#### 2424. Longest Uploaded Prefix Medium Ordered Set **Union Find** Design **Heap (Priority Queue) Binary Indexed Tree Segment Tree Binary Search** Leetcode Link

# **Problem Description** In this problem, we are tasked with simulating the upload process of a series of videos to a server. Each video has a unique identifier

ranging from 1 to n. Our goal is to keep track of the longest continuous sequence of uploaded videos, starting from video 1. To express this sequence, we refer to it as the "longest uploaded prefix," where a prefix is defined as a sequence of videos from 1 to i, inclusive, where i represents the end of this prefix. All the videos within that range must have been successfully uploaded for the prefix to be considered complete. The following operations need to be implemented within a LUPrefix class:

• LUPrefix(int n): Construction of the data structure to handle n videos. void upload(int video): Handles the upload of a video represented by an integer.

- int longest(): Returns the length of the longest uploaded prefix after the various uploads that occurred since the last check or
- initialization.
- The main challenge of this problem is efficiently updating and finding the longest uploaded prefix as each new video is uploaded, especially when uploads might happen out of order.

Intuition

The solution to this problem requires a strategy to efficiently track the videos that are uploaded and to quickly determine the longest

#### uploaded prefix. A naive approach would be to check every video from 1 to n whenever a new video is uploaded, but this would result in a lot of unnecessary work since we would be repeatedly checking videos we have already verified.

The intuition behind this approach is to use a combination of a counter and a set to track the uploads. self.r represents the rightmost edge of the longest uploaded prefix found so far, and the set self.s is responsible for keeping track of the individual videos that have been uploaded. Whenever a new video is uploaded, we add it to the set. Then, we simply check if the video next to

the current rightmost edge (self.r + 1) has been uploaded. If it has, we update the rightmost edge by incrementing self.r and

continue this process until the next video in the sequence is missing from the set. This way, we can efficiently maintain the length of the longest uploaded prefix by only checking the next video in line whenever a new upload occurs. This strategy is efficient because the set provides constant time (0(1)) operations for adding videos and checking for their existence. The counter self.r minimizes work by only considering the immediate next candidate for the longest prefix, rather than starting from the beginning every time. By doing this, we ensure that each upload can be processed in 0(1) average time complexity, given that the while loop in upload will

self.r. **Solution Approach** 

run at most n times across all calls (amortized analysis), and each call to longest() will be 0(1) as it simply returns the value of

The solution for the Longest Uploaded Prefix problem uses a simple yet effective algorithm that revolves around two main components: a counter (self.r) and a set (self.s). Here's how each part of the LUPrefix class works: 1. Initialization (\_\_init\_\_ method): The constructor initializes two attributes:

### • self.r starts at 0, which indicates that initially, there is no uploaded prefix as we start counting from 1. self.s is an empty set that will store the video numbers as they are uploaded.

contiguous uploaded prefix.

a constant time operation.

2. Upload (upload method): Every time a video is uploaded, its number is added to the set self.s. After adding it to the set, we check if the next sequential video (self.r + 1) is in the set:

• We continue incrementing self.r and checking the next number in the sequence until we reach a number that isn't in the set. This is handled by the while loop which only runs as long as the next video in sequence is found in the set, indicating a

• If it is, we increment self.r by 1, since this extends our uploaded prefix.

contiguous videos got uploaded in a batch. With the loop, we can "catch up" and expand the longest uploaded prefix as needed. 3. Longest (longest method): This method simply returns the current value of self.r, which, by the design of the upload method, is always the length of the longest uploaded prefix. Since self.r is always kept up-to-date whenever upload is called, longest is

The use of a while loop here is essential, as it could be the case that earlier uploads were not sequential, and multiple

since the updates to self.r in the while loop will be distributed over many calls to the upload method. By keeping track of the uploaded videos and the length of the longest uploaded prefix in such a manner, we ensure efficient uploads

quick update of the longest uploaded prefix. The loop inside the upload method guarantees that the counter self.r correctly reflects

Overall, the algorithm takes advantage of the properties of a set to efficiently handle insertions and lookups, thereby enabling a

the longest sequence after each video upload. The amortized time complexity for each upload operation can be considered 0(1)

and instantaneous retrievals of the longest uploaded prefix length, which makes the solution both elegant and practical for the

Example Walkthrough Let's consider a scenario where we have a sequence of 7 videos to upload, identified by numbers from 1 to 7. According to the solution approach, we will walk through a series of calls to the upload and longest methods to illustrate how the solution works.

