1024. Video Stitching

Medium Greedy Array Dynamic Programming Leetcode Link

Problem Description

You are tasked with the challenge of creating a highlight reel from a series of overlapping video clips taken during a sporting event. The entire event has a duration of `time` seconds. The video clips are provided in the form of a list, where each clip is represented by a pair of numbers: the start time (`start_i`) and the end time (`end_i`). The clips may overlap and can have different lengths. Interestingly, you have the freedom to cut these clips into any number of segments you wish.

The main objective is to find the smallest number of clips necessary to cover the entire duration of the event from 0 to `time` seconds. If it is not possible to cover the entire event with the given clips, the function should return -1.

Intuition

To solve this problem, we first need to think about how to use the given clips to cover the entire time span of the event. A brute force approach might try to consider all possible combinations of clips, but this would be highly inefficient. Instead, we need a smarter strategy that can quickly determine the optimal set of clips without redundant computations.

The intuition behind the solution is to first figure out, for each second in the event's duration, what is the furthest point we can reach if we were to start a clip at that second. We begin by initializing an array that will hold this information. As we iterate through each clip, we record the farthest end time we can achieve, starting from each start time that is less than `time`.

Next, we iterate through each second in the event, extending our reach as we find greater end times. The goal is to keep extending our current segment's end time (`pre`) until we can no longer do so or until we reach the end of the event. If at any point, the farthest reach (`mx`) is less than or equal to the current second, it means there is a gap in the coverage and hence the task is impossible.

On the other hand, when the current second reaches the previous maximum reach (`pre`), it indicates that we have utilized a clip to its fullest extent, and we need to select another clip to extend our coverage. This selection increments our answer to the minimum number of clips needed. Eventually, if we are able to cover the whole event, we return the number of clips used; otherwise, if we encounter a coverage gap, we return -1.

This strategy is efficient because it takes advantage of the greedy approach, always picking the clip that extends coverage the furthest at each step, and ensuring that a minimum number of clips are used. The algorithm will sweep through the time units only once, maintaining the maximum reach and updating the number of clips used as needed.

Solution Approach

The solution involves a greedy algorithm that aims to extend the reach of the current clip segment as far as possible within the given time. Let's walk through the implementation:

- First, we initialize a list called `last` with a length equal to the `time`. This list is used to track the furthest end time (`last[a]`) that can be reached from each starting second `a`.
- The solution iterates over each clip `[a, b]` from the provided `clips` list. For each clip, it checks if the start time `a` is within the event duration (`a < time`). If so, it updates `last[a]` with the maximum of its current value and the end time `b` of the current clip. This step effectively finds the furthest end time achievable from each start time.

```python
last = [0] * time
for a, b in clips:
    if a < time:
        last[a] = max(last[a], b)
```

- We then initialize three variables: `ans` to keep track of the number of clips used, `mx` to store the furthest end time reachable at any point, and `pre` which represents the end of the current clip segment we are creating.
- The next loop goes through each second (index `i`) in the `last` list, updating `mx` to the maximum of its current value and `last[i]` which represents the furthest time we can reach from second `i`. Here are two important conditions:
  - If at any point `mx` (the maximum reach so far) is less than or equal to the current second `i`, this means that there is a gap in our clip coverage and we cannot create a continuous highlight reel up to this second. Hence, we return `-1`.
  - If the current second `i` equals `pre` (the previous segment's farthest reach), it means that we need to start a new segment by selecting another clip. We increment `ans` (the number of clips used), and update `pre` to `mx` to reflect the new segment's farthest reach.

```python
ans = mx = pre = 0
for i, v in enumerate(last):
    mx = max(mx, v)
    if mx <= i:
        return -1
    if pre == i:
        ans += 1
        pre = mx
```

- After the loop is completed, if we are able to reach the end of the event's duration (`time`), the variable `ans` will hold the minimum number of clips needed. The function returns this value.

