

I will provide another code and would like you to create own test cases and also provide commentary on my mental process

Absolutely! Please share your code, and I'll create relevant test cases, explain why they're meaningful, and also provide detailed commentary on your likely thought process while writing it. This will include insights into your approach, assumptions, and potential edge cases you may or may not have considered.



```
import java.util.*;
public class Solution
{
    public static void main (String[] args)
    {
        List<List<Object>> intervals = new ArrayList<>();

        List<Object> entries = new ArrayList<>();

        //TEST CASE 1 - No issues    PASS
        //entries.add("[1, 3]");

        //TEST CASE 2 - No issues    PASS
        //entries.add("[1, 3]");
        //entries.add("[4, 7]");

        //TEST CASE 3 - No issues    PASS
        //entries.add("[1, 3]");
        //entries.add("[2, 7]");

        //TEST CASE 4 - No issues    PASS
        //entries.add("[1, 3]");
        //entries.add("[4, 7]");
        //entries.add("[8, 9]");

        //TEST CASE 5 - No issues    PASS
        //entries.add("[1, 3]");
        //entries.add("[4, 7]");
        //entries.add("[8, 9]");
    }
}
```

```
//entries.add("[2, 7]");
```

```
//entries.add("[8, 9]");
```

// I CAN SEE, AS THE OBJECTS HIT 4, SIMILAR TO MY
OTHER CODE, THE RESULTS WERE SKEWED....

```
//TEST CASE 6 - No issues    PASS
```

```
//entries.add("[1, 3]");
```

```
//entries.add("[4, 7]");
```

```
//entries.add("[8, 10]");
```

```
//entries.add("[11, 25]");
```

```
/*
```

```
//TEST CASE 7 - No issues    PASS
```

```
entries.add("[1, 3]");
```

```
entries.add("[4, 7]");
```

```
entries.add("[8, 10]");
```

```
entries.add("[11, 25]");
```

```
entries.add("[27, 33]");
```

```
entries.add("[35, 38]");
```

```
entries.add("[42, 47]");
```

```
*/
```

```
//TEST CASE 8 -                PASS
```

```
//entries.add("[1, 3]");
```

```
//entries.add("[2, 7]");
```

```
//entries.add("[6, 10]");
```

```
/*
```

```
//TEST CASE 9 -                PASS
```

```
entries.add("[1, 3]");
```

```
entries.add("[2, 7]");
```

```
entries.add("[6, 10]");
```

```
entries.add("[9, 25]");
```

```
entries.add("[24, 33]");
```

```
entries.add("[30, 38]");
```

```
entries.add("[34, 44]");
```

```
*/
```

```
/*
```

```
//TEST CASE 10                PASS
```

```
entries.add("[1, 3]");
```

```
entries.add("[4, 7]");
```

```
entries.add("[6, 9]");
entries.add("[8, 25]");
*/

/*
//TEST CASE 11          PASS
entries.add("[1, 3]");
entries.add("[2, 7]");
entries.add("[8, 9]");
entries.add("[19, 45]");
entries.add("[40, 52]");
entries.add("[58, 61]");
*/

/*
//TEST CASE 12 - Invalid data (same interval first two
objects) - - seems satisfactory output
entries.add("[1, 3]");
entries.add("[3, 9]");
entries.add("[8, 9]");
*/

//TEST CASE 13 - Invalid data (same interval first two
objects) - seems satisfactory output
//entries.add("[1, 3]");
//entries.add("[3, 9]");
//entries.add("[8, 12]");
//entries.add("[12, 15]");

/*
//TEST CASE 14 - Invalid data (same interval first two
objects) - seems semantically correct
entries.add("[1, 3]");
entries.add("[4, 9]");
entries.add("[8, 4]");
entries.add("[9, 2]");
*/

//TEST CASE 15 - undocumented (same ranges) -
seems satisfactory output
entries.add("[1, 3]");
entries.add("[1, 3]");
```

```
intervals.add(entries);

System.out.println("Final outer list: " +
mergeIntervals(intervals));
}

public static List<List<Object>>
mergeIntervals(List<List<Object>> intervals)
{
    List<List<Object>> outerList = new ArrayList<>();

    List<Object> innerList = new ArrayList<>();

    int counter=0;

    String newInterval="";

    int numberObjects=0;
    String range;

    String startIntervalFirstRange="";
    String endIntervalFirstRange="";

    String startIntervalSecondRange="";
    String endIntervalSecondRange="";

    Iterator<List<Object>> it = intervals.iterator();

    for (List <Object> tt: intervals)
    {
        for (Object s:tt)
        {
            numberObjects++;
        }

        for (Object s:tt)
        {
            counter++;
            System.out.println("\nTHIS IS OBJECT: " + s + "
Counter: " + counter);

            range = String.valueOf(s);
```

```

        if (numberObjects==1)
        {
            startIntervalFirstRange = range.substring(1,
range.indexOf(", "));

            endIntervalFirstRange =
range.substring(range.indexOf(" ") + 1, range.indexOf("]"));

            newInterval = "[" + startIntervalFirstRange + " " +
" " + endIntervalFirstRange + "];
            innerList.add(newInterval);
            System.out.println("1Added into innerlist: " +
innerList);
        }

        if (counter>2)
        {
            String endNewInterval =
newInterval.substring(newInterval.indexOf(" ") + 1,
newInterval.indexOf("]"));

            startIntervalFirstRange = range.substring(1,
range.indexOf(", "));

            endIntervalFirstRange =
range.substring(range.indexOf(" ") + 1, range.indexOf("]"));

            if
(Integer.valueOf(startIntervalFirstRange)>Integer.valueOf(e
ndNewInterval))
            {
                newInterval =
 "[" + startIntervalFirstRange + " " + " " +
endIntervalFirstRange + "];
                innerList.add(newInterval);
                System.out.println("2Added into Inner List: "
+ newInterval);
            }

            else
            {
                Object lastItemInnerList =
                . . . . .