• We create a LUPrefix object for 7 videos, initializing self.r to 0 and self.s to an empty set.

# • We call upload(3), which adds video 3 to self.s. The set now contains {3}. Since self.r is 0 and the next video in line

2. Upload video 3:

3. Upload video 1:

1 lup.upload(1)

5. Upload video 2:

6. Upload video 5:

lup.upload(5)

7. Upload video 4:

1 lup = LUPrefix(7)

1. Instantiate LUPrefix object:

problem at hand.

(video 1) has not been uploaded, self.r remains unchanged. 1 lup.upload(3) print(lup.longest()) # Output: 0

Next, we call upload(1). Video 1 is added to self.s, which now contains {1, 3}. This time, because video 1 is the next in the

sequence from self.r (self.r + 1), we increment self.r to 1. We then check if video 2 (self.r + 1) is in the set, but it's

```
    Calling longest() returns 1, since we now have the first video uploaded.
```

4. Check the longest uploaded prefix:

2 print(lup.longest()) # Output: 1

2 print(lup.longest()) # Output: 3

2 print(lup.longest()) # Output: 5

would be updated accordingly.

self.uploaded\_videos.add(video)

self.next\_video += 1

return self.next\_video - 1

def longest(self) -> int:

public LUPrefix(int n) {

\* Processes the upload of a video.

public void upload(int video) {

public int longest() {

return consecutiveRange;

unordered\_set<int> uploadedVideos;

// int lengthOfSequence = obj->longest();

// LUPrefix\* obj = new LUPrefix(n);

// obj->upload(video);

Typescript Solution

// Usage example (not part of submitted code, just for reference):

// A set to keep track of which videos have been uploaded.

40 };

41

46

not, hence self.r stops at 1.

sequential video in the set, we increment self.r to 2. Immediately afterward, we check if video 3 is in the set (it is), so we increment self.r to 3, completing a prefix of {1, 2, 3}. 1 lup.upload(2) 2 print(lup.longest()) # Output: 3

Uploading video 5 with upload(5) adds it to the set, now {1, 2, 3, 5}. self.r remains at 3 since video 4 is not in self.s.

• We call upload(2) and add video 2 to self.s, which becomes {1, 2, 3}. Since self.r is 1, and now video 2 is the next

 By uploading video 4, the set becomes {1, 2, 3, 4, 5}. Now that video 4 has been uploaded, we can extend self.r to 4, and subsequently to 5, since the next in the sequence, video 5, is also present. 1 lup.upload(4)

# Adds the uploaded video to the set of uploaded videos.

while self.next\_video in self.uploaded\_videos:

\* Constructor which initializes the LUPrefix object.

\* @param video Number of the video being uploaded.

\* @return The length of the longest consecutive range.

# Loop to find the longest consecutive sequence starting from the next video.

# Because next\_video is always looking for the next video to watch,

## contiguous uploaded video sequence efficiently. We can see how the upload method updates self.r only when necessary, avoiding redundant checks and enabling a quick return from the longest method.

6

10

11

12

13

14

15

16

17

18

19

8

9

10

11

12

13

14

15

16

17

18

19

20

26

27

28

29

30

31

32

33

35

36

34 }

/\*\*

**Python Solution** 

8. Final state:

class LUPrefix: def \_\_init\_\_(self, n: int): self.next\_video = 1 # The number representing the next video in the sequence to be watched. self.uploaded\_videos = set() # A set to keep track of all uploaded videos. def upload(self, video: int) -> None:

# Increment next\_video since the current next\_video has been found in the uploaded videos.

private int consecutiveRange; // tracks the longest consecutive range of uploaded videos starting from 1

# Returns the latest video number up to which all videos have been uploaded in sequence.

# we return next\_video - 1 to indicate the last video that can be watched in sequence.

private Set<Integer> uploadedVideos = new HashSet<>(); // stores the uploaded video numbers

\* @param n Total number of videos (not used since we are determining the range dynamically).

// The constructor does not need any logic since we're managing videos dynamically.

