Minimum Platforms :-

Minimum Platforms

Difficulty: **Medium** Accuracy: **26.84%** Submissions: **531K+** Points: **4**

You are given the arrival times **arr[]** and departure times **dep[]** of all trains that arrive at a railway station on the same day. Your task is to determine the minimum number of platforms required at the station to ensure that no train is kept waiting.

At any given time, the same platform cannot be used for both the arrival of one train and the departure of another. Therefore, when two trains arrive at the same time, or when one arrives before another departs, additional platforms are required to accommodate both trains.

Examples:

Input: arr[] = [900, 940, 950, 1100, 1500, 1800], dep[] = [910, 1200, 1120, 1130, 1900, 2000]
Output: 3
Explanation: There are three trains during the time 9:40 to 12:00. So we need a minimum of 3 platforms.

Input: arr[] = [900, 1235, 1100], dep[] = [1000, 1240, 1200]
Output: 1
Explanation: All train times are mutually exclusive. So we need only one platform

Input: arr[] = [1000, 935, 1100], dep[] = [1200, 1240, 1130]
Output: 3
Explanation: All 3 trains have to be there from 11:00 to 11:30

Constraints:
1≤ number of trains ≤ 50000
0000 ≤ arr[i] ≤ dep[i] ≤ 2359
Note: Time intervals are in the 24-hour format(**HHMM**) , where the first two characters represent hour (between 00 to 23) and the last two characters represent minutes (this will be <= 59 and >= 0).

Try more examples

eg: I/p: arr[] = [900, 940, 950, 1100, 1500, 1800]
dep[] = [910, 1200, 1120, 1130, 1900, 2000]

O/p: 3

900 - 910
940 - 1200
950 - 1120
1100 - 1130
1500 - 1900
1800 - 2000

(Naive Approach) Using Two Nested Loops - O(n^2) time & O(1) space :-

The idea is to iterate through each train & for that train, check how many other trains have overlapping timings with it, where current train's arrival time falls b/w the other train's arrival & departure times. We keep track of this count for each train & continuously update our answer with the maximum count found.

Or


```
int minPlatform(vector<int> &arr, vector<int> &dep) {
    int n = arr.size();
    int res = 0;
    for(int i=0; i<n; i++){
        int cnt = 1;
```

[Expect Approach 1] Using Sorting & Two Pointers -

O(nlogn) Time & O(1) Space :-

This approach uses sorting & two-pointer to reduce the complexity. First, we sort the arrival & departure times of all trains. Then, using two pointers, we traverse through both arrays.

- The idea is to maintain a count of platforms needed at any point of time.

Step by Step Implementation :-

- Sort the arrival & departure times so we can process train timings in order.
- Initialize two pointers :-
  - One for tracking arrivals (i=0)
  - One for tracking departures (j=0).
- Iterate through the arrival times.
  - If the current train arrives before or at the departure of an earlier train, allocate a new platform (cnt++).
  - Otherwise, if the arrival time is greater than the departure time, it means a train has left, freeing up a platform (cnt--), & move the departure pointer forward (j++).
- Update the maximum no. of platforms required after each step.
- Repeat until all trains are processed.

after each step.

- Continue this process until all trains are processed.

```
int minPlatform(vector<int> &arr, vector<int> &dep){
    int n= arr.size();
    int res= 0;
    // sort the arrays
    sort(arr.begin(),arr.end());
    sort(dep.begin(), dep.end());
    // pointer to track the departure times
    int j=0;
    // Tracks the number of platforms needed at any
    // given time
    int cnt = 0;
    // check for each train
    for(int i=0; i<n;i++){
    // Decrement count if other trains have left
    while(j<n && dep[j]<arr[i]){
            cnt--;
            j++;}
    // one platform for each train
    cnt++;
    res= max(res,cnt);}
    return res;
}
```

[Expected Approach2] Using Sweep Line Algorithm :-

The Sweep Line algorithm is an efficient technique for solving interval based problems. It works by treating each train's arrival & departure times as events on a timeline. By processing these events in chronological order, we can track the number of trains at the station at any moment, which directly indicates the number of platforms required at that time. The maximum no. of overlapping trains during this process determines the minimum no. of platforms needed.

// Step Implementation :-

determines the minimum no. of pl...

Step by Step Implementation :-

- Create an array v[] of size greater than the maximum departure time. This array will help track the number of platforms needed at each time.
- Mark arrivals & departures;
  - For each arrival time, increment v[arival - time] by 1, indicating that a platform is needed.
  - For each departure time, decrement v[departure_time] by 1, indicating that a platform is freed as the train has left
- Iterate through v[] & compute the cumulative sum.
- The running sum keeps track of the no. of trains present at any given time.
- The maximum value encountered represents the minimum number of platforms needed.

```
int minPlatform(vector<int>& arr, vector<int>& dep){
    int n= arr.size();
    int res=0;
  // Find the max Departure time
    int maxDep = dep[0];
    for(int i=1; i<n; i++){
       maxDep= max(maxDep, dep[i]); }
  //Create a vector to store the count of trains at
  // each time
    vector<int> v(maxDep+2,0);
  //Increment the count at the arrival time & decrement
  // at the departure time
    for(int i=0;i<n; i++){
          v[arr[i]]++;
          v[dep[i]+1]--;
        }
    int count = 0;
  //Iterate over the vector & keep track of maximum
  // sum seen so far.
```

```
for(int i=0; i<=maxDep+1; i++){
        count += v[i];
        res = max(res, count);
      }
    return res;
  }
```

T.C. $O(n+k)$ where n = no. of trains & k = max. value present in the arrays.

S.C. $O(k)$, where k = max. value present in both the arrays.