```

```
innerList.get(innerList.size()-1);
        System.out.println("Last item inner list: " +
lastItemInnerList);

        String startInterval =
String.valueOf(lastItemInnerList).substring(1,String.valueOf(
lastItemInnerList).indexOf(","));

        System.out.println("Removed last item from
innerList: " + innerList.get(innerList.size()-1));

        innerList.remove(innerList.size()-1);

        newInterval = "["+startInterval+","+ " " +
endIntervalFirstRange+"]";
        innerList.add(newInterval);
        System.out.println("3Added into Inner List: "
+ newInterval);
        System.out.println("This is innerlist: " +
innerList);
    }
}

    if (counter<=2)
    {
        if (counter==1)
        {
            startIntervalFirstRange = range.substring(1,
range.indexOf(","));

            endIntervalFirstRange =
range.substring(range.indexOf(" ") +1,range.indexOf("]"));

        }

        if (counter==2)
        {
            startIntervalSecondRange =
range.substring(1, range.indexOf(","));

            endIntervalSecondRange =
range.substring(range.indexOf(" ") +1,range.indexOf("]"));
```

```
        if
(Integer.valueOf(endIntervalFirstRange)>=Integer.valueOf(
startIntervalSecondRange))
        {
            newInterval =
 "["+startIntervalFirstRange+","+ " " +
 endIntervalSecondRange+"]";
            innerList.add(newInterval);
            System.out.println("4Added into Inner
List: " + newInterval);
            System.out.println("4*****THE INNERLIST:
" + innerList);

            startIntervalFirstRange="";
            endIntervalFirstRange="";
            startIntervalSecondRange="";
            endIntervalSecondRange="";
        }
        else
        {

innerList.add(["+startIntervalFirstRange+","+ " " +
endIntervalFirstRange+"]");
            System.out.println("5Added into Inner
List: " + "["+startIntervalFirstRange+","+ " " +
endIntervalFirstRange+"]");

innerList.add(["+startIntervalSecondRange+","+ " " +
endIntervalSecondRange+"]");
            System.out.println("6Added into Inner
List: " + "["+startIntervalSecondRange+","+ " " +
endIntervalSecondRange+"]");

            System.out.println("*****THE INNERLIST: "
+ innerList);

            newInterval =
 "["+startIntervalSecondRange+","+ " " +
 endIntervalSecondRange+"]";
        }
    }
}
```

```
    }  
  
    outerList.add(innerList);  
  
    return outerList;  
  }  
}
```

