Ask a Question

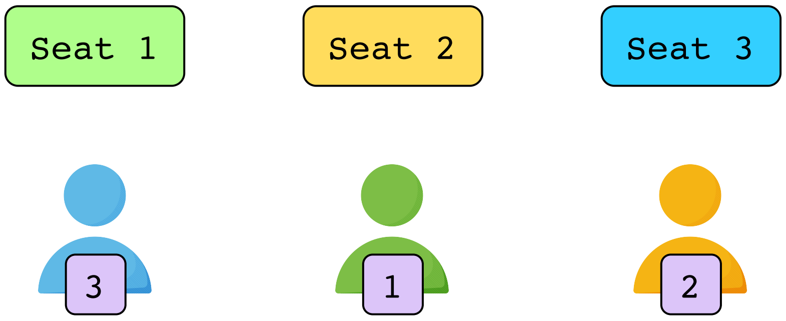
Introduction to Cyclic Sort

Let's go over the Cyclic Sort pattern, its real-world applications, and some problems we can solve with it.

We'll cover the following...

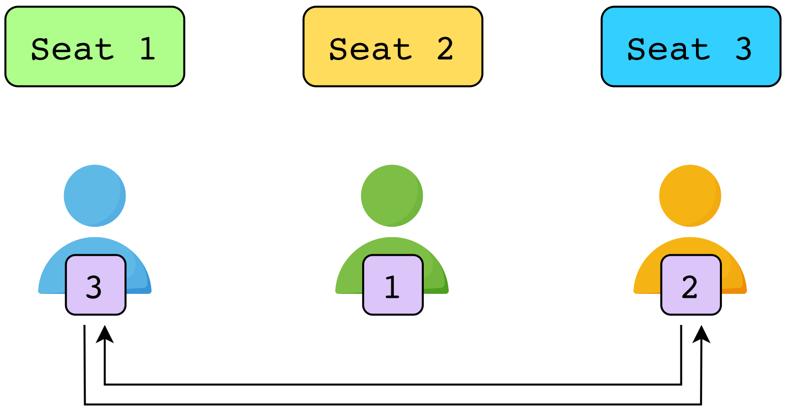
About the pattern

Imagine we have a classroom with numbered seats, and each student is given a card with a number corresponding to their seat number. To maintain classroom decorum and order, students are required to sit in their assigned seats. However, students have randomly taken seats, so the seating arrangement is all jumbled up. If we have to fix this, we start with the first seat, seat 1, and check if the student sitting here has the right card, i.e., the number 1. If not, we move the student to the seat corresponding to their card number. We repeat this process for each seat until every student is in their proper seat according to the number on their card. After completing this process for all seats, the students will be sitting in ascending order according to their seat numbers.



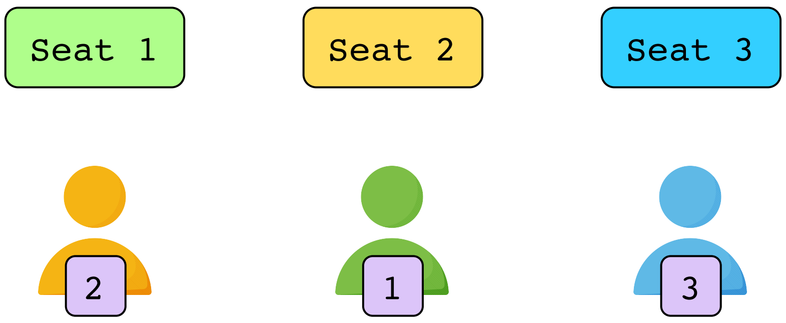
Let's make the students sit in their allocated seats.

1 of 5



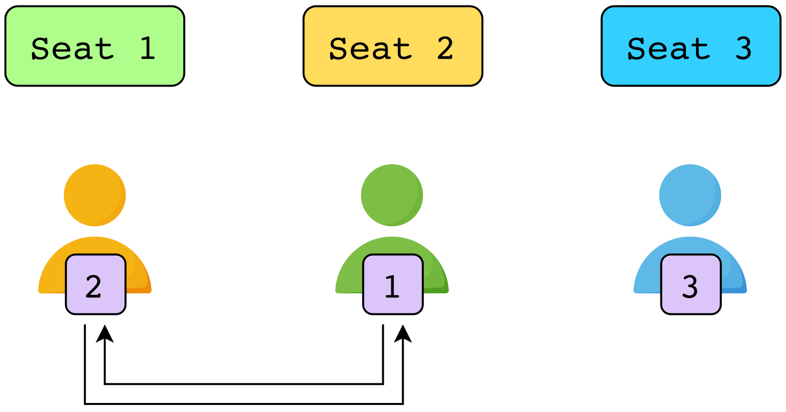
The student with card 3 is in seat 1, so move them to seat 3 and the student in seat 3 to seat 1.

