.

You convert each number to a string, concatenate those strings, then convert the resulting string back to an integer.

Recognize that array elements should be considered from both ends.

- Add this integer to the running total of the concatenation value. If there's only one element left in the process, simply add its value to the total.
- This process is repeated, updating two pointers (i and j) that represent the current first and last elements in the array. When the pointers meet or cross, you've processed all elements.
- The final concatenation value can be returned after considering all elements in the required manner.
- The implementation of the solution uses a straightforward simulation algorithm with two pointers. Let's go through the specifics:
 - Two pointers, i and j, are used to traverse the nums array from both ends toward the center.

We start by initializing an accumulator for the answer, ans, to start summing up the concatenation values.

- i is initialized to 0 as the start of the array. j is initialized to the last index of the array, which is len(nums) - 1.
 - Inside the loop, we concatenate the numerical strings of nums[i] and nums[j], by doing str(nums[i]) + str(nums[j]), and

The resultant integer is added to the ans accumulator.

while loop because it has no pair element.

then convert this string back to an integer using int().

If we exit the loop and find that i == j, it means there is one remaining element in the array which wasn't processed by the

We enter a while loop that continues as long as i < j indicating there are at least two elements in the current subrange.

- After concatenating and adding to ans, we advance i forward by one (i += 1) and move j backward by one (j -= 1) to move towards the center of the array.
 - simulation.

In that case, we simply add the value of this last remaining element (nums [i]) to ans.

operations. The core pattern here is the two-pointer technique that allows us to efficiently process elements from both ends of the array.

This problem does not require any complex data structures or algorithms, as it simply utilizes basic array manipulation and integer

• The solution approach is completed by returning the ans variable which contains the final concatenation value after the

Let's consider a small example to illustrate the solution approach with the nums array nums = [5,6,2,8]. 1. Initialize ans = 0 to keep track of the concatenation value.

Now we start the while loop:

4. Second iteration:

Example Walkthrough

3. First iteration:

- Since i < j, concatenate the first and last elements: nums [i] is 5 and nums [j] is 8. Their concatenation as strings is '5' + '8' which is '58'. Convert this back to an integer to get 58.
 - Add this to ans: ans = ans + 58, which makes ans = 58.

2. Initialize two pointers: i = 0 for the start of the array and j = len(nums) - 1 = 3 for the end of the array.

 The condition i < j still holds true. Concatenate nums [i] which is 6 and nums [j] which is 2 to get '62'.

Now, increment i so i = 1 and decrement j so j = 2.

 \circ Move i to i = 2 and j to j = 1.

unpaired element, we would add its value directly to ans.

def findTheArrayConcVal(self, nums: List[int]) -> int:

answer += int(str(nums[left]) + str(nums[right]))

// convert the result to integer, and add it to the result.

// Move the left index forward and the right index backward.

// Add the middle element directly to the result.

return result; // Return the computed "array conc val".

1 // This function finds a value based on the concatenation of array elements.

Thus, the time complexity is $O(n/2 \times log M)$, which simplifies to $O(n \times log M)$.

function findTheArrayConcVal(nums: number[]): number {

// Get the length of the array

// Initialize the front index

// Initialize the back index

const arrayLength = nums.length;

// Initialize the answer to zero

let backIndex = arrayLength - 1;

2 // Pairs the first and last elements moving inwards and adds their concatenation.

// If there is an unpaired middle element, it is added directly to the result.

result += Long.parseLong(nums[leftIndex] + "" + nums[rightIndex]);

// Check if there's a middle element left (in case of odd number of elements).

left, right = 0, len(nums) - 1

while left < right:

if left == right:

answer += nums[left]

Loop until the pointers meet or cross

- 6. Since all elements have been processed, the final concatenation value is the current value of ans, which is 120. We return ans. Therefore, the final concatenation value for the input array [5,6,2,8] using the solution approach provided would be 120.

As an integer, it is 62. Update ans = 58 + 62, making ans = 120.