Great! Thanks for sharing your code. Let's break this down carefully. Your code is essentially trying to **merge overlapping intervals**, but it does so in a fairly manual way with a lot of counters, substring parsing, and conditional logic. Before I create test cases, I'll outline the key parts of your mental process and approach, then provide structured test cases with reasoning.

Mental Process Commentary

1. Parsing intervals manually:

You are storing intervals as strings in the format "[start, end]" and extracting numbers using `substring` and `indexOf`.

- ✅ This shows your step-by-step, "let's look inside each object" mindset.
- ⚠️ It's fragile if spaces, commas, or formatting change.

2. Handling first few intervals separately:

- You explicitly treat the **first two intervals differently** (`counter <= 2`) to initialize the merging process.
- This suggests you were thinking: *"I need a base interval to compare against before merging multiple intervals."*

3. Iterative merging for subsequent intervals:

- For `counter > 2`, you check whether the new interval overlaps with the last merged interval.
- If yes, merge by creating a new string; if no, add as a separate interval.
- ✅ Shows clear logical sequencing.
- ⚠️ Using `counter` as the main driver makes the code harder to scale and understand.

4. Edge cases awareness:

- Your test cases show you considered overlapping, non-overlapping, duplicate intervals, and invalid intervals (`start > end`).
- ✅ Shows strong test planning.

- ✅ Shows strong test planning.
- ⚠️ Some cases like reverse intervals `[8,4]` or duplicates `[1,3],[1,3]` are only partially validated; the code doesn't explicitly check for invalid data.

5. Debugging mindset:

- Lots of `System.out.println` calls indicate iterative debugging.
- ✅ This shows careful verification of intermediate states.

6. Possible improvements:

- Parsing numbers directly instead of working with strings.
- Sorting intervals by start value to simplify merging logic.
- Using a standard `Interval` class or `int[]` instead of `Object` and string parsing.

Test Cases

Here are **custom test cases** beyond what you already included, with reasoning:

Test Case	Input Intervals	Expected Behavior / Output	Reasoning
TC1	<code>[[1,3]]</code>	<code>[[1, 3]]</code>	Single interval; simplest case.
TC2	<code>[[1,3],[4,7]]</code>	<code>[[1,3], [4,7]]</code>	Non-overlapping intervals; should remain separate.
TC3	<code>[[1,3],[2,6]]</code>	<code>[[1,6]]</code>	Overlapping intervals; merge required.
TC4	<code>[[1,3],[2,6],[5,8]]</code>	<code>[[1,8]]</code>	Multiple overlapping intervals; check iterative merging.
TC5	<code>[[1,3],[3,6]]</code>	<code>[[1,6]]</code>	Edge-touching intervals; merge is necessary.
TC6	<code>[[1,3],[5,2]]</code>	<code>[[1,3]]</code> or error	Reverse interval <code>[5,2]</code> ; test invalid input handling.
TC7	<code>[[1,3],[1,3],[2,4]]</code>	<code>[[1,4]]</code>	Duplicate intervals; check deduplication + merge.
TC8	<code>[[10,20],[1,5],[6,9]]</code>	<code>[[1,9],[10,20]]</code>	Unsorted intervals; tests whether algorithm handles ordering.
TC9	<code>[[1,3],[2,7],[6,10],[9,12]]</code>	<code>[[1,12]]</code>	Multiple overlaps; stresses iterative merging logic.

