Minimum Platforms Problem

- The Minimum Platforms Problem is a classic scheduling problem where the goal is to determine the minimum number of platforms required at a railway station so that no train is delayed due to the unavailability of platforms.

- **Real-World Relevance**:
  - Efficient scheduling of trains.
  - Resource optimization in event management and transportation.

- **Key Challenge**:
  - Handling overlapping intervals efficiently.

Problem Statement

- **Given**:

1. **Arrival Times**: A list of times at which trains arrive at the station.
2. **Departure Times**: A list of times at which trains leave the station.

- **Find**:

- The **minimum number of platforms** required so that no two trains overlap in terms of their presence on the same platform.

Constraints

1. Each train occupies a platform starting at its **arrival time** and leaves at its **departure time**.
2. No two trains can share a platform if their schedules overlap.
3. Arrival and departure times are in 24-hour format (e.g., 900,1230).

Problem Description

- **Input** : n :- Number of trains.
- Two arrays:
  - Arrival times: Times when trains arrive.
  - Departure times: Times when trains leave.
- **Output** : The minimum number of platforms required so that no two trains overlap on the same platform.
- **Constraints**: $1 \leq n \leq 100,000$
- Arrival and departure times are in the 24-hour format.

Naive Solution

- **Approach**:
  - Compare every train with every other train to check overlap.
  - Count the number of overlapping trains for each train.
- **Algorithm**:
  - For each train, iterate through all other trains.
  - Increment the count if two trains overlap.
  - Track the maximum count.
- **Time Complexity**: $O(n^2)$
- **Drawback**: Computationally expensive for large value of n.

Efficient Greedy Algorithm

- **Key Idea**:
  - Use sorting and two pointers to efficiently count overlaps.
- **Algorithm Steps**:
  - Sort the arrival and departure arrays.
  - Use two counter:
    - Increment counter for arrival when a train arrives.
    - Increment counter for departure when a train departs.
  - Keep track of the number of platforms needed and update the maximum count.
- **Time Complexity**: O(nlgn) (considering sorting algorithm complexity).

- **Steps of the Algorithm**

1. **Input**:
    1. Two arrays:
        1. Arrival [ ] : Arrival times of (n) trains.
        2. Departure [ ] : Departure times of (n) trains.
    2. n : Number of trains.
2. **Sort**:
    1. Sort the arrival [ ] array in ascending order (accordingly departure [ ] array will be rearranged.)
3. **Initialize Variables**:
    1. platforms = 0: Tracks the current number of platforms needed.
    2. max_platforms = 0: Tracks the maximum number of platforms needed at any time.
    3. Two pointers/variables/counter:
        1. i = 0: pointer/counter to traverse the arrival [ ] array.
        2. j = 0: pointer/counter to traverse the departure [ ] array.
4. **Traverse the Arrays**:
    1. While both pointers (i and j) are within bounds:
        1. If the arrival time of the next train is less than or equal to the departure time of the current train:
            1. Increment platforms (a new platform is required).
            2. Move the i pointer to the next train (i++).
        2. Otherwise:
            1. Decrement platforms (a platform is free / available as a train departs).
            2. Move the j pointer to the next departure (j++).
        3. Update max_platforms with the maximum value of platforms.
5. **Return**:
    1. The value of max_platforms is the minimum number of platforms required.

Efficient for Large Arrays and Small Max Time] O(n + maxTime) time and O(maxTime) space

*The **sweep line algorithm** is an efficient method for solving problems involving intervals or segment. The idea is to convert the arrival time and departure time of each train in the form of (x, y) coordinate, and then apply the sweep line algorithm to finding the maximum number of overlap at any time.*

Time Complexity: **O(n)**, where n is the number of trains.

Auxiliary space: **O(maxDepartureTime)**

Algorithm

1. First, create an array of size greater than maximum **Departure time** to track the number of platforms needed at each time point of time.
2. Iterate over the **arrival** and **departure times**: for each **arrival time**, do **v[arrival time] += 1** and for each departure time, do **v[departure time + 1] -= 1**.
3. This process effectively marks the times when platforms are occupied and when they are free.
4. After updating the array, we calculate the **cumulative sum** of this array to find the maximum number of platforms needed at any time.
5. This maximum value represents the minimum number of platforms required.

Applications

- **Real-World Use Cases**:
  - **Railway Station Management**:
    - Scheduling trains to avoid delays.
  - **Airport Runway Allocation**:
    - Efficiently assigning runways to arriving and departing flights.
  - **Event Scheduling**:
    - Managing overlapping sessions in conferences or events.
  - **CPU Scheduling**:
    - Allocating CPU resources to overlapping processes in multitasking systems.
- **Why It’s Important**:
  - Resource efficiency.
  - Minimizing delays.

- **Example**
- **Input:**
- arrival = [900, 940, 950, 1100, 1500, 1800]
- departure = [910, 1200, 1120, 1130, 1900, 2000]
- **Execution:**

1. **Sorted Arrays**:
   1. arrival = [900, 940, 950, 1100, 1500, 1800]
   2. departure = [910, 1120, 1130, 1200, 1900, 2000]
2. **Step-by-Step Calculation**:
   1. At 900: Train arrives, platforms=1.
   2. At 910: Train departs, platforms=0.
   3. At 940: Train arrives, platforms=1.
   4. At 950: Train arrives, platforms=2.
   5. At 1100: Train arrives, platforms=3 (max).
   6. At 1120: Train departs, platforms=2.
   7. At 1130: Train departs, platforms=1.
   8. At 1200: Train departs, platforms=0.
   9. Repeat for remaining trains.
3. **Output**:
   1. max_platforms = 3

Conclusion

- **Key Takeaways**:
  - Efficient scheduling is crucial in resource allocation.
  - The greedy approach ensures an optimal solution for large datasets.
  - Sorting and two-counter techniques simplify the problem.

- **Future Scope**:
  - Extending the problem to dynamic scheduling scenarios (e.g., real-time train schedules).
  - Incorporating additional constraints like train priority.