# 630. Course Schedule III

Hard Greedy Array Sorting Heap (Priority Queue) Leetcode Link

## Problem Description

In this problem, you are given `n` online courses, each with a specified duration and a deadline by which it should be completed. The courses must be taken one at a time, consecutively, from day 1 onwards. The goal is to find out the most number of courses you can finish before their respective deadlines.

Each course is described by two parameters: `duration` (how many days it takes to complete the course) and `lastDay` (the last day by which the course must be completed). You are provided this information in an array `courses`, where each element `courses[i]` is another array with two elements: `[duration_i, lastDay_i]`.

Your task is to pick the courses in such a way that you can complete the maximum number of them within their deadlines, keeping in mind that you cannot take more than one course at a time.

## Intuition

The solution to this problem involves sorting and priority queues, specifically a min-heap. The first step is to sort the courses by their deadline (`lastDay`). This initial sort helps to align our course selections in the direction that we attempt to tackle courses with earlier deadlines first.

After sorting the courses by their deadlines, we iterate through each course. As we go through the courses, we add each course's duration to a total duration accumulator (`s`) and push the negative of duration onto a min-heap (`pq`). We use a min-heap to keep track of the courses with the longest duration that we have attended to, and the reason for storing negative durations is to effectively turn the min-heap into a max-heap (since Python's `heapq` only provides a min-heap).

If at any point our total duration accumulator (`s`) exceeds the current course's last day, it means we cannot finish the current course before the deadline. To rectify this, we start removing the longest courses we have taken from our schedule (the top of the heap), until `s` no longer exceeds the last day of the current course.

By iteratively adding courses and potentially removing the longest course from our schedule whenever necessary, we ensure that we are always in a position to have completed the maximum number of courses by their respective deadlines.

The solution's efficiency comes from the fact that when we must drop a course, we drop the one with the longest duration, which has the most significant potential to free up our schedule for fitting in more courses.

The final answer is the number of courses we have in our priority queue, as this represents the maximum number of courses we can complete within their deadlines.

## Solution Approach

The solution takes a greedy approach along with the use of a priority queue to maximize the number of courses taken. Here, we go through the algorithm and patterns used in the solution:

1. **Sorting**: Initially, we sort the `courses` array by their deadlines, which is `lastDay_i` for the `i-th` course. This is done to prioritize courses with the earliest deadlines, allowing us to take courses in a sequence that respects those deadlines.

   ```
   courses.sort(key=lambda x: x[1])
   ```

2. **Priority Queue (Min-Heap)**: We use a priority queue (min-heap) to keep track of the courses we have decided to take. Since Python's `heapq` library only supports a min-heap and we want to have the functionality of a max-heap (to easily pop the longest course when needed), we store negative durations in the heap.

   The priority queue (heap) operations used are:

   - `heappush` to add an element to the heap
   - `heappop` to remove the smallest element, which in our case will be the course with the longest duration due to the negation

   ```
   pq = []
   ```

3. **Iterating Through the Courses**: We iterate through the sorted `courses` list, and for each course, we perform the following steps:

   a. First, we push the negative duration of the current course onto the heap and add the duration to an accumulator `s`.

   ~~~~
   ```python
   heappush(pq, -duration)
   s += duration
   ```
   ~~~~

   b. Then, we check if adding the current course has caused the total duration (`s`) to exceed the last day by which the course must be completed. If it does, we enter a while loop:

   ~~~~
    - Inside the loop, we pop the course with the longest duration (which is at the top of the heap due to being the smallest negat

    - We continue this process until the total duration `s` no longer exceeds the deadline of the current course or until the heap

    ```python
    while s > last:
        s += heappop(pq)
    ```
   ~~~~

4. **Count of Courses Taken**: After iterating through all courses and adjusting our schedule as necessary, the remaining courses in the heap represent the courses that we can complete within their respective deadlines. Hence, the length of the heap gives us the maximum number of courses that can be taken.

   ```
   return len(pq)
   ```

By using the greedy strategy of always picking the next course that can be finished before its deadline after the courses already scheduled, and by adjusting the schedule when necessary by dropping the longest course, the algorithm ensures we optimize for the maximum number of courses taken.

