Documentation

The problem involves maintaining a dynamic list of disjoint intervals as numbers are added to a stream. The core challenge is efficiently managing and merging intervals when necessary while ensuring the list remains sorted and non-overlapping. As each number is added, it may either form a new interval, merge with existing intervals, or connect two separate intervals into one. The key idea is to insert the number into the correct position, check for potential merges with neighboring intervals, and then update the list accordingly. This ensures that the intervals are always disjoint, sorted, and up-to-date.

To solve this, a sorted data structure is employed, such as SortedList, which allows for efficient insertion and merging of intervals. When a new number is added, the algorithm finds its correct position in the list and checks for neighboring intervals that may need to be merged. The merging process involves combining intervals that are adjacent or overlapping and updating the list to reflect the newly merged intervals. This approach minimizes unnecessary operations, allowing the solution to handle dynamic interval updates efficiently.

The time complexity of this solution is dominated by the insertion operation, which is logarithmic, O(log n), where n is the number of intervals. This is due to the use of a sorted data structure that allows for efficient searching and insertion. After insertion, checking for merges involves inspecting at most two intervals (before and after the inserted number), making it a constant-time operation. As a result, the overall time complexity for adding a number is O(log n), while retrieving the current intervals takes linear time, O(n).

In terms of space complexity, the solution requires O(n) space to store the intervals, where n is the number of disjoint intervals at any given point. This space complexity is optimal because the intervals need to be stored explicitly, and their number grows as more numbers are added to the stream. Each interval is represented by a tuple, and the number of intervals increases as new numbers are inserted and possibly merged.

One of the challenges addressed by this approach is efficiently handling the merging of intervals when a new number is added. The solution ensures that when an interval connects with an existing one or extends an interval, the merging happens in constant time. After each insertion, the intervals are kept disjoint and sorted, without the need for reordering the entire list, making the process more efficient. This is particularly useful when many intervals need to be merged, reducing the computational overhead.

Edge cases are also handled effectively. For instance, when a number is added that falls outside of all existing intervals, it simply forms a new interval. If the number connects two separate intervals, they are merged into one. If the number falls within an existing interval, no new intervals are created. These scenarios are efficiently managed by checking and merging intervals only when necessary, ensuring that the intervals remain correct and minimal.

In conclusion, this solution offers an efficient and scalable approach to managing a data stream of numbers as disjoint intervals. By using a sorted data structure, the solution can quickly insert new numbers, merge intervals when necessary, and maintain the list of disjoint intervals in sorted order. With logarithmic time complexity for insertion and linear time complexity for retrieving the intervals, the approach strikes a balance between performance and correctness. This method is well-suited for handling dynamic data streams where intervals need to be updated frequently and efficiently.