2 of 5



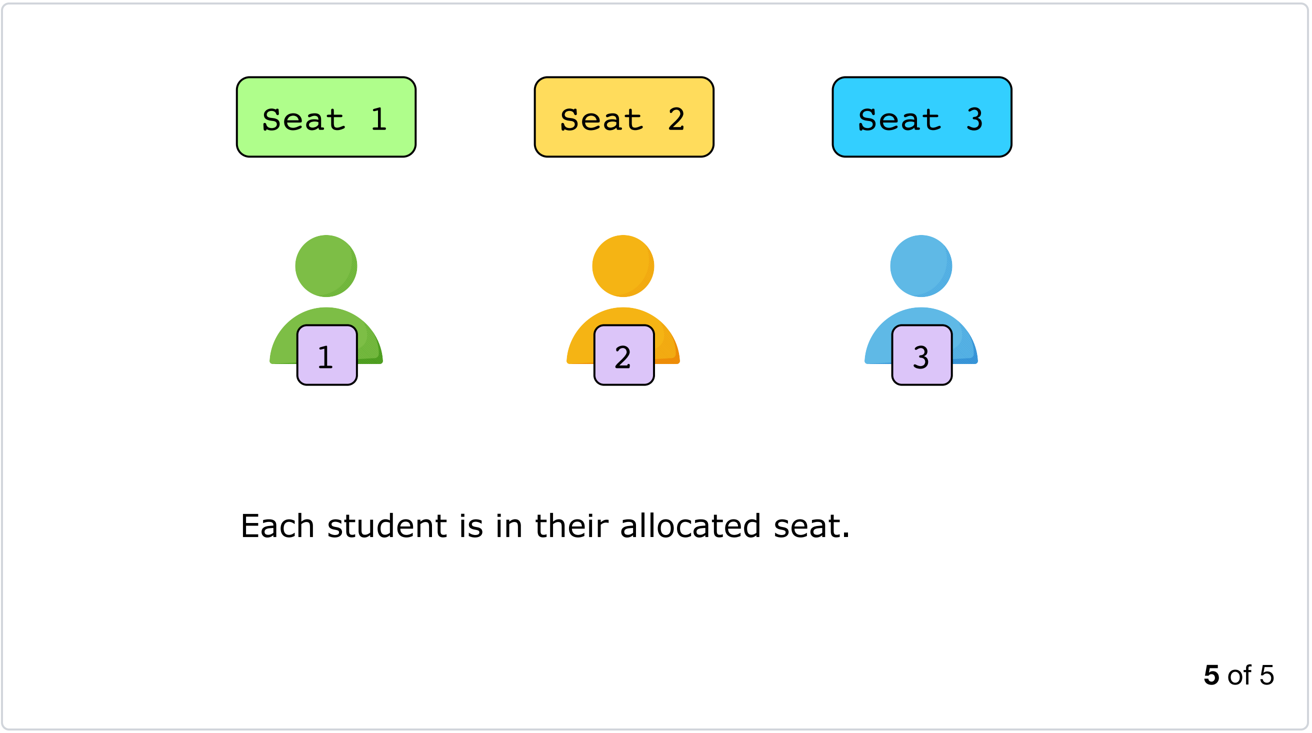
Now, the student with card 3 is in seat 3, and student with card 2 is in seat 1.

3 of 5



The student with card 2 is in seat 1, so move them to seat 2 and the student in seat 2 to seat 1.