### Example Walkthrough

To illustrate the solution approach, let's use a small example of the courses array:

```
courses = [[100, 200], [70, 150], [120, 210]]
```

1. **Sorting by Deadlines**

   First, we sort the courses by their deadlines:

   - `courses.sort(key=lambda x: x[1])`

   After sorting:

   ```
   courses = [[70, 150], [100, 200], [120, 210]]
   ```

   Now, our courses array is sorted by the last day on which the courses need to be completed, enabling us to focus on the courses with the earliest deadlines first.

2. **Initializing the Priority Queue**

   We create a priority queue (min-heap) to help us keep track of the courses we're taking.

   - `pq = []`

   Initially, the priority queue is empty.

3. **Iterating Through the Courses**

   We go through each course and follow a series of steps.

   a. For the first course `[70, 150]`:

   - We add the negative course duration to the heap and update the accumulator, `s`:
     ```
     heappush(pq, -70)
     s = 70
     ```

   b. The total duration `s` does not exceed the last day of the current course (150), so we move to the next course.

   a. For the second course `[100, 200]`:

   - We add the negative course duration to the heap and update the accumulator, `s`:
     ```
     heappush(pq, -100)
     s = 170
     ```

   b. The total duration `s` does not exceed the last day of the current course (200), so we move to the next course.

   a. For the third course `[120, 210]`:

   - We add the negative course duration to the heap and update the accumulator, `s`:
     ```
     heappush(pq, -120)
     s = 290
     ```

   b. Now the total duration `s` exceeds the last day of the current course (210). We need to adjust our schedule:

   - We keep removing the longest duration courses from the pile (the one with the least negative value) until `s` is less than or equal to the deadline:

     ```
     s += heappop(pq)    # Removes -120, adding it back to s gives s = 290 - 120 = 170
     ```

   - As `s` is now 170, it still exceeds the deadline for the course, so we continue the process:

     ```
     s += heappop(pq)    # This time, removes -100, updating s to s = 170 - 100 = 70
     ```

   - Now, with `s` equal to 70, it does not exceed the deadline for the third course, so we can stop adjusting the schedule.

4. **Count of Courses Taken**:

   After iterating through all the courses and adjusting the schedule, the number of courses left in the queue is the number we can complete within their respective deadlines. So the final answer is:

   - `return len(pq)`

   Since the heap now contains only the first course `[-70]`, we get the final result:

   ```
   Maximum number of courses = 1
   ```

This example highlights the greedy decision-making process: by sorting the courses by deadlines and using a priority queue to manage durations, dropping the longest course if the total duration exceeds a course's deadline, we optimize for the maximum number of courses that can be completed.

## Python Solution

```python
from heapq import heappush, heappop

class Solution:
    def scheduleCourse(self, courses: List[List[int]]) -> int:
        # Sort the courses based on their end day
        courses.sort(key=lambda x: x[1])

        # Initialize a max heap to keep track of the longest courses we have taken
        max_heap = []

        # This variable will hold the total duration of all courses we have taken so far
        total_duration = 0

        # Iterate over each course
        for duration, last_day in courses:
            # Add the current course to the max heap and increment the total duration
            heappush(max_heap, -duration)
            total_duration += duration

            # If the total duration exceeds the last day of the current course,
            # remove the longest course from our schedule
            while total_duration > last_day:
                # heappop returns a negative number because we are using max heap,
                # so we add it back to reduce the total duration
                total_duration += heappop(max_heap)

        # The number of courses in the max heap is the number of courses we can take
        return len(max_heap)

