<https://leetcode.com/problems/data-stream-as-disjoint-intervals/description/>

Given a data stream input of non-negative integers a1, a2, ..., an, summarize the numbers seen so far as a list of disjoint intervals.

Implement the SummaryRanges class:

SummaryRanges() Initializes the object with an empty stream.

void addNum(int value) Adds the integer value to the stream.

int[][] getIntervals() Returns a summary of the integers in the stream currently as a list of disjoint intervals [starti, endi]. The answer should be sorted by starti.

**Example 1:**

**Input**

["SummaryRanges", "addNum", "getIntervals", "addNum", "getIntervals", "addNum", "getIntervals", "addNum", "getIntervals", "addNum", "getIntervals"][[], [1], [], [3], [], [7], [], [2], [], [6], []]

**Output**

[null, null, [[1, 1]], null, [[1, 1], [3, 3]], null, [[1, 1], [3, 3], [7, 7]], null, [[1, 3], [7, 7]], null, [[1, 3], [6, 7]]]

**Explanation**

SummaryRanges summaryRanges = new SummaryRanges();

summaryRanges.addNum(1); // arr = [1]

summaryRanges.getIntervals(); // return [[1, 1]]

summaryRanges.addNum(3); // arr = [1, 3]

summaryRanges.getIntervals(); // return [[1, 1], [3, 3]]

summaryRanges.addNum(7); // arr = [1, 3, 7]

summaryRanges.getIntervals(); // return [[1, 1], [3, 3], [7, 7]]

summaryRanges.addNum(2); // arr = [1, 2, 3, 7]

summaryRanges.getIntervals(); // return [[1, 3], [7, 7]]

summaryRanges.addNum(6); // arr = [1, 2, 3, 6, 7]

summaryRanges.getIntervals(); // return [[1, 3], [6, 7]]

**Constraints:**

0 <= value <= 10^4

At most 3 \* 10^4 calls will be made to addNum and getIntervals.

At most 10^2 calls will be made to getIntervals.

**Follow up:**

What if there are lots of merges and the number of disjoint intervals is small compared to the size of the data stream?

**Attempt 1: 2024-01-06**

**Solution 1: Insert + Merge Interval (60min)**

**This solution naturally based on L56.Merge Intervals + L57.Insert Intervals, the improvement from L56 / L57 template is we merge original intervals into one interval and remove original intervals at the same time (in same for loop iteration), not like L56 / L57, we use extra spaces to only add megred intervals and remain original intervals into it**

**Two intervals [x, y] & [p, q] are said to be disjoint if they do not have any point in common.**

**Wrong Solution**

class SummaryRanges {

List<int[]> intervals;

public SummaryRanges() {

intervals = new ArrayList<>();

}

public void addNum(int value) {

int[] newInterval = new int[]{value, value};

int len = intervals.size();

int i = 0;

while(i < len && intervals.get(i)[1] + 1 < newInterval[0]) {

i++;

}

// TODO: Merge interval here

while(i < len && newInterval[1] + 1 >= intervals.get(i)[0]) {

//while(i < len && newInterval[0] <= intervals.get(i)[1] + 1) {

//while(i < len && intervals.get(i)[0] + 1 >= newInterval[1]) {

newInterval[0] = Math.min(newInterval[0], intervals.get(i)[0]);

newInterval[1] = Math.max(newInterval[1], intervals.get(i)[1]);

// Remove original interval since already merge into newInterval

// and since remove one element, all elements with their indexes

// shift one position left, if we only increase index 'i' for

// next iteration, it will skip the original next element suppose

// to compare, so we have to decrease index by one right after

// the removal action, but before the increase of index which

// preparing for next iteration, so combine is index 'i' no change

intervals.remove(i);

i--;

i++;

if(i == len - 1) {

break;

}

}

//intervals.add(newInterval);

// Based on potential removal of interval happening, instead of directly

// add the new interval to the tail of list, we have to insert at the

// position very left ahead of current element

intervals.add(i, newInterval);

}

public int[][] getIntervals() {

return intervals.toArray(new int[intervals.size()][]);

//return intervals.toArray(new int[0][]);

}

}

/\*\*

\* Your SummaryRanges object will be instantiated and called as such:

\* SummaryRanges obj = new SummaryRanges();

\* obj.addNum(value);

\* int[][] param\_2 = obj.getIntervals();

\*/

**Test out by**

Input:

["SummaryRanges","addNum","getIntervals","addNum","getIntervals","addNum","getIntervals","addNum","getIntervals","addNum","getIntervals","addNum","getIntervals","addNum","getIntervals","addNum","getIntervals","addNum","getIntervals","addNum","getIntervals"]

[[],[6],[],[6],[],[0],[],[4],[],[8],[],[7],[],[6],[],[4],[],[7],[],[5],[]]

Expect:

[null,null,[[6,6]],null,[[6,6]],null,[[0,0],[6,6]],null,[[0,0],[4,4],[6,6]],null,[[0,0],[4,4],[6,6],[8,8]],null,[[0,0],[4,4],[6,8]],null,[[0,0],[4,4],[6,8]],null,[[0,0],[4,4],[6,8]],null,[[0,0],[4,4],[6,8]],null,[[0,0],[4,8]]]

Output:

java.lang.IndexOutOfBoundsException: Index 2 out of bounds for length 2

==============================================================================

"SummaryRanges",

"addNum", 6

"getIntervals", [[6,6]]

"addNum", 6

"getIntervals", [[6,6]]

"addNum", 0

"getIntervals", [[0,0],[6,6]]

"addNum", 4

"getIntervals", [[0,0],[4,4],[6,6]]

"addNum", 8

"getIntervals", [[0,0],[4,4],[6,6],[8,8]]

"addNum", 7

"getIntervals", [[0,0],[4,4],[6,8]]

"addNum", 6

"getIntervals", [[0,0],[4,4],[6,8]]

"addNum", 4

"getIntervals", [[0,0],[4,4],[6,8]]

"addNum", 7

"getIntervals", [[0,0],[4,4],[6,8]]

"addNum", 5

"getIntervals" [[0,0],[4,8]]]

**Correct Solution**

class SummaryRanges {

    List<int[]> intervals;

    public SummaryRanges() {

        intervals = new ArrayList<>();

    }

    public void addNum(int value) {

        int[] newInterval = new int[]{value, value};

        int len = intervals.size();

        int i = 0;

// No overlap

        while(i < len && intervals.get(i)[1] + 1 < newInterval[0]) {

            i++;

        }

        // Overlap happen, Insert or Merge interval

        int removeCount = 0;

        while(i < len && newInterval[1] + 1 >= intervals.get(i)[0]) {

            newInterval[0] = Math.min(newInterval[0], intervals.get(i)[0]);

            newInterval[1] = Math.max(newInterval[1], intervals.get(i)[1]);

            // Remove original interval since already merge into newInterval

            // and since remove one element, all elements with their indexes

            // shift one position left, if we only increase index 'i' for

            // next iteration, it will skip the original next element suppose

            // to compare, so we have to decrease index by one right after

            // the removal action, but before the increase of index which

            // preparing for next iteration, so combine is index 'i' no change

            intervals.remove(i);

            // Record how many times we remove elements, used for terminate

            // while loop, if not terminate will encounter 'intervals.get(i)'

            // IndexOutOfBoundsException in next iteration

            removeCount++;

// Even consolidate result of index 'i' is no change, but still list

// out to elaborate why no change

// Since remove element happen, current index 'i' rollback one step

            i--;

// Increase one for next iteration

            i++;

            // If always 'len - 1' is wrong, if remove few times, the shift

            // of index will be 'len - removeCount'

            //if(i == len - 1) {

            if(i == len - removeCount) {

                break;

            }

        }

        // Based on potential removal of interval happening, instead of directly

        // add the new interval to the tail of list, we have to insert at the

        // position very left ahead of current element

        //intervals.add(newInterval);

        intervals.add(i, newInterval);

    }

    public int[][] getIntervals() {

        return intervals.toArray(new int[intervals.size()][]);

        //return intervals.toArray(new int[0][]);

    }

}

/\*\*

 \* Your SummaryRanges object will be instantiated and called as such:

 \* SummaryRanges obj = new SummaryRanges();

 \* obj.addNum(value);

 \* int[][] param\_2 = obj.getIntervals();