4 of 5



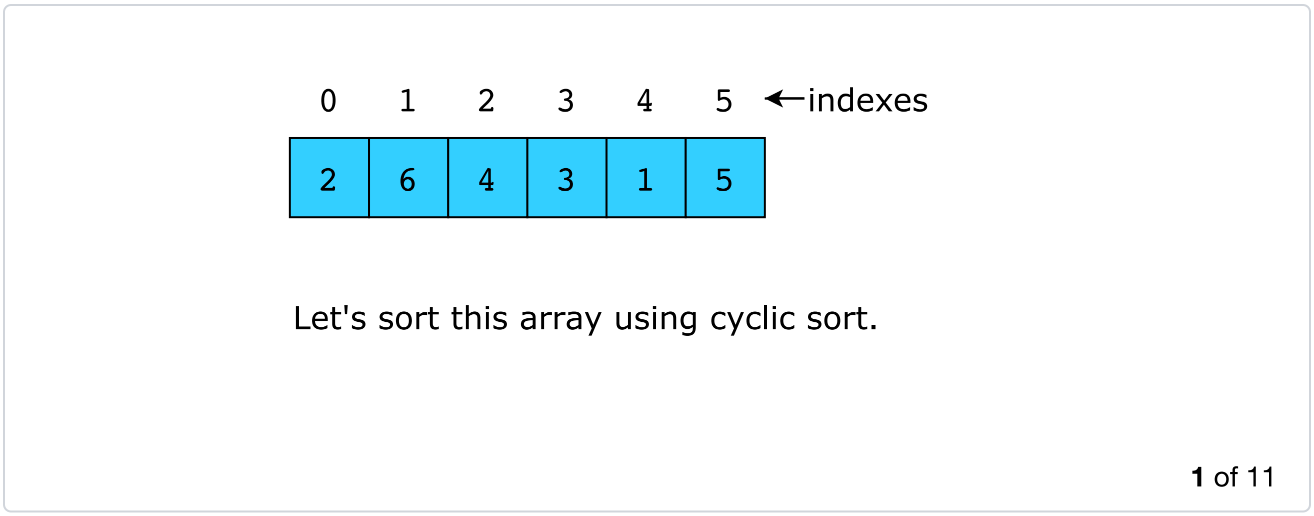
Each student is in their allocated seat.

The repeated swapping of students until they find their correct positions is nothing but the cyclic sort. It can be seen as the cycling of elements through the array until they settle into their proper places.

Cyclic sort is a simple and efficient in-place sorting algorithm used for sorting arrays with integers in a specific range, most of the time $[1-n]$. It places each element in its correct position by iteratively swapping it with the element at the position where it should be located. This process continues until all elements are in their proper places, resulting in a sorted array.

But how do we know the correct position of any element? This is where the algorithm makes things even easier: the correct place of any elemen is simply the value of element - 1. For example, if we have the array $[3, 1, 2]$, then the correct position of the first element, 3, is $(3-1)^{th}$ index, i.e., index 2 and not 0. Similarly, the correct position for the elements 1 and 2 is index 0 and 1, respectively.

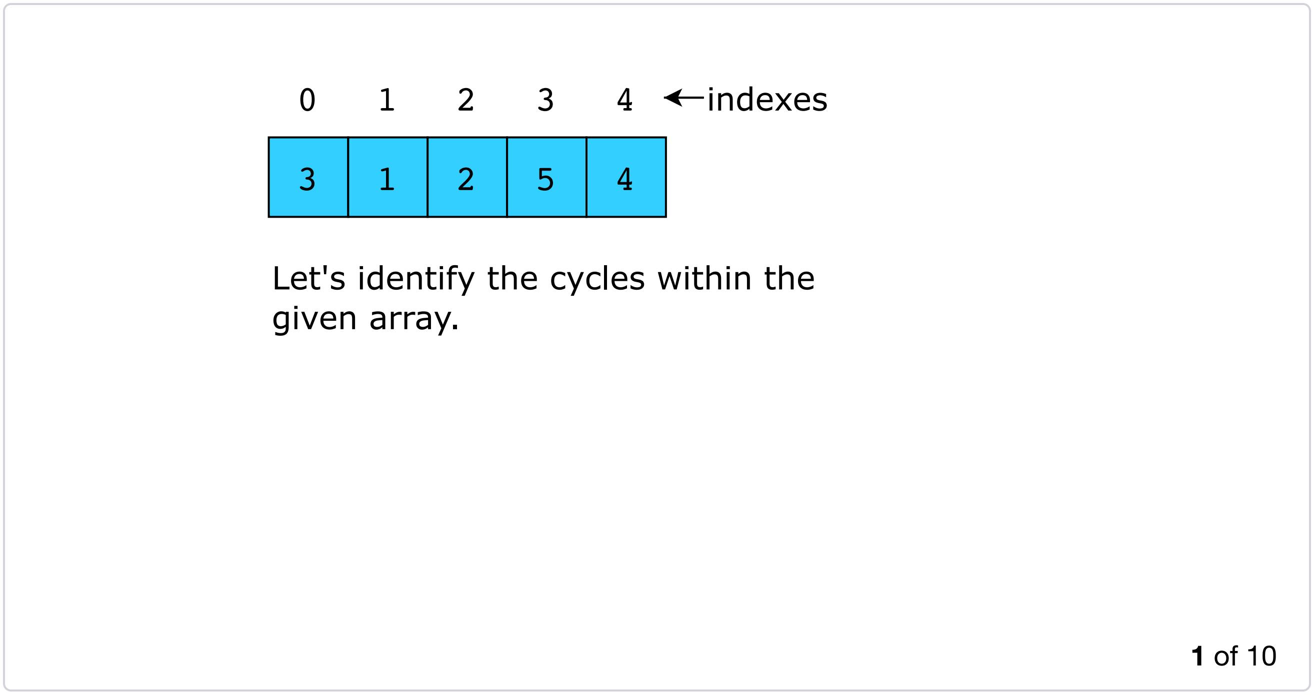
The slides below help visualize how cyclic sort works:



Is there a way to determine whether to use cyclic sort on a given unsorted array of integers? The answer is: Yes. We can identify cycles within the array, which are nothing but the subsequences of numbers that are not in sorted order. A cycle occurs when there’s an element that is out of place, and swapping it with the element at its correct position moves another element out of place, forming a cycle.

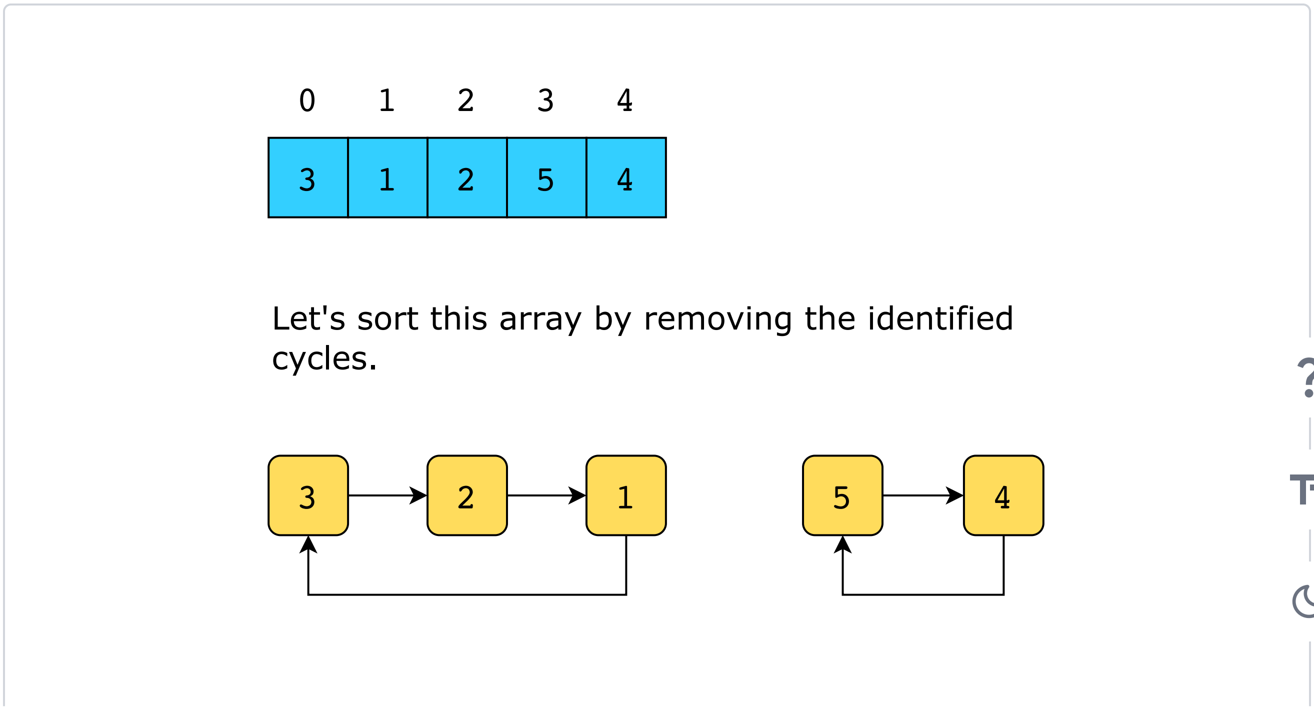
> **Note:** The unsorted arrays with numbers ranging from $[1 - n]$, where n is the length of the array, are guaranteed to have one or more cycles in them.