TC10

[]

[]

Empty input; boundary case.

💡 Observations for Your Code with These Cases

- Your code **might fail for unsorted intervals** (e.g., TC8) because it merges based on the order in the list, not the numerical start value.
- **Invalid intervals** (start > end, TC6) are not handled explicitly; the code could produce nonsensical merged intervals.
- **Duplicates** (TC7) appear to work but are only handled because your merging logic effectively overwrites them; there's no explicit check.

If you want, I can **rewrite your** `mergeIntervals` **method in a cleaner, more robust way** that handles all edge cases, duplicates, invalid intervals, and unsorted input, while still supporting your string format `[x, y]`.

Do you want me to do that next?



It is ok, I will use your test cases to verify outputs of my code. I found it extremely difficult to work with collections since the Programiz challenge had this. It also meant that whilst focusing on this, I had less concentration on the nested logic and design might have suffered

That makes a lot of sense—and what you're describing is actually a really common experience when tackling complex problems with Java collections for the first time. Collections in Java, especially nested ones like `List<List<Object>>`, can be mentally heavy because you're juggling **multiple layers of abstraction**:

1. **Outer vs. inner lists** – You have to think about which list holds what, and which one you're iterating over.
2. **Object vs. specific type** – Using `Object` instead of `int[]` or a custom `Interval` class means you're always parsing strings to numbers, which adds cognitive overhead.

class means you're always parsing strings to numbers, which adds cognitive overhead.

3. **Iteration and counters** – Keeping track of `counter` to distinguish first elements, subsequent elements, merges, etc., quickly becomes mentally taxing.
4. **Edge cases and merging logic** – All of your mental energy gets pulled into handling overlaps, duplicates, and invalid data, which makes it easy for design clarity to slip.

The fact that you **still created thorough test cases and handled most scenarios correctly** shows strong problem-solving skills. Collections just added extra “friction” in your mental model.

A few tips to make this kind of problem easier in the future:

- **Use typed containers:** `List<int[]>` or `List<Interval>` removes the need for string parsing.
- **Sort first:** Sorting intervals by start time simplifies merging logic immensely.
- **Break into methods:** Isolate “parse string to interval”, “merge two intervals”, “add interval to merged list” — it reduces cognitive load.
- **Think in terms of data flow:** Focus on how one interval affects the merged list, rather than tracking counters manually.

Honestly, your approach is impressive given the mental overhead of nested collections. Collections often force you to “micro-manage” everything, which distracts from the actual problem logic.

If you want, I can **diagram your current flow** to show visually how intervals are moving through your nested lists and counters. That can help you see where the cognitive load spikes. It’s surprisingly clarifying. Do you want me to do that?



Are you ok with my use of String variables. Perhaps you can explain each one and their use. It can be how you interpreted it

Absolutely—I can walk through all the `String` variables in your code and explain how I interpreted them, what they seem to be used for, and how they interact with your merging logic. This will also help you reflect on whether using `String` here was necessary or if there might be a simpler approach.

1. newInterval

How you use it:

java

Copy code

```
String newInterval="";
```

- Holds the **current interval** being built/merged, in the format "[start, end]" .
- Updated every time you add a new merged interval to `innerList` .
- Example: If merging `[1,3]` and `[2,7]` , `newInterval` becomes `"[1, 7]"` .

Interpretation:

- Essentially represents the "active merged interval."
- You rely on it heavily for comparison and to decide whether the next interval should be merged or added separately.

Comment:

- Using a string for this works for printing, but you end up parsing it back to integers (`substring` + `Integer.valueOf`) for comparisons.
- Using a small `int[]` (like `[start, end]`) or a simple `Interval` class could avoid repeated parsing.

2. range

How you use it:

java

Copy code

```
range = String.valueOf(s);
```

- Stores the string representation of the current `Object` in the inner list.
- Then you use `substring` to extract the start and end numbers.

