# Interval Intersection Code (with Comments)

class Solution:  
 def intervalIntersection(self, firstList: list[list[int]], secondList: list[list[int]]) -> list[list[int]]:  
   
 # This will store our final answer  
 intersections = []  
   
 # i is the pointer for firstList (Alice's schedule 👩‍💻)  
 # j is the pointer for secondList (Bob's schedule 👨‍💻)  
 i = 0  
 j = 0  
   
 # We loop as long as BOTH people still have items in their schedule  
 while i < len(firstList) and j < len(secondList):  
   
 # Get Alice's current interval  
 a\_start, a\_end = firstList[i] # e.g., [0, 2]  
   
 # Get Bob's current interval  
 b\_start, b\_end = secondList[j] # e.g., [1, 5]  
   
 # --- 1. Find the potential overlap --- 🔍  
   
 # The overlap can only START when the \*last\* person arrives.  
 # max(Alice's start, Bob's start)  
 start\_overlap = max(a\_start, b\_start)  
   
 # The overlap must END as soon as the \*first\* person leaves.  
 # min(Alice's end, Bob's end)  
 end\_overlap = min(a\_end, b\_end)  
   
 # --- 2. Check if the overlap is valid --- ✅  
   
 # An overlap is valid only if the start time is before or   
 # at the same time as the end time.  
 # (e.g., start=1, end=2 is valid. start=3, end=2 is not)  
 if start\_overlap <= end\_overlap:  
 # If it's valid, add it to our answer list  
 intersections.append([start\_overlap, end\_overlap])  
   
 # --- 3. Move the correct pointer --- ⏩  
   
 # This is the key to the Two Pointers technique.  
 # We must move the pointer of the person who finishes first.  
   
 # If Alice finishes her current task first (or at the same time)  
 if a\_end <= b\_end:  
 # She's done with this task. Move to her next task.  
 i += 1  
 else:  
 # Bob finishes his current task first  
 # He's done. Move to his next task.  
 j += 1  
   
 # The loop ends when i or j goes past the end of its list.  
 # We return all the intersections we found.  
 return intersections

**The Rule**

You always move the pointer of the person (or list) whose **current task ends first**.

Think of it as "checking off" a task from a to-do list. You can only check off a task once it's **guaranteed to be finished** and can no longer interact with the other person's schedule.

**Why This Works (The Logic)**

Let's use our two people, Alice (list A) and Bob (list B).

**Case 1: Alice's task finishes first (a\_end <= b\_end)**

* **Alice's Task (A):** [5, 10]
* **Bob's Task (B):** [8, 12]

Alice finishes at time **10**. Her task [5, 10] is now **completely done**.

**Question:** Can this *specific* task [5, 10] overlap with any of Bob's *future* tasks (like [15, 20], [22, 30], etc.)?

**Answer:** **No.** Bob's future tasks will all start *after* his current one (e.g., after time 8, and definitely after time 12). Since Alice's task ended at 10, it's impossible for it to overlap with anything starting at 12 or later.

**Conclusion:** We are 100% done with Alice's [5, 10] task. We "check it off" and move to her next task. **Action:** i += 1

**Case 2: Bob's task finishes first (b\_end < a\_end)**

* **Alice's Task (A):** [5, 20]
* **Bob's Task (B):** [8, 12]

Bob finishes at time **12**. His task [8, 12] is now **completely done**.

**Question:** Can this *specific* task [8, 12] overlap with any of Alice's *future* tasks (like [21, 30])?

**Answer:** **No.** Alice's future tasks will all start *after* her current one (after time 20). Since Bob's task ended at 12, it's impossible for it to overlap.

**Conclusion:** We are 100% done with Bob's [8, 12] task. We "check it off" and move to his next task. **Action:** j += 1

By only "checking off" the task that finishes first, you make sure the task that lasts longer (like Alice's [5, 20] in the second case) gets a chance to be compared against the other person's *next* task.