The slides below illustrate a traversal technique that is utilized for identifying any cycles present within the array.



Let's identify the cycles within the given array.

Once we have identified the cycles within the input array, we can remove them using the cyclic sort. Let's look at an illustration to understand how the cycles are removed.



Let's sort this array by removing the identified cycles.

1 of 9

Note: In practice, the process of identifying cycles and performing cyclic sort is seamlessly integrated and considered as one cohesive operation. The separate illustrations provided are merely for explanatory purposes, simplifying the understanding of the overall process.

Unlike algorithms with nested loops or recursive calls, cyclic sort achieves a linear time complexity of $O(n)$due to its streamlined approach of iteratively placing each element in its correct position within a single pass through the array. This makes it fast and reliable, especially for small arrays. Moreover, cyclic sort doesn't require any extra space, as it sorts the elements in place.

Cyclic sort, while efficient in specific scenarios, does come with its limitations. Let's go over a few:

- **Limited range:** Cyclic sort's efficiency is contingent upon the range of elements being limited and known beforehand. When dealing with arrays with an unknown or expansive range, the cyclic sort may not perform optimally. For example, if we have an array of prices of goods ranging from 1 to 100, cyclic sort would efficiently organize them. However, if the price range extended beyond 100 or was unknown, cyclic sort might not provide optimal results.

- **Not stable:** Cyclic sort lacks stability as a sorting algorithm. This means it may alter the relative order of equal elements during sorting, which could be undesirable in scenarios where maintaining the original order is important. For example, in a task queue where tasks with equal priority should be processed in the order they were received, cyclic sort might reorder tasks with the same priority, disrupting the original sequence.
- **Not suitable for noninteger arrays:** Cyclic sort is optimized for sorting arrays of integers, attempting to use it on noninteger arrays may not produce the desired outcome. Suppose we want to sort an array of names alphabetically, then using cyclic sort is not a good choice.
- **Not suitable when multiple attributes play a significant role:** Cyclic sort is primarily designed for arrays of integers only, so it may not handle cases where the input has multiple attributes associated with it. For example, given an array containing objects representing employees, where each object has attributes such as name, age, and salary, if we need to sort the objects with respect to all three attributes then using cyclic sort on just salary may not produce the desired outcome. This is because the other two attributes play an equal role in deciding the final order, so we can’t just take one attribute while sorting the array.

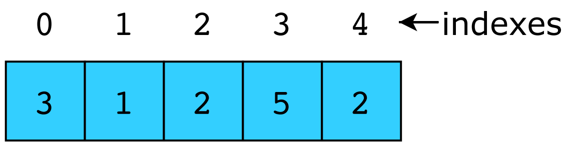
Normally, it is quite easy to know when to terminate the algorithm for cyclic sort. If we have the input array of length n containing all the numbers from 1 to n, then the cyclic sort would end once every element of the array has been traversed from index 0 to $n-1$. However, it gets tricky to determine the termination when the input array of length n doesn’t contain every number from 1 to n. In any such scenario, we’ll end up either with an infinite loop or incorrect placing of elements in their respective positions. In both cases, it’s essential to incorporate appropriate error-handling mechanisms or termination conditions within the cyclic sort algorithm to prevent infinite loops and ensure correct sorting outcomes. This may involve additional checks for duplicates or missing values, as well as adjustments to the sorting logic to accommodate such scenarios.

> **Note:** The examples in the section below represent two such scenarios where termination of cyclic sort requires an additional check.

Examples

The following examples illustrate some problems that can be solved with this approach:

1. **Set mismatch:** Given a set of numbers from 1 to n, we have to find one number
   that is repeated, and another number that is missing from the set.

