1539. Kth Missing Positive Number

linear search that would have a time complexity of O(n).

Let's walk through the steps of the algorithm using the provided Python code:

**Binary Search** 

## **Problem Description**

integer that is not present in the array arr. For example, if the array is [2, 3, 4, 7, 11] and k is 5, we need to find the 5th positive integer missing from this sequence. The missing numbers are [1, 5, 6, 8, 9, 10,...], and the 5th one is 9, which will be our answer.

The problem provides us with an array arr which contains positive integers in increasing order. Note that the array might not

contain some positive integers; hence it's not consecutive. We're also given an integer k. Our goal is to find the kth positive

Intuition

Easy

To find the kth missing positive integer, we're using a binary search algorithm to optimize the process. Binary search helps us reduce the search space to find the answer quickly, instead of inspecting each missing number one by one—which would be less efficient.

increasing, the number of positive integers missing before any array element arr[i] can be found as arr[i] - i - 1. This is because if there were no missing numbers, the value at arr[i] would be i + 1. The binary search algorithm exploits this by repeatedly halving the array to find the smallest arr[i] such that arr[i] - i - 1 is

The essence of the solution lies in understanding how we can identify if a number is kth missing or not. Since the array is strictly

still at least k. The algorithm keeps moving the left or right boundaries according to the comparison of arr[mid] - mid - 1 with k.

Once the left boundary crosses the right boundary, we know that the missing number we are looking for is not in the array. It must be between arr[left - 1] and arr[left] (or after arr[left - 1] if left is equal to the length of the array). Hence, the answer can be computed by adding k to arr[left - 1] - (left - 1) - 1 which is the number of missing numbers before arr[left - 1].

This solution has a time complexity of O(log n), where n is the number of elements in arr, which is significantly faster than a

**Solution Approach** The implemented solution employs a binary search algorithm to efficiently locate the kth missing positive integer in the sorted

array arr. Binary search is a popular algorithm for finding an item in a sorted list by repeatedly dividing the search interval in half.

## 1. Check if k is less than the first element: Before starting the binary search, it's checked whether arr [0] is greater than k. If it is, k itself is the kth missing number since all k missing numbers are before arr[0]. The code returns k directly in this case.

**if** arr[0] > k:

return k

right to mid.

right = mid

left = mid + 1

else:

which is 5.

**Python** 

Java

class Solution {

#include <vector>

class Solution {

public:

class Solution:

from typing import List

**if** arr[0] > k:

return k

while left < right:</pre>

left, right = 0, len(arr)

return kth\_missing\_positive

public int findKthPositive(int[] arr, int k) {

int findKthPositive(std::vector<int>& arr, int k) {

if (arr[mid] - mid - 1 >= k) {

function findKthPositive(arr: number[], k: number): number {

// number must be k itself.

**if** (arr[0] > k) {

return k;

while (left < right) {</pre>

} else {

right = mid;

int left = 0;

// If the first element in the array is greater than k, the kth missing

int right = arr.size(); // The right boundary for the binary search.

int mid = left + (right - left) / 2; // Prevents potential overflow.

// If the number of missing numbers up to arr[mid] is at least k,

left = mid + 1; // Otherwise, we search on the right side.

// If the first element in the array is greater than k, then the kth missing number must be k itself.

// After the loop, left is the smallest index such that the number of

mid = (left + right) // 2 # Use integer division for Python 3

# If the number of missing elements is greater or equals k, look in the left half

# Calculate the number of negative elements up to index mid

left = mid + 1 # Otherwise, look in the right half

# and then adjust it by subtracting the missing count until that point

kth missing positive = arr[left - 1] + k - missing\_until\_left\_minus\_one

# After binary search, calculate the k-th missing positive number

# by adding k to the number at the index `left - 1` in the array

missing until left minus one = arr[left - 1] - (left - 1) - 1

missing until mid = arr[mid] - mid - 1

if missing until\_mid >= k:

right = mid

return kth\_missing\_positive

Time and Space Complexity

// Binary search to find the lowest index such that the number of

// positive integers missing before arr[index] is at least k.

// we need to search on the left side (including mid).

2. Initialize the binary search boundaries: The variables left and right are initialized to represent the search space of the binary search, where left is the start index (0) and right is the length of the array arr. left, right = 0, len(arr)

Perform Binary Search: The binary search loop continues until left is less than right. In each iteration, a midpoint mid is

calculated. The algorithm checks the number of missing numbers up to arr[mid] by calculating arr[mid] - mid - 1.

o If the calculated number of missing elements is greater than or equal to k, it means the kth missing number is before or at mid. We set

```
\circ If the number is less than k, we move left forward to mid + 1.
while left < right:</pre>
    mid = (left + right) >> 1
    if arr[mid] - mid - 1 >= k:
```

numbers are missing up to arr[left - 1], and then add k to reach the kth missing number. return arr[left - 1] + k - (arr[left - 1] - (left - 1) - 1)

This formula takes the last known value before the kth missing number, adds k, and then subtracts the count of missing

This approach results in an effective solution with a time complexity of O(log n), leveraging the power of binary search to

4. Calculate and Return the Missing Number: After the loop, the kth missing number is not present in the array, and it must be found after the

number at index left - 1. To find it, the formula arr[left - 1] + k - (arr[left - 1] - (left - 1) - 1) is used to calculate how many

The >> 1 is a bitwise operation equivalent to dividing by 2, efficiently calculating the mid index.

numbers before the arr[left - 1] to land exactly on the kth missing number.

drastically reduce the potential search space compared to more naive approaches.

move on to the binary search. No numbers are returned in this initial step.

k - (arr[left - 1] - (left - 1) - 1) which equals 11 + 5 - 7 = 9.