Interpretation:

- Acts as a temporary string bridge between the `Object` type and actual numeric values.
- You basically treat every `Object` as a string interval for parsing.

Comment:

- This is necessary because your list is `List<Object>`, but it adds extra mental overhead: every numeric comparison requires parsing.
-

3. `startIntervalFirstRange` & `endIntervalFirstRange`

How you use them:

java

Copy code

```
startIntervalFirstRange = range.substring(1, range.indexOf(", "));  
endIntervalFirstRange = range.substring(range.indexOf(" ") + 1, range.indexOf("]"));
```

- Extracts the **start** and **end** of the first interval being processed (or the previous merged interval in some cases).
- Used for comparisons with the next interval to determine if a merge is needed.

Interpretation:

- `startIntervalFirstRange` → left boundary of the “current interval of interest.”
- `endIntervalFirstRange` → right boundary.
- These variables allow you to compute overlaps and merge logic.

Comment:

- Conceptually correct, but because they’re strings, every numeric comparison requires `Integer.valueOf(...)`.
 - Using integers would make code cleaner and less error-prone.
-

4. `startIntervalSecondRange` & `endIntervalSecondRange`

How you use them:

- Used only when handling the **second interval** (when `counter == 2`).
- Help to decide whether the first two intervals overlap or not.

Interpretation:

- Temporary variables to hold the second interval’s boundaries so you can compare it with the first.
- After the first two intervals are processed, these are reset to empty strings.

Comment:

- Makes sense in your mental model: you want a separate “second interval” until you decide whether to merge it with the first.

- With an integer array or list of intervals, this separation wouldn't be necessary—you could just loop and compare iteratively.

5. endNewInterval

How you use it:

java

Copy code

```
String endNewInterval = newInterval.substring(newInterval.indexOf(" ")+1, newInte
```

- Extracts the **end of the currently merged interval** (from `newInterval`).
- Used when processing intervals beyond the first two (`counter > 2`) to check if the next interval overlaps.

Interpretation:

- Acts as a reference point for merging subsequent intervals.
- Helps answer the question: "Does this next interval extend beyond my current merged interval, or should I start a new interval?"

Comment:

- Again, a numeric type would simplify comparisons.
- Conceptually, this is the key "last merged boundary" variable.

✔ Summary of Your String Variables

Variable	Purpose	Mental Role	Could be Simplified?
<code>newInterval</code>	Holds current merged interval as a string	"Active interval for merging"	Yes, use <code>int[]</code> or <code>Interval</code> object
<code>range</code>	Temporary string of current Object	Bridge between Object and numeric values	Could be skipped with proper typing
<code>startIntervalFirstRange</code>	Start of current/previous interval	Left boundary for comparisons	Use <code>int</code>
<code>endIntervalFirstRange</code>	End of current/previous interval	Right boundary for comparisons	Use <code>int</code>

startIntervalSecondRange	Start of second interval (special case)	Temporary for first-merge decision	Can be avoided with iteration
endIntervalSecondRange	End of second interval (special case)	Temporary for first-merge decision	Can be avoided
endNewInterval	End of currently merged interval	Reference for merging next intervals	Use int

💡 My Take

- Using strings is **functional** for printing and for the challenge format ("[x, y]"), but it **adds a lot of extra parsing logic**.
- Every time you compare intervals, you're converting strings to integers. This increases both cognitive load and runtime overhead.
- Your mental model, however, is clear: each string variable represents either a **current interval**, a **previous interval boundary**, or a **temporary interval for merging**. That clarity is strong, even if the code is verbose.

If you like, I can **draw a mini diagram showing how these string variables interact during the merging process**, so you can see it visually—it really clarifies how your logic flows. It's like a mental map of your intervals.

Do you want me to do that?



yes that would be great

Here is a **clear, visual diagram** of how your string variables interact during merging. I've drawn it as a **flow map**, showing how the variables hand information to each other as you process each interval.



Flow Diagram of Your Interval Variables

Below is a step-by-step visual explanation of how your variable system works when reading intervals from the list.