 \*/

Time Complexity: O(N)

Space Complexity: O(N)

**Test Program**

import java.util.\*;

public class Solution {

List<int[]> intervals;

public Solution() {

intervals = new ArrayList<>();

}

public void addNum(int value) {

int[] newInterval = new int[]{value, value};

int len = intervals.size();

int i = 0;

while(i < len && intervals.get(i)[1] + 1 < newInterval[0]) {

i++;

}

// Insert or Merge interval

int removeCount = 0;

while(i < len && newInterval[1] + 1 >= intervals.get(i)[0]) {

newInterval[0] = Math.min(newInterval[0], intervals.get(i)[0]);

newInterval[1] = Math.max(newInterval[1], intervals.get(i)[1]);

// Remove original interval since already merge into newInterval

// and since remove one element, all elements with their indexes

// shift one position left, if we only increase index 'i' for

// next iteration, it will skip the original next element suppose

// to compare, so we have to decrease index by one right after

// the removal action, but before the increase of index which

// preparing for next iteration, so combine is index 'i' no change

intervals.remove(i);

// Record how many times we remove elements, used for terminate

// while loop, if not terminate will encounter 'intervals.get(i)'

// IndexOutOfBoundsException in next iteration

removeCount++;

// Even consolidate result of index 'i' is no change, but still list

// out to elaborate why no change

// Since remove element happen, current index 'i' rollback one step

            i--;

// Increase one for next iteration

            i++;

// If always 'len - 1' is wrong, if remove few times, the shift

// of index will be 'len - removeCount'

// Test out by:

/\*\*

\* "SummaryRanges",

\* "addNum", 6

\* "getIntervals", [[6,6]]

\* "addNum", 6

\* "getIntervals", [[6,6]]

\* "addNum", 0

\* "getIntervals", [[0,0],[6,6]]

\* "addNum", 4

\* "getIntervals", [[0,0],[4,4],[6,6]]

\* "addNum", 8

\* "getIntervals", [[0,0],[4,4],[6,6],[8,8]]

\* "addNum", 7

\* "getIntervals", [[0,0],[4,4],[6,8]] --> we will observe IndexOutOfBoundsException here

\* error out on 'if(i == len - 1) {break}', reason is we may remove multiple times cause

\* index shift multiple times, not always one time

\* "addNum", 6

\* "getIntervals", [[0,0],[4,4],[6,8]]

\* "addNum", 4

\* "getIntervals", [[0,0],[4,4],[6,8]]

\* "addNum", 7

\* "getIntervals", [[0,0],[4,4],[6,8]]

\* "addNum", 5

\* "getIntervals" [[0,0],[4,8]]]

\*/

//if(i == len - 1) {

if(i == len - removeCount) {

break;

}

}

// Based on potential removal of interval happening, instead of directly

// add the new interval to the tail of list, we have to insert at the

// position very left ahead of current element

//intervals.add(newInterval);

intervals.add(i, newInterval);

}

public int[][] getIntervals() {

return intervals.toArray(new int[intervals.size()][]);

//return intervals.toArray(new int[0][]);

}

public static void main(String[] args) {

/\* Solution so = new Solution();

so.addNum(1); // arr = [1]

int[][] a = so.getIntervals(); // return [[1, 1]]

System.out.println(a);

so.addNum(3); // arr = [1, 3]

int[][] b = so.getIntervals(); // return [[1, 1], [3, 3]]

System.out.println(b);

so.addNum(7); // arr = [1, 3, 7]

int[][] c = so.getIntervals(); // return [[1, 1], [3, 3], [7, 7]]

System.out.println(c);

so.addNum(2); // arr = [1, 2, 3, 7]

int[][] d = so.getIntervals(); // return [[1, 3], [7, 7]]

System.out.println(d);

so.addNum(6); // arr = [1, 2, 3, 6, 7]

int[][] e = so.getIntervals(); // return [[1, 3], [6, 7]]

System.out.println(e);\*/

Solution so = new Solution();

so.addNum(6); // arr = [6]

int[][] a = so.getIntervals(); // return [[6,6]]

System.out.println(a);

so.addNum(6); // arr = [6,6]

int[][] b = so.getIntervals(); // return [[6,6]]

