<https://leetcode.com/problems/course-schedule-iii/>

There are n different online courses numbered from 1 to n. You are given an array courses where courses[i] = [durationi, lastDayi] indicate that the ith course should be taken **continuously** for durationi days and must be finished before or on lastDayi.

You will start on the 1st day and you cannot take two or more courses simultaneously.

Return *the maximum number of courses that you can take*.

**Example 1:**

Input: courses = [[100,200],[200,1300],[1000,1250],[2000,3200]]

Output: 3

Explanation:

There are totally 4 courses, but you can take 3 courses at most:

First, take the 1st course, it costs 100 days so you will finish it on the 100th day, and ready to take the next course on the 101st day.

Second, take the 3rd course, it costs 1000 days so you will finish it on the 1100th day, and ready to take the next course on the 1101st day.

Third, take the 2nd course, it costs 200 days so you will finish it on the 1300th day.

The 4th course cannot be taken now, since you will finish it on the 3300th day, which exceeds the closed date.

**Example 2:**

Input: courses = [[1,2]]

Output: 1

**Example 3:**

Input: courses = [[3,2],[4,3]]

Output: 0

**Constraints:**

1 <= courses.length <= 10^4

1 <= durationi, lastDayi <= 10^4

**Attempt 1: 2023-01-09**

**Solution 1: Divide and Conquer + Memoization (360 min, intuitive but TLE)**

class Solution {

    public int scheduleCourse(int[][] courses) {

        Arrays.sort(courses, (a, b) -> a[1] - b[1]);

        Integer[][] memo = new Integer[courses.length][courses[courses.length - 1][1] + 1];

        return helper(courses, 0, 0, memo);

    }

    private int helper(int[][] courses, int index, int time, Integer[][] memo) {

        if(index == courses.length) {

            return 0;

        }

        if(memo[index][time] != null) {

            return memo[index][time];

        }

        // Take

        int take = 0;

        if(time + courses[index][0] <= courses[index][1]) {

            take = 1 + helper(courses, index + 1, time + courses[index][0], memo);

        }

        // Not take

        int not\_take = helper(courses, index + 1, time, memo);

        memo[index][time] = Math.max(take, not\_take);

        return memo[index][time];

    }

}

**Refer to**

<https://leetcode.com/problems/course-schedule-iii/solutions/127745/course-schedule-iii>

**Approach 1: Brute Force**

**Algorithm**

The most naive solution will be to consider every possible permutation of the given courses and to try to take as much courses as possible by taking the courses in a serial order in every permutation. We can find out the maximum number of courses that can be taken from out of values obtained from these permutations.

**Complexity Analysis**

Time complexity : O((n+1)!). A total of n! permutations are possible for the courses *s* array of length n. For every permutation, we scan over the n elements of the permutation to find the number of courses that can be taken in each case.

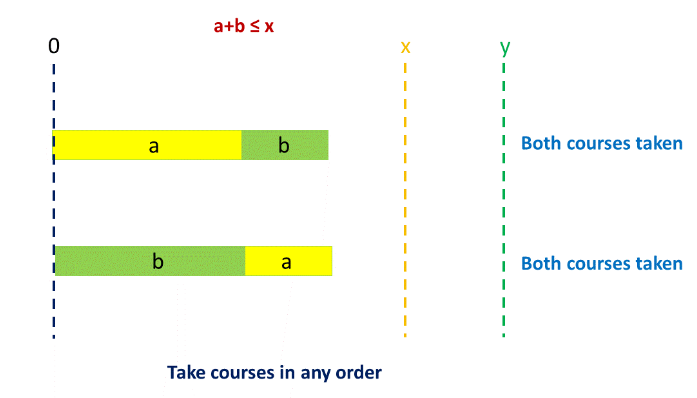
Space complexity : O(n). Each permutation needs n space.

