Explanantion

Why Your Original Code Failed:

1. **You only tracked maximum departure time**: Your code kept updating `max_dep` but never considered when trains actually left. This meant you were checking if new trains overlapped with a "phantom" time window.
2. **No platform release**: You incremented platform count but never decremented it when trains departed.
3. **Wrong overlap detection**: You compared `cur_arr <= max_dep`, but this doesn't tell you how many trains are *actually* at the station simultaneously.

The Correct Mental Model:

Think of it like a **parking lot**:

- When a car arrives → occupy a parking spot
- When a car leaves → free up that parking spot
- The question is: "What's the maximum number of cars in the parking lot at any single moment?"

Key Algorithm Steps:

1. **Process events chronologically**: Both arrivals and departures, in time order
2. **Track current occupancy**: +1 for arrival, -1 for departure
3. **Record the peak**: The maximum simultaneous occupancy is your answer

Try the different examples in the visualizer above. You'll see how:

- **Simple Case**: Trains never overlap, so only 1 platform needed
- **Complex Case**: Multiple trains overlap, requiring 3 platforms at peak
- **Your Failing Case**: Two trains overlap completely, needing 2 platforms

Core Concept

The algorithm simulates processing arrival and departure events in chronological order by using two sorted arrays and two pointers.

Step-by-Step Breakdown:

1. Sorting the Arrays

```
arr.sort()# [900, 940, 950, 1100, 1500, 1800]
dep.sort()# [910, 1120, 1130, 1200, 1900, 2000]
```

- We sort arrivals and departures **separately**
- This allows us to process the earliest arrival and earliest departure at each step

2. Two-Pointer Technique

```
pt1 = 0# Points to next arrival to process
pt2 = 0# Points to next departure to process
```

3. Event Processing Logic

```
if arr[pt1] <= dep[pt2]:
    platform_count += 1# A train arrives
    pt1 += 1
else:
```

```
    platform_count -= 1# A train departs
    pt2 += 1
```

Why This Works:

The key insight is that **we only care about the chronological order of events**, not which specific train is arriving/departing.

- **Next arrival time**: `arr[pt1]` (earliest unprocessed arrival)
- **Next departure time**: `dep[pt2]` (earliest unprocessed departure)
- **If arrival ≤ departure**: Process the arrival first
- **If departure < arrival**: Process the departure first

Example Walkthrough:

```
arr = [900, 940, 950, 1100, 1500, 1800] (sorted)
dep = [910, 1120, 1130, 1200, 1900, 2000] (sorted)
```

Step	arr[pt1]	dep[pt2]	Event	Platforms	Max
1	900	910	900 ≤ 910 → Arrival	1	1
2	940	910	940 > 910 → Departure	0	1
3	940	1120	940 ≤ 1120 → Arrival	1	1
4	950	1120	950 ≤ 1120 → Arrival	2	2
5	1100	1120	1100 ≤ 1120 → Arrival	3	**3**
6	1500	1120	1500 > 1120 → Departure	2	3
7	1500	1130	1500 > 1130 → Departure	1	3
8	1500	1200	1500 > 1200 → Departure	0	3
9	1500	1900	1500 ≤ 1900 → Arrival	1	3
10	1800	1900	1800 ≤ 1900 → Arrival	2	3

Answer: 3 platforms

Why Separate Sorting Works:

Even though we lose track of which train corresponds to which arrival/departure, **it doesn't matter**! We only need to know:

- How many trains are arriving vs departing
- In what chronological order

The algorithm correctly simulates the "pressure" on platforms by tracking when the station is busiest.

Edge Case Handling:

```
if arr[pt1] <= dep[pt2]:# Note the <= (not just <)
```

When arrival and departure happen at the same time, we process the arrival first. This is the **worst-case scenario** - if a train can arrive exactly when another departs, we conservatively assume they need separate platforms.