System.out.println(b);

so.addNum(0); // arr = [6,6,0]

int[][] c = so.getIntervals(); // return [[0,0],[6,6]]

System.out.println(c);

so.addNum(4); // arr = [6,6,0,4]

int[][] d = so.getIntervals(); // return [[0,0],[4,4],[6,6]]

System.out.println(d);

so.addNum(8); // arr = [6,6,0,4,8]

int[][] e = so.getIntervals(); // return [[0,0],[4,4],[6,6],[8,8]]

System.out.println(e);

so.addNum(7); // arr = [6,6,0,4,8,7]

int[][] f = so.getIntervals(); // return [[0,0],[4,4],[6,8]]

System.out.println(f);

so.addNum(6); // arr = [6,6,0,4,8,7,6]

int[][] g = so.getIntervals(); // return [[0,0],[4,4],[6,8]]

System.out.println(g);

so.addNum(4); // arr = [6,6,0,4,8,7,6,4]

int[][] h = so.getIntervals(); // return [[0,0],[4,4],[6,8]]

System.out.println(h);

so.addNum(7); // arr = [6,6,0,4,8,7,6,4,7]

int[][] i = so.getIntervals(); // return [[0,0],[4,4],[6,8]]

System.out.println(i);

so.addNum(5); // arr = [6,6,0,4,8,7,6,4,7,5]

int[][] j = so.getIntervals(); // return [[0,0],[4,8]]]

System.out.println(j);

}

}

**Solution 2: TreeMap + Binary Search (60min)**

**使用TreeMap的最大好处就是在merge的时候不需要while loop，而是直接通过get完成了，单论merge这个动作本身时间从O(N)下降到O(1)**

class SummaryRanges {

    TreeMap<Integer, int[]> intervals;

    public SummaryRanges() {

        intervals = new TreeMap<>();

    }

    public void addNum(int value) {

        // Find the interval immediately before the new value

        Integer leftIndex = intervals.floorKey(value);

        // Find the interval immediately after the new value

        Integer rightIndex = intervals.ceilingKey(value);

        // Check if there's a need to merge the intervals to the left and right of the new value

            // Merge both intervals since the new value bridges them

            intervals.get(leftIndex)[1] = intervals.get(rightIndex)[1];

            // Remove the redundant interval (the one that was to the right)

            intervals.remove(rightIndex);

        // Note: Condition not '==' but '>='

        // e.g if current interval is [1,2] or [1,4], given new interval [2,2]

        // both need to merge [2,2] into it, and for 'intervals.get(leftIndex)[1]'

        // the value as '2' or '4' both '>=' val as 2

        // Extend the interval to the left if the new value fits within or adjacent to it

        } else if(leftIndex != null && intervals.get(leftIndex)[1] + 1 >= value) {

            intervals.get(leftIndex)[1] = Math.max(intervals.get(leftIndex)[1], value);

        // Extend the interval to the right if the new value fits within or adjacent to it

        } else if(rightIndex != null && intervals.get(rightIndex)[0] - 1 <= value) {

            intervals.get(rightIndex)[0] = Math.min(intervals.get(rightIndex)[0], value);

        // If the value is not adjacent to any interval, add it as a new interval

        } else {

            intervals.put(value, new int[]{value, value});

        }

    }

    public int[][] getIntervals() {

        int size = intervals.size();

        int[][] result = new int[size][2];

        int i = 0;

        for(int[] interval : intervals.values()) {

            result[i++] = interval;

        }

        return result;

    }

}

/\*\*

 \* Your SummaryRanges object will be instantiated and called as such:

 \* SummaryRanges obj = new SummaryRanges();

 \* obj.addNum(value);

 \* int[][] param\_2 = obj.getIntervals();

 \*/

Time Complexity: addNum O(logN), getIntervals O(N)

Space Complexity: O(N)

**Q1: Why we can update TreeMap value by 'get' method ?**

**Answer by chatGPT:**

Usually to update the map value we have to use 'put' method, but from above code, we find it directly update the value with 'get' method such as "intervalsMap.get(left)[1] = intervalsMap.get(right)[1];", can you explain how it works ?