# If the first element is larger than k, the k-th positive missing number would be k

```
Example Walkthrough
  Let's illustrate the solution approach with an example. Consider the array arr = [2, 3, 7, 11, 12], and we want to find the 5th
  missing positive integer (k = 5).
```

Initial check: We first check if the first element of the array is greater than k. Since arr[0] = 2 is not greater than 5, we

Set up binary search: We then initialize the binary search boundaries with left set to 0 and right to the length of the array,

∘ Second iteration: New midpoint is mid = (3 + 5) >> 1 = 4. At arr[4] = 12, the missing count is 12 - 4 - 1 = 7, which is greater than k. Now we update right to mid = 4. Third iteration: As left < right no longer holds (since left = 4 and right = 4), we exit the loop.</li>

∘ First iteration: Calculate mid = (0 + 5) >> 1 = 2. At arr[2] = 7, the count of missing numbers is 7 - 2 - 1 = 4, which is less than k =

**Determine answer:** We are now left with left = 4. The number in the array at index left - 1 = 3 is arr[3] = 11. From

arr[3], we have 11 - 3 - 1 = 7 missing numbers before it. The kth missing number is then calculated as arr[left - 1] +

Hence, the 5th missing positive integer is 9. This matches our expectation because the missing numbers before 9 are [1, 4,

Solution Implementation

def findKthPositive(self, arr: List[int], k: int) -> int:

# Use binary search to find k-th positive missing number

mid = (left + right) // 2 # Use integer division for Python 3

# After binary search, calculate the k-th missing positive number

# and then adjust it by subtracting the missing count until that point

kth missing positive =  $arr[left - 1] + k - missing_until_left_minus_one$ 

// If the first element in the array is greater than k, the kth missing

# by adding k to the number at the index `left - 1` in the array

missing until left minus one = arr[left - 1] - (left - 1) - 1

5, 6, 8], and 9 is indeed the fifth missing number.

**Binary search**: Next, we begin the binary search:

o Initial values are left = 0 and right = 5.

5. Thus, we update left to mid + 1 = 3.

```
# Calculate the number of negative elements up to index mid
missing_until_mid = arr[mid] - mid - 1
# If the number of missing elements is greater or equals k, look in the left half
if missing until_mid >= k:
    right = mid
else:
    left = mid + 1 # Otherwise, look in the right half
```

```
// positive number would just be k, since all numbers before arr[0] are missing
        if (arr[0] > k) {
            return k;
        // Initializing binary search boundaries
        int left = 0. right = arr.length;
        while (left < right) {</pre>
            // Finding the middle index using bitwise operator to avoid overflow
            int mid = (left + right) >> 1;
            // If the number of missing numbers until arr[mid] is equal to or greater than k
            // then the kth missing number is to the left of mid, including mid itself
            if (arr[mid] - mid - 1 >= k) {
                right = mid;
            } else {
                // Otherwise, the kth missing number is to the right of mid, so we move left
                left = mid + 1;
        // Once left is the smallest index such that the number of missing numbers until arr[left]
        // is less than k, the kth positive integer that is missing from the array is on the right
        // of arr[left-1]. To find it, we add k to arr[left-1] and then subtract the number of
        // missing numbers until arr[left-1] (which is arr[left-1] - (left-1) - 1).
        return arr[left - 1] + k - (arr[left - 1] - (left - 1) - 1);
C++
```

### // positive integers missing before arr[left] is at least k. Using the // index left - 1, we find the kth missing number. return arr[left - 1] + k - (arr[left - 1] - (left - 1) - 1); **}**;

**TypeScript** 

**if** (arr[0] > k) {

return k;

while left < right:</pre>

else:

```
let left = 0;
    let right = arr.length; // The right boundary for the binary search.
   // Binary search to find the lowest index such that the number of
   // positive integers missing before arr[index] is at least k.
   while (left < right) {</pre>
        let mid = left + Math.floor((right - left) / 2); // Math.floor is used to prevent floats since TypeScript does not do integer
       // If the number of missing numbers up to arr[mid] is at least k,
       // we need to search on the left side (including mid).
        if (arr[mid] - mid - 1 >= k) {
            right = mid;
        } else {
            left = mid + 1; // Otherwise, we search on the right side.
   // After the loop, left is the smallest index such that the number of
   // positive integers missing before arr[left] is at least k. Using the
   // index left - 1, we find the kth missing number.
   return arr[left - 1] + k - (arr[left - 1] - (left - 1) - 1);
from typing import List
class Solution:
   def findKthPositive(self, arr: List[int], k: int) -> int:
       # If the first element is larger than k, the k-th positive missing number would be k
       if arr[0] > k:
           return k
       # Use binary search to find k-th positive missing number
        left, right = 0, len(arr)
```

# **Time Complexity**

The provided code uses a binary search algorithm to find the k-th positive integer that is missing from the array arr. This is evident from the while loop that continues halving the search space until left is less than right. In a binary search, the time complexity is O(log n) where n is the number of elements in arr, because with each comparison, the search space is reduced by half.

**Space Complexity** 

The space complexity of the code is O(1) since there are only a few variables used (left, right, mid, k), and no additional data structures or recursive calls that would require more space proportional to the input size. The algorithm operates in-place with constant extra space.