You are given a series of video clips from a sporting event that lasted time seconds. These video clips can be overlapping with each other and have varying lengths.

Each video clip is described by an array clips where clips[i] = [starti, endi] indicates that the ith clip started at starti and ended at endi.

We can cut these clips into segments freely.

* For example, a clip [0, 7]  [0, 1] + [1, 3] + [3, 7]

Return **the minimum number of clips needed so that we can cut the clips into segments that cover the entire sporting event** [0, time]. If the task is impossible, return -1.

**Example 1:**

**Input:** clips = [[0,2],[4,6],[8,10],[1,9],[1,5],[5,9]], time = 10

**Output:** 3

**Explanation:** We take the clips [0,2], [8,10], [1,9]; a total of 3 clips.Then, we can reconstruct the sporting event as follows:We cut [1,9] into segments [1,2] + [2,8] + [8,9].Now we have segments [0,2] + [2,8] + [8,10] which cover the sporting event [0, 10].

**Example 2:**

**Input:** clips = [[0,1],[1,2]], time = 5

**Output:** -1

**Explanation:** We cannot cover [0,5] with only [0,1] and [1,2].

**Example 3:**

**Input:** clips = [[0,1],[6,8],[0,2],[5,6],[0,4],[0,3],[6,7],[1,3],[4,7],[1,4],[2,5],[2,6],[3,4],[4,5],[5,7],[6,9]], time = 9

**Output:** 3

**Explanation:** We can take clips [0,4], [4,7], and [6,9].

**Constraints:**

* 1 <= clips.length <= 100
* 0 <= starti <= endi <= 100
* 1 <= time <= 100

**Attempt 1: 2023-12-13**

**Solution 1: Native DFS (60 min,TLE 45/60)**

**这道题使用的是L45. Jump Game II的全套思路，唯一的tricky point是在构建jump array的时候需要把clip array按照L45中的定义转化**

**Based on L45.Jump Game II create a jump position array**

**Recall the definition in L45 and below logic is how L1024 translate into L45:**

**In L45: Each element nums[i] represents the maximum length of a forward jump from index i. In other words, if you are at nums[i], you can jump to any nums[i + j]**

**In L1024: Each video clip is described by an array clips where clips[i] = [starti, endi] indicates that the ith clip started at starti and ended at endi.**

**Now we transform L1024 definition of clip[start, end] into**

**clip[start] => nums[i]**

**clip[end] => nums[j]**

**The jump range we will store at clip[start] is maximum value between existing 'jump[clip[start]]' and 'clip[end] - clip[start]', then we can exactly create a jump position array based on each element in clips array, each element in jump position array means: At clip[start] we can jump to max(jump[clip[start]], clip[end] - clip[start])**

class Solution {

    public int videoStitching(int[][] clips, int time) {

        // Based on L45.Jump Game II create a jump position array

        // Recall the definition in L45 and below logic is how L1024

        // translate into L45:

        // In L45: Each element nums[i] represents the maximum length

        // of a forward jump from index i. In other words, if you are

        // at nums[i], you can jump to any nums[i + j]

        // In L1024: Each video clip is described by an array clips

        // where clips[i] = [starti, endi] indicates that the ith clip

        // started at starti and ended at endi.

        // Now we transform L1024 definition of clip[start, end] into

        // clip[start] => nums[i]

        // clip[end] => nums[j]

        // The jump range we will store at clip[start] is maximum value

        // between existing 'jump[clip[start]]' and 'clip[end] - clip[start]',

        // then we can exactly create a jump position array based on each

        // element in clips array, each element in jump position array

        // means: At clip[start] we can jump to max(jump[clip[start]],

        // clip[end] - clip[start])

        int[] jump = new int[time + 1];

        for(int[] clip : clips) {

            if(clip[0] < time) {

                jump[clip[0]] = Math.max(jump[clip[0]], clip[1] - clip[0]);

            }

        }

        int result = helper(jump, 0);

        return result == 102 ? -1 : result;

