**Solution**

**Approach: Greedy**

**Intuition**

Finding the minimum number of intervals to remove is equivalent to finding the maximum number of non-overlapping intervals. This is the famous [interval scheduling problem](https://en.wikipedia.org/wiki/Interval_scheduling).

Let's start by considering the intervals according to their end times. Consider the two intervals with the earliest end times. Let's say the earlier end time is x and the later one is y. We have x < y.

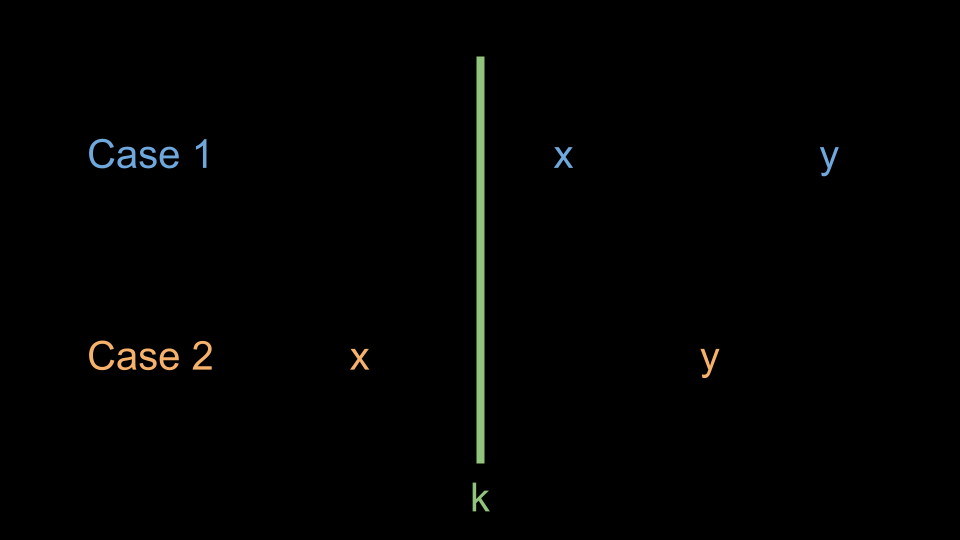
If we can only choose to keep one interval, should we choose the one ending at x or ending at y? To avoid overlap, We should always greedily choose the interval with an earlier end time x. The intuition behind this can be summarized as follows:

* We choose either x or y. Let's call our choice k.
* To avoid overlap, the next interval we choose must have a start time greater than or equal to k.
* We want to maximize the intervals we take (without overlap), so we want to maximize our choices for the next interval.
* Because the next interval must have a start time **greater** than or equal to k, a larger value of k can **never** give us more choices than a smaller value of k.
* As such, we should try to minimize k. Therefore, we should always greedily choose x, since x < y.

In general, k is equal to the end time of the most recent interval we kept.

Start by sorting intervals according to the end times so that we can process the intervals in order. We'll keep the variable k described above.

As we iterate over the intervals, let's consider the possible cases. In the following image, k is the vertical line, and x, y is the next interval to be considered.



Because we sorted the end times, y must be greater than k. This gives us two cases:

* Case 1, x >= k: we can safely take this interval because it won't cause an overlap. We should update k = y since this interval is now the most recent interval we are keeping.
* Case 2, x < k: taking this interval would cause an overlap. As we established earlier, we should always take intervals with earlier end times. Because y > k, we must delete the current interval (don't update k).

For those interested in a formal proof of this algorithm's correctness, please refer to [this paper](https://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect07-greedy-sched.pdf), pages 2 - 4.

**Algorithm**

1. Sort intervals according to the end times.
2. Initialize an answer variable ans = 0 and an integer k to represent the most recent end time. k should be initialized to a small value like INT\_MIN.
3. Iterate over the intervals. For each interval:
   * If the start time is greater than or equal to k, update k to the end time of the current interval.
   * Otherwise, increment ans.
4. Return ans.

**Implementation**

bool compareSecondElement(vector<int>& a, vector<int>& b) {

    return a[1] < b[1];

}

class Solution {

public:

    int eraseOverlapIntervals(vector<vector<int>>& intervals) {

        sort(intervals.begin(), intervals.end(), compareSecondElement);

        int ans = 0;

        int k = INT\_MIN;

        for (int i = 0; i < intervals.size(); i++) {

            int x = intervals[i][0];

            int y = intervals[i][1];

            if (x >= k) {

                // Case 1

                k = y;

            } else {

                // Case 2

                ans++;

            }

        }

        return ans;

    }

};

**Complexity Analysis**

Given nn*n* as the length of intervals,

* Time complexity: O(n⋅log⁡n)O(n \cdot \log n)*O*(*n*⋅log*n*)

We sort intervals, which costs O(n⋅log⁡n)O(n \cdot \log n)*O*(*n*⋅log*n*). Then, we iterate over the input, performing constant time work at each iteration. This means the iteration costs O(n)O(n)*O*(*n*), which is dominated by the sort.

* Space Complexity: O(log⁡n)O(\log n)*O*(log*n*) or O(n)O(n)*O*(*n*)

The space complexity of the sorting algorithm depends on the implementation of each programming language:

* + In Java, Arrays.sort() is implemented using a variant of the Quick Sort algorithm, which has a space complexity of O(log⁡n)O(\log n)*O*(log*n*)
  + In C++, the sort() function provided by STL uses a hybrid of Quick Sort, Heap Sort and Insertion Sort, with a worst case space complexity of O(log⁡n)O(\log n)*O*(log*n*)
  + In Python, the sort() function is implemented using the Timsort algorithm, which has a worst-case space complexity of O(n)O(n)*O*(*n*)