This approach uses a greedy algorithm for selecting clips, as well as dynamic programming techniques to efficiently calculate the maximum coverage at each point in time. By continuously updating the furthest reach and tracking the number of clips, the algorithm ensures that it always progresses towards the goal and only uses the necessary clips to cover the entire event duration.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have an event that lasts for 10 seconds (`time = 10`), and we have the following video clips provided in the form of `(start_i, end_i)` pairs:

```python
Clips = [(0, 3), (1, 5), (4, 8), (7, 10)]
```

Following the solution approach:

1. Initialize the `last` list with a length equal to `time` (which is 10 in this case), to track the furthest end time we can reach from each starting second.

   ```python
   last = [0]*10  # [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
   ```

2. Iterate through each clip to update the `last` list.
   - For clip `(0, 3)`, since `0 < 10`, update `last[0]` to 3.
   - For clip `(1, 5)`, since `1 < 10`, update `last[1]` to 5.
   - For clip `(4, 8)`, since `4 < 10`, update `last[4]` to 8.
   - For clip `(7, 10)`, since `7 < 10`, update `last[7]` to 10.

   After iterating, `last` looks like this:

   ```python
   last = [3, 5, 0, 0, 8, 0, 0, 10, 0, 0]
   ```

3. Initialize `ans` (the number of clips used), `mx` (the farthest end time reachable at any point), and `pre` (the end of the current clip segment) to 0.
4. Iterate through the `last` list to find the minimum number of clips:
   - For `i = 0`, `mx` becomes max(0, 3), so `mx = 3`.
   - For `i = 1`, `mx` becomes max(3, 5), so `mx = 5`.
   - Since `i` has not reached `pre` yet, we continue without incrementing `ans`.
   - For `i = 2`, `mx` remains 5 as `last[2]` is 0 and `5 > 2`.
   - ...
   - When `i = 3` and `pre = 3`, we find our first segment covering from second 0 to second 5. Increment `ans` to 1, and update `pre` to `mx` (5).
   - Continue this process until `i=7`.
   - For `i = 7`, `mx` becomes max(5, 10), so `mx = 10`.
   - `i = 7` is equal to `pre` which is 5, so increment `ans` to 2, and update `pre` to `mx` (10).
   - We now have covered the duration from 0 to 10 seconds with 2 segments.
5. Finally, since the `last` list has been fully traversed and we've been able to cover every second up to the end of the event (`10`), we conclude that 2 is the minimum number of clips needed to cover the event.

Following this example, the function would return `2` since we can cover the entire event with the [(1, 5), (7, 10)] clips.

Python Solution

```python
from typing import List

class Solution:
    def videoStitching(self, clips: List[List[int]], target_time: int) -> int:
        # Initialize a list to keep track of the furthest point each second can reach
        furthest_reach = [0] * target_time

        # Pre-process the clips to fill in the furthest_reach list
        for start, end in clips:
            # Only consider clips that start before the target_time
            if start < target_time:
                # Update the furthest_reach for the second that this clip starts
                furthest_reach[start] = max(furthest_reach[start], end)

        clips_required = 0  # Counter for the minimum number of clips required
        current_end = 0     # The furthest point we can reach without adding another clip
        next_end = 0        # The furthest point we can reach by including another clip

        # Iterate through each second up to the target_time
        for second in range(target_time):
            # Determine the most distant point in time we can reach from this second
            next_end = max(next_end, furthest_reach[second])

            # If the next_end is less or equal to the current second, it means we have a gap.
            # We can't reach the current second from any previous clips.
            if next_end <= second:
                return -1

            # When the current second reaches the current_end, we need to select a new clip
            if current_end == second:
                clips_required += 1
                # Update current_end to the furthest point reachable from this clip
                current_end = next_end

        return clips_required

# Example Usage
solution = Solution()
clips = [[0,2], [4,6], [8,10], [1,9], [1,5], [5,9]]
target_time = 10
print(solution.videoStitching(clips, target_time))  # Outputs the minimum number of clips needed