Answer from chatGPT:

The key to understanding this behavior lies in how Java handles array objects. In Java, arrays are reference types, and when you retrieve an array from a collection (such as a **TreeMap** in this case), you obtain a reference to the same array object stored in the collection. Therefore, any modifications made to the array through this reference will affect the actual array stored in the collection.

In the code snippet:

intervalsMap.get(left)[1] = intervalsMap.get(right)[1];

Here's the breakdown:

intervalsMap.get(left): Retrieves the array associated with the key left from the TreeMap.

[1]: Accesses the second element of the retrieved array (since arrays in Java are 0-indexed).

= intervalsMap.get(right)[1]: Assigns the value of the second element of the array associated with the key right to the second element of the array associated with the key left.

This line effectively updates the end value of the interval associated with the key **left** with the end value of the interval associated with the key **right**. Since the retrieved array is a reference to the actual array stored in the **TreeMap**, the modification directly affects the interval stored in the collection.

It's important to note that this approach works because the arrays stored in the **TreeMap** are mutable, and the modifications are made to the same array object. This technique can be convenient but requires caution to ensure that you don't unintentionally modify shared state.

**Q2: Why we cannot run customized Binary Search to find floor or ceiling key against TreeMap key set, and have to comply with 'floorKey()' or 'ceilingKey()' method ?**

**Q2-1: If we have a defined TreeMap as "TreeMap<Integer, int[]> map = new TreeMap<>();", can we run binary search against the set get by "map.keySet();" ?**

**Answer by chatGPT:**

A **TreeMap** in Java already maintains its keys in sorted order, and you can use the **floorKey** method directly on the **TreeMap** without needing a separate binary search. The **keySet** of a **TreeMap** is a **NavigableSet**, which supports methods like **floor**, **ceiling**, **lower**, and **higher** for efficient retrieval of elements based on their order.

Here's an example:

import java.util.TreeMap;

import java.util.NavigableSet;

public class TreeMapExample {

public static void main(String[] args) {

TreeMap<Integer, int[]> map = new TreeMap<>();

map.put(3, new int[]{30});

map.put(6, new int[]{60});

map.put(9, new int[]{90});

NavigableSet<Integer> keySet = map.keySet();

int targetKey = 7;

Integer floorKey = keySet.floor(targetKey);

if (floorKey != null) {

int[] value = map.get(floorKey);

System.out.println("Floor key: " + floorKey + ", Value: " + value[0]);

} else {

System.out.println("No floor key found for " + targetKey);

}

}

}

In this example, **keySet.floor(targetKey)** is used to find the floor key for the given target key directly from the **NavigableSet** obtained from **map.keySet()**. This approach is more straightforward than using a separate binary search when working with a **TreeMap**.

**Q2-2: Is "NavigableSet" implementation includes indexes or not ?**

**Answer by chatGPT:**

**NavigableSet** in Java, specifically its most commonly used implementation **TreeSet**, does not directly expose indexes like a list-based structure. Unlike **List** implementations (e.g., **ArrayList** or **LinkedList**), which allow random access through indexes, **NavigableSet** provides methods that operate based on the natural ordering of elements.

While **NavigableSet** provides methods like **floor**, **ceiling**, **lower**, and **higher** for efficient retrieval of elements based on their order, it does not have an index-based API.

If you need to access elements by index or perform operations related to indices, you may want to consider using a **List** implementation or other data structures that support direct access by index. Keep in mind that the choice of data structure depends on the specific requirements of your use case.

**Refer to**

<https://algo.monster/liteproblems/352>

**Problem Description**

This LeetCode problem involves writing a class called SummaryRanges that is designed to keep track of a stream of non-negative integers and provide a summarized view of the data in the form of disjoint intervals. The class should support two operations:

addNum(value): This method takes a non-negative integer value and adds it to the stream.

getIntervals(): This method returns the current summary of the stream as a list of disjoint intervals [start\_i, end\_i]. Each interval represents a sequence of consecutive numbers that have been added to the stream. The list of intervals is sorted based on the start\_i of each interval.

The goal of SummaryRanges is to condense the stream of integers into as few intervals as possible by merging consecutive numbers together.

**Intuition**

To solve this problem, one needs to efficiently manage intervals of consecutive numbers with operations to insert a number into the stream and then update and merge these intervals. The key challenge is handling numbers that might need to merge with existing intervals when they fit just right before the start, after the end, or in between two existing intervals.

