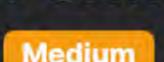
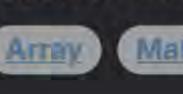
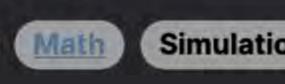
1806. Minimum Number of Operations to Reinitialize a Permutation







Problem Description



Medium Array Math Simulation Leetcode Link

You are given an even integer n. Initially, you have a permutation perm of size n where each element of perm follows the rule perm[i] == i (with 0-based indexing). You will perform a series of operations on this permutation, and after each operation, a new array arr is created with the following rules:

If i is odd, then the value of arr[i] becomes the value of perm[n / 2 + (i - 1) / 2].

we have the answer, which is the number of operations performed.

If i is even, then the value of arr[i] becomes the value of perm[i / 2].

After each operation, the arr becomes the new perm, and then you perform the same operation again on this new perm. The goal is

to determine the minimum number of operations required to make the perm array return to its initial state where each element is perm[i] == i.

To solve the problem efficiently, we need to observe the patterns that emerge when performing the operations on the permutation.

Intuition

before others. In particular, the element at index 1 has the potential to move through each position in the permutation since the operation exchanges elements in a certain cyclical pattern. The solution involves tracking the new position of the element at index 1 after each operation. The cycle's length is determined by the number of steps it takes for index 1 to return to its original position (as other indices will follow a similar cycle pattern due to the

The intuition comes from the fact that after each operation, some elements of the permutation will return to their original position

same repetition of operations). The cycle length indicates the minimum non-zero number of operations required to return the entire permutation back to its initial state. Hence, the approach is to simulate the process, tracking the index of the element at index 1 after each operation until it returns to its starting position. We increment a counter each time we perform an operation, and when the element at index 1 is back to position 1,

The steps for moving the element from index i during each operation, depending on whether i is even or odd, are directly applied in the code as conditional statements, with bitwise operations used to efficiently perform arithmetic calculations.

 i < n >> 1: In place of dividing n by 2 (n / 2), we use right shift operation which is faster. • i <= 1: Instead of multiplying i by 2 (i * 2), we use a left shift, which effectively doubles the value of i.

• i = (i - (n >> 1)) << 1 | 1: This is an optimized way of doing (2 * (i - n / 2)) + 1, again using bitwise shift and bitwise

The bitwise operations used are:

- OR to set the last bit to 1 for odd positioning.
- The solution capitalizes on these patterns and efficient bitwise operations to arrive at a quick and optimal solution.
- Solution Approach

The code provided uses a simple while loop to simulate the operations on the permutation and identifies the number of operations

The variable ans is used to count the number of operations performed.

needed to bring the array back to its initial state.

• The variable i represents the current index of the element that started at index 1. The element at index 1 is special because, as per the problem statement, every other element's final position depends on this element's path back to index 1.

1.

We increment ans by 1 for every iteration, representing an operation.

Inside the while loop:

The while loop continues indefinitely until the element at index 1 returns to its initial position, which triggers the break condition 1 ==

 We then use a conditional statement to determine the new position of the element initially at index 1 after the operation. If the current index i is less than half the size of the permutation (n >> 1), then the element goes to the position i * 2

(achieved by i <= 1), which is an even-indexed position in the permutation. If the current index i is greater or equal to half the size of the permutation, the element goes to the position (2 * (i - n /

defined pattern.

- 2)) + 1 (achieved by i = (i (n >> 1)) << 1 | 1), which is an odd-indexed position.
- After each operation, the conditional checks if the element at index 1 has returned to its initial position. If i == 1, the loop

terminates, and the variable ans is returned, which represents the minimum number of operations needed to reinitialize the

permutation. There are no additional data structures used, and the algorithm is not complex, as it relies on a simple simulation of the defined

operation on the permutation. The pattern identified is that of a cycle wherein the element at index 1 eventually returns to its position

after a set number of operations, which is the basis for calculating the answer. This simulation does not require modifying the array at each step and instead tracks the position of a single element, which keeps the space complexity to a constant. This method is optimal because it avoids unnecessary computation by directly tracking the one element that determines the

permutation's return to the initial state, taking advantage of the permuted array's specific structure of cycling through positions in a

Let's illustrate the solution approach with an example where n = 4. The initial permutation perm is [0, 1, 2, 3]. 1. After the first operation, according to the rules:

o arr[2] = perm[2 / 2] = perm[1] = 1

of the permutation.

index = 1

while True:

else:

while (true) {

} else {

if (index < n / 2) {</pre>

// break out of the loop

9

11

12

13

14

15

16

18

11

12

13

14

16

17

18

19

20

21

26

27

28

29

Typescript Solution

of the size of n.

1 // Function to determine the minimum number of operations required

numOperations++; // Increment the operation count

let numOperations: number = 0; // Initialize the count of operations to 0

// Check if the current index is in the first half of the array.

for (let index: number = 1; ;) { // Start with an index set to 1

// to reinitialize a permutation to its original configuration.

function reinitializePermutation(n: number): number {

29 }

Example Walkthrough

 \circ arr[3] = perm[4 / 2 + (3 - 1) / 2] = perm[2 + 1] = perm[3] = 3 The resulting arr after the first operation is [0, 2, 1, 3].