    }

    // Return remain steps required to get last position of array

    private int helper(int[] nums, int index) {

        // Able to reach, no more steps required

        if(index == nums.length - 1) {

            return 0;

        }

        // Not able to reach, infinite steps remained

        // Note: actually not required, since minimum steps maximum value

        // set as 101, even return because of not approach last position

        // but already stuck on certain index won't increase the maximum value

        if(nums[index] == 0) {

            // In definition "1 <= time <= 100", nums.length max is 101 to

            // include 0 as [0, 100], so return 101 + 1 = 102 as max steps

            return 102;

        }

        int min\_steps = 102;

        // I can make jumps ranging from index + 1, till index + nums[index],

        // and hence will run a loop to cover all those possbile jumps

        for(int i = index + 1; i <= index + nums[index]; i++) {

            if(i < nums.length) {

                min\_steps = Math.min(min\_steps, 1 + helper(nums, i));

            }

        }

        return min\_steps;

    }

}

Time Complexity: O(N!)

At each index i we have N-i choices and we recursively explore each of them till end. So we require O(N\*(N-1)\*(N-2)...1) = O(N!).

Space Complexity: O(N)

**Solution 2: Top Down DP (Memoization) (10 min)**

class Solution {

    public int videoStitching(int[][] clips, int time) {

        // Based on L45.Jump Game II create a jump position array

        // Recall the definition in L45 and below logic is how L1024

        // translate into L45:

        // In L45: Each element nums[i] represents the maximum length

        // of a forward jump from index i. In other words, if you are

        // at nums[i], you can jump to any nums[i + j]

        // In L1024: Each video clip is described by an array clips

        // where clips[i] = [starti, endi] indicates that the ith clip

        // started at starti and ended at endi.

        // Now we transform L1024 definition of clip[start, end] into

        // clip[start] => nums[i]

        // clip[end] => nums[j]

        // The jump range we will store at clip[start] is maximum value

        // between existing 'jump[clip[start]]' and 'clip[end] - clip[start]',

        // then we can exactly create a jump position array based on each

        // element in clips array, each element in jump position array

        // means: At clip[start] we can jump to max(jump[clip[start]],

        // clip[end] - clip[start])

        int[] jump = new int[time + 1];

        for(int[] clip : clips) {

            if(clip[0] < time) {

                jump[clip[0]] = Math.max(jump[clip[0]], clip[1] - clip[0]);

            }

        }

        Integer[] memo = new Integer[jump.length];

        int result = helper(jump, 0, memo);

        return result == 102 ? -1 : result;

    }

    // Return remain steps required to get last position of array

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

        // Able to reach, no more steps required

        if(index == nums.length - 1) {

            return 0;

        }

        // Not able to reach, infinite steps remained

        // Note: actually not required, since minimum steps maximum value

        // set as 101, even return because of not approach last position

        // but already stuck on certain index won't increase the maximum value

        if(nums[index] == 0) {

            // In definition "1 <= time <= 100", nums.length max is 101 to

            // include 0 as [0, 100], so return 101 + 1 = 102 as max steps

            return 102;

        }

        if(memo[index] != null) {

            return memo[index];

        }

        int min\_steps = 102;

        // I can make jumps ranging from index + 1, till index + nums[index],

        // and hence will run a loop to cover all those possbile jumps

        for(int i = index + 1; i <= index + nums[index]; i++) {

            if(i < nums.length) {

                min\_steps = Math.min(min\_steps, 1 + helper(nums, i, memo));

            }

        }

        return memo[index] = min\_steps;

    }

}

Time Complexity : O(N^2)

Space Complexity: O(N)

**Solution 3: Bottom Up DP (10 min)**

**Style 1: Traverse backward**

class Solution {

    public int videoStitching(int[][] clips, int time) {

        int[] jump = new int[time + 1];

        for(int[] clip : clips) {

            if(clip[0] < time) {

                jump[clip[0]] = Math.max(jump[clip[0]], clip[1] - clip[0]);

            }

        }

        // dp[i] means minimum steps can reach last position from index 'i'

        int n = jump.length;

        int[] dp = new int[n];