To address this efficiently, we can use a data structure that maintains the intervals in a sorted order to easily locate where the new number fits relative to existing intervals. This would simplify operations for finding, adding, or merging intervals. A SortedDict, which is a dictionary that keeps its keys sorted, can be used for this purpose. Each key is an integer from the data stream, and the corresponding value is an interval [start, end] representing consecutive numbers.

Here is the intuition behind handling the addition of a new number, val, to the stream:

Check if val can merge with an existing interval on its left or right or if it fills a gap between two intervals:

If val is just after the end of one interval and just before the start of the next, we merge these two intervals.

If val is just after the end of an existing interval but not close enough to the next interval, we extend the existing interval to include val.

Similarly, if val is just before the start of an existing interval but not close enough to the previous interval, we adjust the start of this interval to include val.

If val does not fit next to or between existing intervals, we create a new interval [val, val].

For getting the intervals, since we maintain a sorted dictionary where each interval is stored as values and the keys are the starting points of each interval, we simply return these values.

The provided solution uses this approach with a SortedDict to ensure that we can quickly locate adjacent intervals and merge intervals or adjust their boundaries as necessary when a new number is added.

**Solution Approach**

The implementation of the SummaryRanges class utilizes a SortedDict from the sortedcontainers module in Python. A SortedDict keeps keys in a sorted order, which is vital for efficient interval management in this problem.

**\_\_init\_\_ Method:**

The constructor initializes a SortedDict named self.mp. This will store the intervals where the keys will be the start of the intervals.

**addNum Method:**

This method handles the addition of a new number, val, to the stream and updates the intervals accordingly:

We start by determining the position (right index ridx) of the value with respect to the sorted keys using self.mp.bisect\_right(val). This gives us the index of the first key that is greater than val (if any). Similarly, we calculate the left index (lidx) which is just ridx - 1 if val is not the first number to be inserted or n (the length of the SortedDict) if ridx is 0.

With keys and values of the SortedDict extracted, we use the indices to find the adjacent intervals (if they exist).

We check if the new value val can merge with adjacent intervals or if it belongs to a new, separate interval:

If the new value val is in between two existing intervals and can merge them (values[lidx][1] + 1 == val and values[ridx][0] - 1 == val), we extend the left interval to cover the right one and remove the right interval.

If the new value val can be appended to the end of the left interval (val <= values[lidx][1] + 1), we update the end of this interval.

If the new value val can be prepended to the start of the right interval (val >= values[ridx][0] - 1), we update the start of this interval.

If none of the above conditions are met, val creates its own interval [val, val].

We then insert the new interval or update the existing intervals in the SortedDict accordingly.

**getIntervals Method:**

This is a straightforward method that returns all values (which are the intervals) in the SortedDict. Since the SortedDict maintains the intervals in ascending order based on their starting points, no further sorting is needed.

Overall, the algorithm effectively maintains and manipulates intervals, using the fast lookup and ordered nature of the SortedDict. The choice of data structure greatly simplifies the complexity of adding and merging intervals while keeping the stream data summarized efficiently.

**Example Walkthrough**

Let's walk through a small example to illustrate the solution approach.

Assume we have an instance of SummaryRanges and we perform the following operations:

addNum(1)

addNum(3)

getIntervals() - Should return [[1, 1], [3, 3]]

addNum(2)

getIntervals() - Should return [[1, 3]]

Here's what happens internally with each operation:

After operation 1: addNum(1)

There are no existing intervals.

val = 1 creates a new interval [1, 1].

self.mp now stores {1: [1, 1]}.

After operation 2: addNum(3)

Keys in self.mp are [1].

val = 3 does not fit with the existing interval [1, 1] as it is not consecutive.

A new interval [3, 3] is created since it is separate.

self.mp now stores {1: [1, 1], 3: [3, 3]}.

After operation 3: getIntervals()

The getIntervals method is called.

We simply return the values of self.mp as a list: [[1, 1], [3, 3]].

After operation 4: addNum(2)

Keys in self.mp are [1, 3].

val = 2 is checked against the intervals.

val = 2 fits between the intervals [1, 1] and [3, 3] and can merge them since 1 + 1 = 2 = 3 - 1.