// Videos start being relevant from the first upload, so no setup required.

uploadedVideos.add(video); // Add the uploaded video number to the set

\* Returns the longest consecutive range of videos starting from video number 1.

The longest() method continues to return 5, as videos 6 and 7 have not been uploaded. If they were uploaded, self.r

Through this example, it is evident that the set and counter mechanism within the LUPrefix class allows for tracking the longest

Java Solution class LUPrefix {

#### 21 // Check if the next consecutive video is already uploaded. If so, increment the range. 22 while (uploadedVideos.contains(consecutiveRange + 1)) { 23 consecutiveRange++; 24 25

/\*\*

/\*\*

/\*\*

```
* Sample usage of the LUPrefix class is as follows, though it is not expected to be part of the rewritten code:
    * LUPrefix obj = new LUPrefix(n); // Instantiate with the total number of videos n.
                                      // Call upload() method whenever a video is uploaded.
    * obj.upload(video);
    * int param_2 = obj.longest(); // Call longest() method to get the longest consecutive uploaded video range starting from 1.
41
    */
42
C++ Solution
   #include <unordered_set>
   // The LUPrefix class is designed to handle video uploads and track the longest
  // consecutive sequence of videos starting from 1 without any gaps.
5 class LUPrefix {
6 public:
       // Constructor initializes the object with the total number of videos.
       // Since we're tracking the longest sequence from the beginning, no other setup is needed.
       LUPrefix(int n) {
9
           // No need to use 'n' as the problem defined doesn't store all videos,
           // just tracks the consecutive sequence from the start.
11
12
13
       // upload is a method to simulate the uploading of a video.
14
       // The video number is added to a set for tracking and the longest consecutive
15
       // sequence is updated.
16
       void upload(int video) {
17
           // Insert the video into the set.
18
19
           uploadedVideos.insert(video);
20
21
           // Check if the next consecutive video number is present;
           // if so, increment the counter for the longest sequence.
22
23
           while (uploadedVideos.count(longestSequence + 1)) {
24
               ++longestSequence;
25
26
27
28
       // longest returns the length of the longest consecutive sequence of videos
       // starting from video number 1.
       int longest() {
30
31
           // Return the current counter which is the length of the longest sequence.
32
           return longestSequence;
33
34
   private:
       // Private data member to track the longest sequence starting from video 1.
       int longestSequence = 0;
37
       // A set to keep track of which videos have been uploaded.
38
```

```
const uploadedVideos: Set<number> = new Set();
   // Variable to track the longest sequence starting from video 1.
   let longestSequence: number = 0;
   /**
    * Simulates the upload of a video and updates the longest consecutive sequence.
    * @param {number} video - The number of the video being uploaded.
10
11 function upload(video: number): void {
       // Insert the video number into the set of uploaded videos.
13
       uploadedVideos.add(video);
14
       // Check if the next consecutive video number is present;
15
       // if so, increment the counter for the longest sequence.
16
       while (uploadedVideos.has(longestSequence + 1)) {
            longestSequence++;
19
20 }
21
22
   /**
    * Gets the length of the longest consecutive sequence of videos
    * starting from video number 1.
    * @return {number} The current length of the longest sequence.
26
    */
   function longest(): number {
       // Return the current length of the longest continuous sequence from the start.
28
29
       return longestSequence;
30
31
   // Example usage:
   // upload(1);
   // upload(3);
  // upload(2);
   // const lengthOfSequence: number = longest(); // This would return 3
37
Time and Space Complexity
```

## The upload method has a time complexity of 0(1) for adding a video to the set (self.s.add(video)). However, the while loop checking for the next video (while self.r + 1 in self.s) can iterate up to n times in the worst-case scenario, where n is the

**Time Complexity** 

number of videos. Therefore, in the worst-case scenario, the time complexity of upload could be O(n), but the amortized time complexity over a sequence of operations is 0(1) since each video gets uploaded only once. The longest method simply returns the current value of self.r, which is a direct access operation with a time complexity of 0(1).

# **Space Complexity** The space complexity is dominated by the set self.s that stores the uploaded videos. In the worst-case scenario, when all n videos

have been uploaded, the space complexity is O(n), where n is the number of videos. The rest of the variables use constant space.