        Arrays.fill(dp, 102);

        // Since no step required for last position, minimum step as 0

        dp[n - 1] = 0;

        for(int i = n - 2; i >= 0; i--) {

            for(int j = 1; j <= jump[i]; j++) {

                if(i + j < n) {

                    dp[i] = Math.min(dp[i], 1 + dp[i + j]);

                }

            }

        }

        return dp[0] == 102 ? -1 : dp[0];

    }

}

Time Complexity : O(N^2)

Space Complexity: O(N)

**Style 2: Traverse forward**

class Solution {

    public int videoStitching(int[][] clips, int time) {

        int[] jump = new int[time + 1];

        for(int[] clip : clips) {

            if(clip[0] < time) {

                jump[clip[0]] = Math.max(jump[clip[0]], clip[1] - clip[0]);

            }

        }

        // dp[i] means minimum steps can reach last position from index 'i'

        int n = jump.length;

        int[] dp = new int[n];

        Arrays.fill(dp, 102);

        // Since no step required for last position, minimum step as 0

        dp[0] = 0;

        for(int i = 0; i <= n - 2; i++) {

            for(int j = 1; j <= jump[i]; j++) {

                if(i + j < n) {

                    dp[i + j] = Math.min(dp[i + j], 1 + dp[i]);

                }

            }

        }

        return dp[n - 1] == 102 ? -1 : dp[n - 1];

    }

}

Time Complexity : O(N^2)

Space Complexity: O(N)

**Solution 3: Another style DP (30 min)**

class Solution {

    public int videoStitching(int[][] clips, int time) {

        int[] dp = new int[time + 1];

        Arrays.fill(dp, time + 1);

        dp[0] = 0;

        for(int i = 1; i <= time; i++) {

            for(int[] clip : clips) {

                // When clip includes timestamp 'i', means

                // clip[0] can reach 'i' with one more step

                if(clip[0] <= i && i <= clip[1]) {

                    dp[i] = Math.min(dp[i], dp[clip[0]] + 1);

                }

            }

        }

        return dp[time] == time + 1 ? -1 : dp[time];

    }

}

Time Complexity: O(NT)

Space Complexity: O(T)

**Refer to**

<https://www.cnblogs.com/grandyang/p/14395758.html>

下面这种也是 DP 解法，不过并不用给片段排序，因为 dp 的更新方法不同，定义还是跟上面相同，不过这里就可以定义为 T+1 的大小，且均初始化为 T+1，除了 dp[0] 要赋值为0。然后此时是从1遍历到T，对于每个时间点，遍历所有的片段，假如当前时间点i在该片段中间，则用 dp[clip[0]]+1 来更新 dp[i]，注意和上面解法的不同之处，参见代码如下：

class Solution {

public:

    int videoStitching(vector<vector<int>>& clips, int T) {

        vector<int> dp(T + 1, T + 1);

        dp[0] = 0;

        for (int i = 1; i <= T; ++i) {

            for (auto &clip : clips) {

                if (i >= clip[0] && i <= clip[1]) {

                    dp[i] = min(dp[i], dp[clip[0]] + 1);

                }

            }

        }

        return dp[T] == T + 1 ? -1 : dp[T];

    }

};

**Refer to**

<https://leetcode.com/problems/video-stitching/solutions/270036/java-c-python-greedy-solution-o-1-space/>

Loop on i form 0 to T, loop on all clips, if clip[0] <= i <= clip[1], we update dp[i]

Time O(NT), Space O(T)

    public int videoStitching(int[][] clips, int T) {

        int[] dp = new int[T + 1];

        Arrays.fill(dp, T + 1);

        dp[0] = 0;

        for (int i = 1; i <= T && dp[i - 1] < T; i++) {

            for (int[] c : clips) {

                if (c[0] <= i && i <= c[1])

                    dp[i] = Math.min(dp[i], dp[c[0]] + 1);

            }

        }

        return dp[T] == T + 1 ? -1 : dp[T];

    }