We extend the left interval to [1, 3] and remove the interval [3, 3].

self.mp now stores {1: [1, 3]}.

After operation 5: getIntervals()

The getIntervals method is called once again.

We simply return the values of self.mp as a list: [[1, 3]].

In conclusion, as shown in the example, the SummaryRanges class efficiently manages the addition and merging of intervals. The addNum method updates the SortedDict as required, with the main logic focused on checking if the new value can be merged with existing intervals or if it forms a new interval. The getIntervals method provides the current summarized view as ordered intervals. The effective use of SortedDict dramatically simplifies complex interval logic and ensures efficient manipulation of the data stream.

**Java Solution**

import java.util.Map;

import java.util.TreeMap;

class SummaryRanges {

// TreeMap to hold the intervals. The key is the start of the interval,

// and the value is an array containing the start and end of the interval.

private TreeMap<Integer, int[]> intervalsMap;

// Constructor to initialize the TreeMap

public SummaryRanges() {

intervalsMap = new TreeMap<>();

}

// Function to add a number into the set and merge intervals if necessary

public void addNum(int val) {

// Find the interval immediately before the new value

Integer left = intervalsMap.floorKey(val);

// Find the interval immediately after the new value

Integer right = intervalsMap.ceilingKey(val);

// Check if there's a need to merge the intervals to the left and right of the new value

// Merge both intervals since the new value bridges them

intervalsMap.get(left)[1] = intervalsMap.get(right)[1];

// Remove the redundant interval (the one that was to the right)

intervalsMap.remove(right);

} else if (left != null && val <= intervalsMap.get(left)[1] + 1) {

// Extend the interval to the left if the new value fits within or adjacent to it

intervalsMap.get(left)[1] = Math.max(val, intervalsMap.get(left)[1]);

} else if (right != null && val >= intervalsMap.get(right)[0] - 1) {

// Extend the interval to the right if the new value fits within or adjacent to it

intervalsMap.get(right)[0] = Math.min(val, intervalsMap.get(right)[0]);

} else {

// If the value is not adjacent to any interval, add it as a new interval

intervalsMap.put(val, new int[] {val, val});

}

}

// Function to return a list of intervals

public int[][] getIntervals() {

// Initialize the result array with the size of the intervalsMap

int[][] result = new int[intervalsMap.size()][2];

// Populate the result with the intervals

int i = 0;

for (int[] range : intervalsMap.values()) {

result[i++] = range;

}

// Return the populated list of intervals

return result;

}

}

**Time and Space Complexity**

**Time Complexity**

The addNum function:

bisect\_right: O(logN) where N is the number of existing intervals in self.mp. This operation performs a binary search.

Update operations: O(1) for basic integer operations.

pop: O(logN) because when popping from a sorted dictionary, it maintains the order. Hence, the time complexity for addNum is O(logN) per operation due to the binary search and potential pop operations.

The getIntervals function simply returns the values of the sorted dictionary:

This operation is O(N) because it constructs a list from the values stored in mp.

**Space Complexity**

The main space usage in SummaryRanges comes from the sorted dictionary self.mp:

For addNum: Additional space complexity is O(1) because it stores the interval as a pair of integers only when a new interval is created or extended.

The SortedDict self.mp can have at most N elements if no intervals overlap, where N is the total number of times addNum is called. Hence, space complexity is O(N) for self.mp.

For getIntervals: O(N) to store the output list.

Therefore, the overall space complexity is O(N).

**另外的做法:**

<https://grandyang.com/leetcode/352/>

这道题说有个数据流每次提供一个数字，然后让我们组成一系列分离的区间，这道题跟之前那道 [Insert Interval](http://www.cnblogs.com/grandyang/p/4367569.html) 很像，思路也很像，每进来一个新的数字 val，都生成一个新的区间 [val, val]，并且新建一个空的区间数组 res，用一个变量 cur 来保存要在现有的区间数组中加入新区间的位置。此时遍历现有的区间数组 intervals，对于每一个遍历到的当前区间 interval，假如要加入的区间的结尾位置加1比当前区间的起始位置小，说明二者不相连，将当前区间加入 res。否则当要加入区间的起始位置大于当前位置的结束位置加1，说明二者也没有交集，可以将当前区间加入 res，不过此时 cur 要自增1，因为要加入区间的位置在当前区间的后面。再否则的话，二者就会有交集，需要合并，此时用二者起始位置中较小的更新要加入区间的起始位置，同理，用二者结束位置中较大的去更新要加入区间的结束位置。最终将要加入区间放在 res 中的 cur 位置，然后将 res 赋值给 intervals 即可，参见代码如下:

class SummaryRanges {

public:

SummaryRanges() {}

void addNum(int val) {

vector<int> newInterval{val, val};

vector<vector<int>> res;

int cur = 0;

for (auto interval : intervals) {

if (newInterval[1] + 1 < interval[0]) {

res.push\_back(interval);

} else if (newInterval[0] > interval[1] + 1) {

res.push\_back(interval);

++cur;

} else {

newInterval[0] = min(newInterval[0], interval[0]);

newInterval[1] = max(newInterval[1], interval[1]);

}

}

res.insert(res.begin() + cur, newInterval);

intervals = res;

}

vector<vector<int>> getIntervals() {

return intervals;

}

private:

vector<vector<int>> intervals;

};

// Java version (convert from chatGPT)

import java.util.ArrayList;

import java.util.List;

class SummaryRanges {

private List<int[]> intervals;

public SummaryRanges() {

intervals = new ArrayList<>();

}

public void addNum(int val) {

int[] newInterval = {val, val};

List<int[]> res = new ArrayList<>();

int cur = 0;

for (int[] interval : intervals) {

if (newInterval[1] + 1 < interval[0]) {

res.add(interval);

} else if (newInterval[0] > interval[1] + 1) {

res.add(interval);

cur++;

} else {

newInterval[0] = Math.min(newInterval[0], interval[0]);

newInterval[1] = Math.max(newInterval[1], interval[1]);

}

}

res.add(cur, newInterval);

intervals = res;

}

public List<int[]> getIntervals() {

return intervals;

}

}

我们可以对上面的解法进行优化。由于上面的方法每次添加区间的时候，都要把 res 赋值给 intervals，整个区间数组都要进行拷贝，十分的不高效。这里换一种方式，用一个变量 overlap 来记录所有跟要加入区间有重叠的区间的个数，用变量i表示新区间要加入的位置，这样只要最后 overlap 大于0了，现在 intervals 中将这些重合的区间删掉，然后再将新区间插入，这样就不用进行整体拷贝了，提高了效率，参见代码如下：

class SummaryRanges {

public:

SummaryRanges() {}

void addNum(int val) {

vector<int> newInterval{val, val};

int i = 0, overlap = 0, n = intervals.size();

for (; i < n; ++i) {

if (newInterval[1] + 1 < intervals[i][0]) break;

if (newInterval[0] <= intervals[i][1] + 1) {

newInterval[0] = min(newInterval[0], intervals[i][0]);

newInterval[1] = max(newInterval[1], intervals[i][1]);

++overlap;

}

}

if (overlap > 0) {

intervals.erase(intervals.begin() + i - overlap, intervals.begin() + i);

}

intervals.insert(intervals.begin() + i - overlap, newInterval);

}

vector<vector<int>> getIntervals() {

return intervals;

}

private:

vector<vector<int>> intervals;

};

// Java version (convert from chatGPT)

import java.util.ArrayList;

import java.util.List;

class SummaryRanges {

private List<int[]> intervals;

public SummaryRanges() {

intervals = new ArrayList<>();

}

public void addNum(int val) {

int[] newInterval = {val, val};

int i = 0, overlap = 0, n = intervals.size();

while (i < n) {

if (newInterval[1] + 1 < intervals.get(i)[0]) break;

if (newInterval[0] <= intervals.get(i)[1] + 1) {

newInterval[0] = Math.min(newInterval[0], intervals.get(i)[0]);

newInterval[1] = Math.max(newInterval[1], intervals.get(i)[1]);

overlap++;

}

i++;

}

if (overlap > 0) {

intervals.subList(i - overlap, i).clear();

}

intervals.add(i - overlap, newInterval);

}

public List<int[]> getIntervals() {

return intervals;

}

}