```

## Java Solution

```java
class Solution {
    public int scheduleCourse(int[][] courses) {
        // Sort the courses by their end day
        Arrays.sort(courses, (course1, course2) -> course1[1] - course2[1]);

        // Create a max-heap (priority queue) to keep track of the longest courses taken so far
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>((duration1, duration2) -> duration2 - duration1);

        int currentTime = 0; // Keep track of the total duration of all courses taken so far

        // Iterate through each course
        for (int[] course : courses) {
            // Extract the course duration and the last day the course can be taken
            int duration = course[0], lastDay = course[1];

            // Add the current course duration to the heap and increment total time
            maxHeap.offer(duration);
            currentTime += duration;

            // If the current time exceeds the end day of the last course, remove the longest course duration
            while (currentTime > lastDay) {
                currentTime -= maxHeap.poll(); // This keeps us within the last day limit
            }
        }

        // The size of the heap indicates the maximum number of courses that can be taken
        return maxHeap.size();
    }
}

```

## C++ Solution

```cpp
#include <vector>
#include <algorithm> // for std::sort
#include <queue>     // for std::priority_queue

class Solution {
public:
    int scheduleCourse(vector<vector<int>>& courses) {
        // Sort the input vector of courses by their end days using a custom comparator.
        sort(courses.begin(), courses.end(), [](const vector<int>& courseA, const vector<int>& courseB) {
            return courseA[1] < courseB[1];
        });

        // Use a max-heap (priority queue) to keep track of the longest courses taken.
        priority_queue<int> maxHeap;
        int totalTime = 0; // This will accumulate the total time taken by selected courses.

        // Iterate through each course in the sorted course list.
        for (auto& course : courses) {
            int duration = course[0]; // Duration of the current course.
            int deadline = course[1]; // Deadline (last day) of the current course.

            // Add the duration to the total time and push it onto the max-heap.
            maxHeap.push(duration);
            totalTime += duration;

            // If the total time exceeds the current course's deadline,
            // remove the longest course from the total time and the max-heap.
            // Repeat this process to ensure we stay within the deadline.
            while (totalTime > deadline) {
                totalTime -= maxHeap.top();
                maxHeap.pop();
            }
        }

        // The number of courses in the max-heap gives us the maximum number of courses
        // that can be taken without exceeding their respective deadlines.
        return maxHeap.size();
    }
};

```

## Typescript Solution

```typescript
// Importing the MaxPriorityQueue from a library such as 'typescript-collections'
import { MaxPriorityQueue } from 'typescript-collections';

/**
 * Finds the maximum number of courses that can be taken given the duration and last day to finish each course.
 * @param courses - An array of pairs [duration, lastDay], where duration represents how long the course takes to complete,
 *  and lastDay represents the last day the course can be finished.
 * @returns The maximum number of courses that can be taken.
 */
function scheduleCourse(courses: number[][]): number {
    // Sorts courses based on their ending time in ascending order
    courses.sort((a, b) => a[1] - b[1]);

    // Initialize a Max Priority Queue to manage the durations of the courses taken
    const priorityQueue = new MaxPriorityQueue<number>();

    // Represents the current accumulated duration of all selected courses
    let totalDuration = 0;

    // Iterate through each course
    for (const [duration, lastDay] of courses) {
        // Insert current course duration into the priority queue
        priorityQueue.enqueue(duration);

        // Add the current course's duration to the total duration
        totalDuration += duration;

        // If the total duration exceeds the current course's last day,
        // remove the course with the longest duration to meet the deadline
        while (totalDuration > lastDay) {
            totalDuration -= priorityQueue.dequeue().element;
        }
    }

    // At this point, the priority queue contains the maximum number of courses that fit within their deadlines
    return priorityQueue.size();
}

// Note: You must include the appropriate library for the MaxPriorityQueue.
// The 'typescript-collections' library is assumed in this example, but you
// might be using a different one, so adjust imports accordingly.

```

## Time and Space Complexity

The time complexity of the above code is `O(n log n)` for sorting the courses, where `n` is the number of courses. The insertion and deletion operations in the priority queue take `O(log k)`, where `k` is the number of courses currently in the queue. Since each course can be pushed onto and popped from the queue once, in the worst case the total cost for these operations is `O(n log k)`. Assuming the queue could have at most `n` courses, the total time complexity of the algorithm becomes `O(n log n)` due to the initial sort dominating the overall complexity.

The space complexity of the code is `O(n)` which is due to the priority queue storing at most `n` courses at any time.