Python Solution from typing import List

5. Now, i >= j, and the while loop condition is not met. However, we don't have an element that's unpaired. If there were an

Initialize the answer to 0 answer = 0 # Set pointers for the start and end of the array

Concatenate the numbers at the pointers, convert to int and add to the answer

```
15
               # Move the pointers towards the center
16
                left, right = left + 1, right - 1
17
18
           # If there is a middle element (odd number of elements), add it to the answer
19
```

class Solution:

9

11

12

13

14

20

21

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

```
22
23
           # Return the final answer
24
            return answer
25
Java Solution
   class Solution {
        * Calculates the "array conc val" by concatenating pairs of elements
4
        * from the beginning and end of the array moving towards the center.
        * If there's a middle element (odd number of elements), it adds it as is.
        * @param nums An array of integers.
        * @return The calculated "array conc val".
9
10
       public long findTheArrayConcVal(int[] nums) {
11
12
            long result = 0; // Initialize the result variable.
13
14
           int leftIndex = 0; // Start at the beginning of the array.
            int rightIndex = nums.length - 1; // Start at the end of the array.
15
16
17
           // Loop through the array from both ends until indices meet or cross.
           while (leftIndex < rightIndex) {</pre>
18
               // Concatenate the elements at current indices as strings,
19
```

35 36 } 37

leftIndex++;

rightIndex--;

if (leftIndex == rightIndex) {

result += nums[leftIndex];

```
C++ Solution
 1 class Solution {
 2 public:
       // Function to find the array concatenated value
       long long findTheArrayConcVal(vector<int>& nums) {
            long long concatenatedSum = 0; // Initialize sum of concatenated values
           int left = 0; // Starting index from the beginning of the array
           int right = nums.size() - 1; // Starting index from the end of the array
           // Loop through the array from both ends until the pointers meet or cross
           while (left < right) {</pre>
10
               // Concatenate the values at the current indices, convert to number, and add to sum
               concatenatedSum += stoll(to_string(nums[left]) + to_string(nums[right]));
12
13
               // Move the left pointer forward and the right pointer backward
14
               ++left;
15
               -- right;
16
18
19
           // If there is a middle element (odd number of elements), add it to the sum
20
           if (left == right) {
               concatenatedSum += nums[left];
21
22
24
           // Return the final concatenated sum
25
           return concatenatedSum;
26
27 };
28
Typescript Solution
```

13 14 // Loop until frontIndex is less than backIndex while (frontIndex < backIndex) {</pre> 15 16

let answer = 0;

let frontIndex = 0;

```
// Concatenate the elements by converting them to strings, adding them, and then converting back to a number
           answer += Number(`${nums[frontIndex]}${nums[backIndex]}`);
           // Increment the front index
           frontIndex++;
           // Decrement the back index
20
21
           backIndex--;
22
23
       // If there is a middle element, add it to the answer
24
       if (frontIndex === backIndex) {
25
           answer += nums[frontIndex];
26
27
28
29
       // Return the calculated answer
30
       return answer;
31 }
32
Time and Space Complexity
The provided Python code calculates a special value based on the input nums list. Let's analyze the time and space complexity of this
code.
Time Complexity
```

Space Complexity

The time complexity of the algorithm is determined by the while loop, which runs as long as i < j. Since the indices i and j move towards each other with each iteration, the loop executes approximately n/2 times, where n is the total number of elements in nums. Inside this loop, the algorithm concatenates the string representations of numbers at indices i and j, which takes 0(\log M) time, where M is the maximum value in the array (since the number of digits of a number x is proportional to $(\log x)$.

The space complexity of the algorithm is determined by the extra space needed to store the intermediate string representations created during the concatenation operation. The longest possible string is the concatenation of the two largest numbers in nums. Thus, the space complexity is proportional to the length of this string, which is 0(2 × \log M). This simplifies to 0(\log M) since constant factors are dropped in Big O notation.