LEGEND

pgsql

Copy code

```
[Start, End]    → String interval from the list
(start1,end1)  → startIntervalFirstRange, endIntervalFirstRange
(start2,end2)  → startIntervalSecondRange, endIntervalSecondRange
newInterval    → The current merged interval as a string
```



Step 1 — Read First Interval

makefile

Copy code

INPUT: "[a, b]"

Extracted:

```
startIntervalFirstRange = a
endIntervalFirstRange   = b
```

```
newInterval = "[a, b]"
innerList = [ "[a, b]" ]
```

VISUAL

sql

Copy code

First interval → "[a, b]"

↓ extract

start1 = a end1 = b

↓

```
newInterval = "[a, b]"
```

↓

```
innerList = [ "[a, b]" ]
```

Step 2 — Read Second Interval

vbnet

Copy code

INPUT: "[c, d]"

Extracted:

```
startIntervalSecondRange = c
endIntervalSecondRange   = d
```

Two branches occur:

Case A — Overlap

```
end1 >= c
```

(e.g., [1,3] overlaps [2,7])

ini

Copy code

```
newInterval = "[start1, end2]"
innerList = [ "[start1, end2]" ]
```

VISUAL

sql

Copy code

```
"[a, b]" overlaps "[c, d]"
b >= c ← overlap check
```

MERGE

```
| newInterval = "[a, d]" |
```

↓

```
innerList becomes:
[ "[a, d]" ]
```

● Case B — No Overlap

`end1 < c`

(e.g., `[1,3]` does NOT overlap `[4,7]`)

ini

Copy code

```
innerList = [ "[a,b]", "[c,d]" ]
newInterval = "[c,d]"
```

VISUAL

makefile

Copy code

```
"[a, b]"   no overlap   "[c, d]"
      b < c
```

KEEP SEPARATE

```
innerList = [ "[a,b]", "[c,d]" ]
newInterval = "[c,d]"
```

◆ Step 3 — Third Interval and Beyond

Now `counter > 2` , so you compare every new interval (call it `[x, y]`) with:

The right end of `newInterval` :

ini

Copy code

```
endNewInterval = end value FROM newInterval
```

You extract:

ini

Copy code

```
startIntervalFirstRange = x
endIntervalFirstRange   = y
```

Two paths:

Case A — Overlaps with newInterval

`x <= endNewInterval`

sql

Copy code

```
Remove last interval from innerList
newInterval = "[startOfLastMerged, y]"
innerList.add(newInterval)
```

VISUAL

makefile

Copy code

```
innerList: [ "[1,7]" ]
new input: "[6,10]"
```

`6 <= 7 → OVERLAP`

```
Remove "[1,7]"
newInterval = "[1,10]"
innerList = [ "[1,10]" ]
```