2. We now set perm to the new arr, and repeat the operation. The element at index 1 is now at index 2.

o arr[0] = perm[0 / 2] = perm[0] = 0 o arr[1] = perm[4 / 2 + (1 - 1) / 2] = perm[2 + 0] = perm[2] = 1

o arr[2] = perm[2 / 2] = perm[1] = 2

Following the provided solution approach step-by-step:

3. Applying the operation a second time:

o arr[0] = perm[0 / 2] = perm[0] = 0

```
\circ arr[3] = perm[4 / 2 + (3 - 1) / 2] = perm[2 + 1] = perm[3] = 3
    The resulting arr after the second operation is [0, 1, 2, 3], which is the initial state.
Therefore, the number of operations required for the permutation to return to its initial state is 2. This process can be generalised to
find the cycle length for any even integer n.
```

 \circ arr[1] = perm[4 / 2 + (1 - 1) / 2] = perm[2 + 0] = perm[2] = 2

• We start with ans = 0 (number of operations performed) and 1 = 1 (element at index 1 in the initial permutation). We execute the while loop with the break condition i == 1 not met since i starts at 1. In the loop, we use bitwise operations to find the next index i based on whether it is currently in the first half or the second half

The loop terminates, and the value of ans tells us the minimum number of operations necessary.

Use a loop to simulate the permutation process until the original order is restored.

If the current index is in the second half of the array (from n/2 to n-1),

Increment the operation count each time a permutation is applied.

apply the permutation formula for the first half.

apply the permutation formula for the second half.

operationsCount++; // Increment the operation count

index *= 2; // Double the current index

// it corresponds to the first half of the permuted array

return operationsCount; // Return the number of operations needed

// If the current index is less than n/2,

index = (index - n / 2) * 2 + 1;

If the current index is in the first half of the array (before n/2),

// Loop indefinitely until we break out when the original position is restored

// Otherwise, it corresponds to the second half of the permuted array

// If the element has returned to its original position (index 1),

// Perform the calculation of the new index according to the problem's formula

Python Solution 1 class Solution:

operation_count += 1

if index < (n >> 1):

index <<= 1

```
def reinitializePermutation(self, n: int) -> int:
    # Initialize the number of operations performed to zero.
    operation_count = 0
   # Start with the index of the second element in the permutation.
```

We continue this process and increment ans each time until the element initially at index 1 returns to index 1 again.

Applying bitwise operations makes each calculation within the loop more efficient compared to using standard arithmetic operations.

```
index = ((index - (n >> 1)) << 1) | 1
19
20
21
               # If the index is back to 1, the array has been reinitialized.
22
               # Return the number of operations performed to reach this point.
23
               if index == 1:
24
                   return operation_count
25
Java Solution
   class Solution {
       public int reinitializePermutation(int n) {
           int operationsCount = 0; // Initialize a counter to keep track of the number of operations
           int index = 1; // Start with the element at index 1
```

22 **if** (index == 1) { 23 break; 24 25

```
30
C++ Solution
 1 class Solution {
 2 public:
       // Function to determine the minimum number of operations required
       // to reinitialize a permutation to its original configuration.
       int reinitializePermutation(int n) {
           int numOperations = 0; // Initialize the count of operations to 0
           for (int index = 1; ; ) { // Start with index set to 1
                ++numOperations; // Increment the operation count
               // Check if the current index is in the first half of the array
10
               if (index < (n / 2)) {</pre>
12
                   // If so, double the index
13
                   index *= 2;
14
                } else {
15
                   // Else, calculate the new index based on the second half's rule
16
                    index = (index - (n / 2)) * 2 + 1;
17
               // If the index is back to 1, we've completed the reinitialization
19
20
               if (index == 1) {
21
                    return numOperations; // Return the number of operations needed
22
23
               // Loop continues until index is back to 1
24
25
           // No need for a return statement here, as the loop will eventually return
26
           // the count once the reinitialization condition is met.
27
28 };
```

```
if (index < (n / 2)) {</pre>
               // If so, double the index.
10
               index *= 2;
12
           } else {
13
               // Else, calculate the new index based on the second half's rule.
               index = (index - (n / 2)) * 2 + 1;
14
15
16
           // If the index is back to 1, we've completed the reinitialization.
           if (index === 1) {
               return numOperations; // Return the number of operations needed.
19
20
           // Loop continues until index is back to 1.
22
23
       // No need for a return statement outside of the loop, as the loop will
       // always return the count once the reinitialization condition is met.
24
25 }
26
   // Example usage:
   // const minOperations = reinitializePermutation(4);
Time and Space Complexity
The time complexity of the provided code is O(\log n). This is because each iteration of the while loop either doubles the value of i
(when i < n >> 1) or performs a sequence of operations that ultimately results in looping back towards 1 (when i >= n >> 1). The i
```

value is halved in terms of its position in the original array. Since the value of i is halved in every step, it requires at most log2(n) steps to reach back to the starting position of i = 1. The space complexity of the code is 0(1). This is because the algorithm uses a fixed number of variables (ans, i, n) and does not

allocate any additional space that scales with the input size n. Therefore, the amount of memory used remains constant regardless