**Approach 2: Recursion with Memoization**

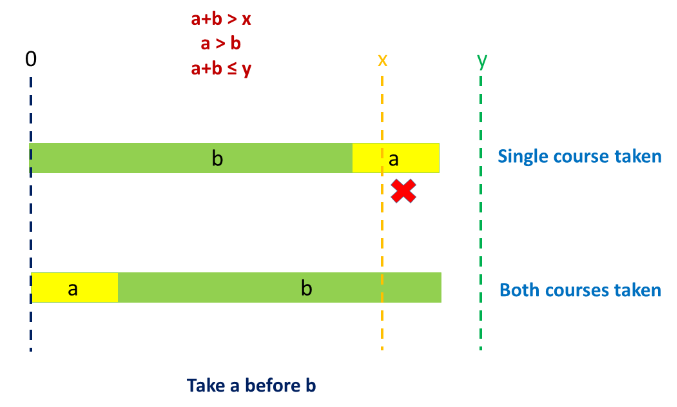
**Algorithm**

Before we move on to the better approaches, let's discuss one basic idea to solve the given problem. Suppose, we are considering only two courses (a,x) and (b,y). Let's assume y>x. Now, we'll look at the various relative values which a and b can take, and which course should be taken first in each of these cases. In all the cases, we assume that the course's duration is always lesser than its end day i.e. a<x and b<y.

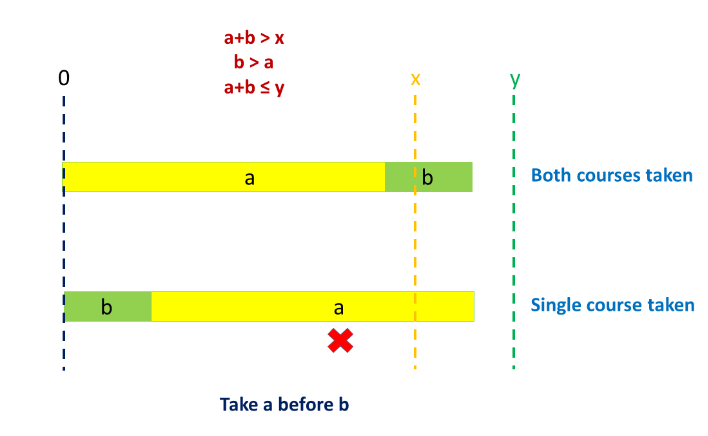
(a+b)≤x: In this case, we can take the courses in any order. Both the courses can be taken irrespective of the order in which the courses are taken.



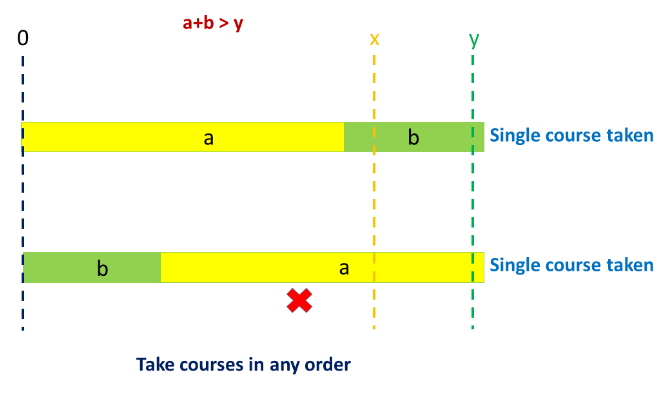
(*a*+*b*)>*x*, a>b, (a+b)≤y: In this case, as is evident from the figure, both the courses can be taken only by taking course a before b.



(*a*+*b*)>*x*, b>a, (a+b)≤y: In this case also, both the courses can be taken only by taking course a before b.



(*a*+*b*)>*y*: In this case, irrespective of the order in which we take the courses, only one course can be taken.



From the above example, we can conclude that it is always profitable to take the course with a smaller end day prior to a course with a larger end day. This is because, the course with a smaller duration, if can be taken, can surely be taken only if it is taken prior to a course with a larger end day.

Based on this idea, firstly, we sort the given courses array based on their end days. Then, we try to take the courses in a serial order from this sorted courses array.

In order to solve the given problem, we make use of a recursive function schedule(courses, i, time) which returns the maximum number of courses that can be taken starting from the ith course(starting from 0), given the time aleady consumed by the other courses is time, i.e. the current time is time, given a courses array as the schedule.

Now, in each function call to schedule(courses, i, time), we try to include the current course in the taken courses. But, this can be done only if time+durationi < end\_dayi​. Here, durationi refers to the duration of the ith course and end\_dayi refers to the end day of the ith course.

If the course can be taken, we increment the number of courses taken and obtain the number of courses that can be taken by passing the updated time and courses' index. i.e. we make the function call schedule(courses, i + 1, time + duration\_i). Let's say, we store the number of courses that can be taken by taking the current course in taken variable.