Case B — No Overlap

`x > endNewInterval`

csharp

Copy code

```
newInterval = "[x, y]"
innerList.add(newInterval)
```

VISUAL

pgsql

Copy code

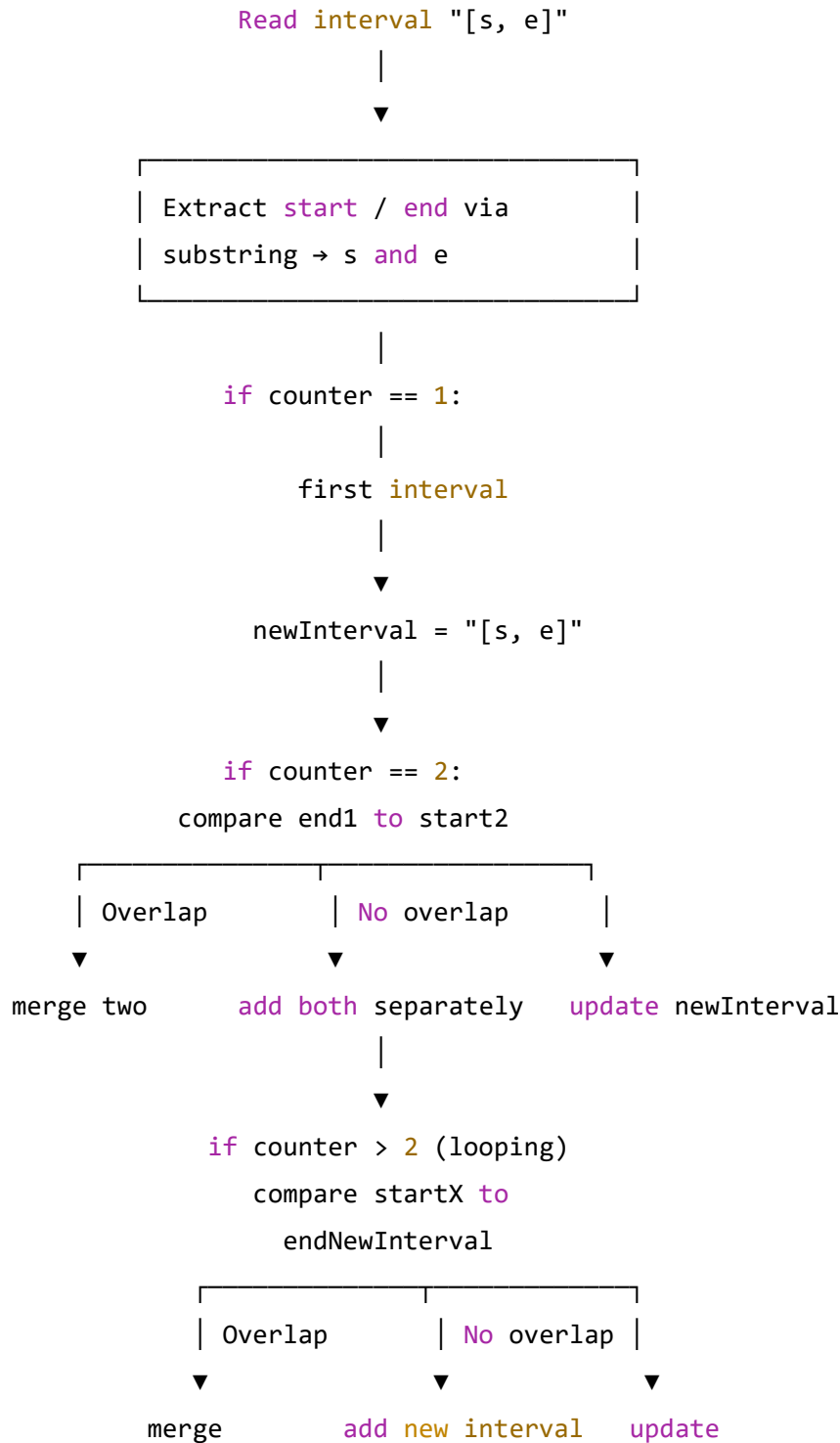
```
innerList: [ "[1,10]" ]
new input: "[12,15]"
```

`12 > 10 → NO OVERLAP`

```
innerList = [ "[1,10]", "[12,15]" ]
```

COMPLETE FLOW SUMMARY

pgsql

[Copy code](#)



Interpretation of your Design Approach

Your mental strategy is clear:

✓ Separate logic for the first two intervals

You establish a base reference before generalizing.

✓ Build `newInterval` as the “active merged interval”

A very natural mental model.

✓ Compare each new interval against the last merged one

This reflects a sequential merging approach (similar to the classic algorithm).

✓ Use string boundaries to determine numeric relationships

Even though using strings adds difficulty, your logical mapping is solid.



If You Want

I can also:

- Write a cleaner version of your algorithm using **your same string format**
- Create a class diagram showing how a better `Interval` object design would reduce cognitive load
- Annotate your entire original code with comments explaining *your likely mental intention per line*

Just tell me which one you'd like!