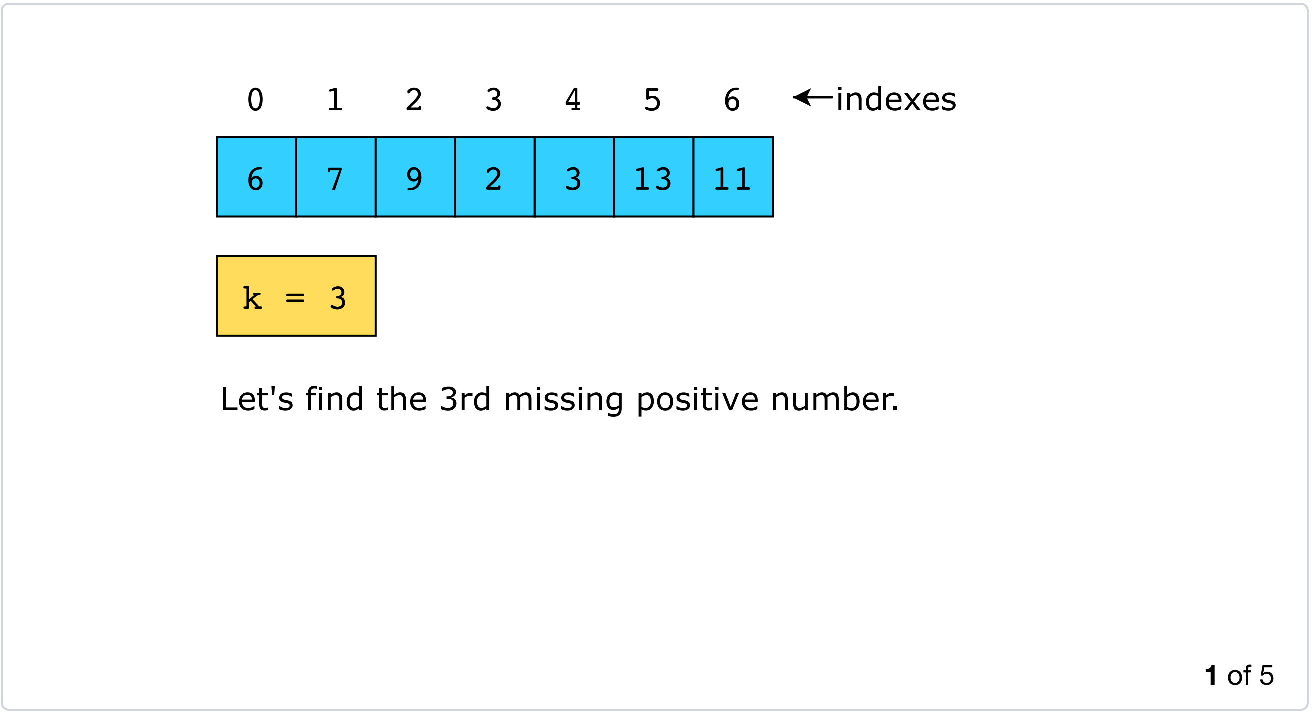


We solve this problem by placing the elements at their correct indexes.

To place any element at its correct index, we swap it with the element that is already there.

1 of 12

2. **Find the Kth missing positive number:** Given an array of numbers, we have to find the k^{th} missing number from the list.



Let's find the 3rd missing positive number.

1 of 5

Does your problem match this pattern?

- Yes, if either of these conditions are fulfilled:
  - **Limited range integer arrays:** The problem involves an input array of integers in a small range, usually $[1 - n]$.
  - **Finding missing or duplicate elements:** The problem requires us to identify missing or duplicate elements in an array.
- No, if any of these conditions is fulfilled:

- **Noninteger values:** The input array contains noninteger values.
- **Nonarray format:** The input data is not originally in an array, nor can it be mapped to an array.
- **Stability requirement:** The problem requires stable sorting.

Real-world problems

Many problems in the real world share the cyclic sort pattern. Let's look at some examples.

- **Computational biology:** The species on a planet have n distinct genes numbered 1…n. Find the kth missing gene in a given DNA sequence.
- **Playing card sorting:** If we have a deck of playing cards represented as integers in the range $[1 - 52]$, and they are randomly shuffled, we can use cyclic sort to rearrange the cards into sorted order efficiently.
- **Data validation:** Cyclic sort can be used for data validation tasks, especially when dealing with datasets that are expected to have distinct values within a certain range. By applying cyclic sort to the dataset, we can quickly identify any missing or duplicate values, which can help in validating the integrity and accuracy of the data.
- **Package delivery routing:** In logistics or package delivery systems, drivers may have a list of addresses to visit. We can map these addresses to integers in a defined range. Then, we can use cyclic sort to optimize the route by rearranging the addresses based on these mapped integers, thus minimizing travel time and fuel consumption.

Strategy time!

Match the problems that can be solved using the cyclic sort pattern.

> **Note:** Select a problem in the left-hand column by clicking it, and then click one of the two options in the right-hand column.

Match The Answer

Select an option from the left-hand side

Given a string representing a number, return the closest number that is a palindrome.	Cyclic Sort
Given an array of numbers	Some other pattern