Further, for every current course, we also leave the current course, and find the number of courses that can be taken thereof. Now, we need not update the time, but we need to update the courses' index. Thus, we make the function call, schedule(courses, i + 1, time). Let's say, we store the count obtained in not\_taken variable.

While returning the number of courses at the end of each function call, we return the maximum value out of taken and not\_taken.

Thus, the function call schedule(courses, 0, 0) gives the required result.

In order to remove this redundancy, we make use of a memoization array memo, such that memo[i][j] is used to store the result of the function call schedule(courses, i, time). Thus, whenever the same function call is made again, we can return the result directly from the memo array. This helps to prune the search space to a great extent.

public class Solution {

    public int scheduleCourse(int[][] courses) {

        Arrays.sort(courses, (a, b) -> a[1] - b[1]);

        Integer[][] memo = new Integer[courses.length][courses[courses.length - 1][1] + 1];

        return schedule(courses, 0, 0, memo);

    }

    public int schedule(int[][] courses, int i, int time, Integer[][] memo) {

        if (i == courses.length)

            return 0;

        if (memo[i][time] != null)

            return memo[i][time];

        int taken = 0;

        if (time + courses[i][0] <= courses[i][1])

            taken = 1 + schedule(courses, i + 1, time + courses[i][0], memo);

        int not\_taken = schedule(courses, i + 1, time, memo);

        memo[i][time] = Math.max(taken, not\_taken);

        return memo[i][time];

    }

}

**Complexity Analysis**

Time complexity : O(n∗d). memo array of size n x d is filled once. Here, n refers to the number of courses in the given courses array and d refers to the maximum value of the end day from all the end days in the courses array.

Space complexity : O(n∗d). memo array of size n x d is used.

**Solution 2:  PriorityQueue + Greedy (60 min)**

class Solution {

    public int scheduleCourse(int[][] courses) {

        Arrays.sort(courses, (a, b) -> a[1] - b[1]);

        PriorityQueue<Integer> maxPQ = new PriorityQueue<Integer>((a, b) -> b - a);

        int time = 0;

        for(int[] course : courses) {

            time += course[0];

            maxPQ.offer(course[0]);

            if(time > course[1]) {

                time -= maxPQ.poll();

            }

        }

        return maxPQ.size();

    }

}

**Refer to**

<https://www.cnblogs.com/grandyang/p/7126289.html>

这道题给了我们许多课程，每个课程有两个参数，第一个是课程的持续时间，第二个是课程的最晚结束日期，让我们求最多能上多少门课。博主尝试了递归的暴力破解，TLE了。**这道题给的提示是用贪婪算法，那么我们首先给课程排个序，按照结束时间的顺序来排序，我们维护一个当前的时间，初始化为0，再建立一个优先数组，然后我们遍历每个课程，对于每一个遍历到的课程，当前时间加上该课程的持续时间，然后将该持续时间放入优先数组中，然后我们判断如果当前时间大于课程的结束时间，说明这门课程无法被完成，我们并不是直接减去当前课程的持续时间，而是取出优先数组的顶元素，即用时最长的一门课，这也make sense，因为我们的目标是尽可能的多上课，既然非要去掉一门课，那肯定是去掉耗时最长的课，这样省下来的时间说不定能多上几门课呢，最后返回优先队列中元素的个数就是能完成的课程总数啦**，参见代码如下：

<https://leetcode.com/problems/course-schedule-iii/solutions/104845/short-java-code-using-priorityqueue/>

public class Solution {

    public int scheduleCourse(int[][] courses) {

        Arrays.sort(courses,(a,b)->a[1]-b[1]); //Sort the courses by their deadlines (Greedy! We have to deal with courses with early deadlines first)

        PriorityQueue<Integer> pq=new PriorityQueue<>((a,b)->b-a);

        int time=0;

        for (int[] c:courses)

        {

            time+=c[0]; // add current course to a priority queue

            pq.add(c[0]);

            if (time>c[1]) time-=pq.poll(); //If time exceeds, drop the previous course which costs the most time. (That must be the best choice!)

        }

        return pq.size();

    }

}

**Complexity Analysis**

Time complexity : O(nlog⁡n). At most n elements are added to the queue. Adding each element is followed by heapification, which takes O(log⁡n) time.

Space complexity : O(n). The queue containing the durations of the courses taken can have at most n elements