```

Java Solution

```java
class Solution {
    public int videoStitching(int[][] clips, int T) {
        int[] maxReach = new int[T];

        // Iterate over each clip and record the furthest end time for each start time
        for (int[] clip : clips) {
            int start = clip[0], end = clip[1];
            if (start < T) {
                maxReach[start] = Math.max(maxReach[start], end);
            }
        }

        int count = 0; // the minimum number of clips needed
        int maxEnd = 0; // the farthest end time we can reach so far
        int prevEnd = 0; // the end time of the last clip we have included in the solution

        // Loop through each time unit up to T
        for (int i = 0; i < T; i++) {
            maxEnd = Math.max(maxEnd, maxReach[i]);

            // If the maxEnd we can reach is less or equal to current time 'i',
            // it is impossible to stitch the video up to 'i'
            if (maxEnd <= i) {
                return -1;
            }

            // When we reach the end of the previous clip, increment the count of clips
            // and set the prevEnd to maxEnd to try and reach further in the next iteration
            if (prevEnd == i) {
                count++;
                prevEnd = maxEnd;
            }
        }

        // Return the minimum number of clips needed
        return count;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int videoStitching(vector<vector<int>>& clips, int time) {
        // lastSeen represents the farthest time we can get to starting from each second
        vector<int> lastSeen(time, 0);

        // Iterate through the clips and fill up the lastSeen vector with the farthest end time
        // for clips that start at or before 'i' and are relevant to our goal (i.e., within 'time').
        for (auto& clip : clips) {
            int start = clip[0], end = clip[1];
            if (start < time) {
                lastSeen[start] = max(lastSeen[start], end);
            }
        }

        // 'maxReach' holds the farthest time we can reach at any moment
        int maxReach = 0;
        // 'answer' is the minimum number of clips needed to cover [0, time)
        int answer = 0;
        // 'previous' holds the end time of the last selected clip
        int previous = 0;

        // Iterate from 0 to time to determine if we can reach 'time' and what minimum number of clips are needed
        for (int i = 0; i < time; ++i) {
            // Extend the maxReach if the current second allows for a farther reach
            maxReach = max(maxReach, lastSeen[i]);

            // If maxReach is less or equal to 'i', it means we can't move past this second, thus return -1
            if (maxReach <= i) {
                return -1;
            }

            // If 'i' equals 'previous', it means we've used one clip and need to select the next
            // The selection is based on how far we can reach from here (maxReach)
            if (previous == i) {
                ++answer; // Increment the number of clips used
                previous = maxReach; // Update 'previous' to be the farthest time we can reach now
            }
        }

        // Return the minimum number of clips needed
        return answer;
    }
};

```

Typescript Solution

```typescript
// Defines the type for a video clip, which is an array of 2 numbers
type Clip = [number, number];

// Initializes an array to represent the farthest time reachable starting from each second
let lastSeen: number[] = [];

// Calculates the minimum number of clips required to cover a range from 0 to time
// clips: an array of video clips defined by their start and end times
// time: the total time that needs to be covered by the clips
function videoStitching(clips: Clip[], time: number): number {
    // Reset the lastSeen array for the current computation
    lastSeen = Array(time).fill(0);

    // Populate the lastSeen array with the furthest end time for relevant clips
    for (let clip of clips) {
        let [start, end] = clip;
        if (start < time) {
            lastSeen[start] = Math.max(lastSeen[start], end);
        }
    }

    // Variables to track the farthest time we can reach and the minimum clips needed
    let maxReach = 0;
    let answer = 0;
    let previousReach = 0;

    // Iterate through each second until time to compute the minimum number of clips required
    for (let i = 0; i < time; i++) {
        // Extend the maxReach with the furthest time we can achieve from this second
        maxReach = Math.max(maxReach, lastSeen[i]);

        // If we cannot progress beyond this second, return -1 to indicate it's impossible
        if (maxReach <= i) {
            return -1;
        }

        // If we've reached the end of the current clip, select the next clip
        if (previousReach == i) {
            answer++;
            previousReach = maxReach; // Update our reach to the furthest end time encountered
        }
    }

    // Return the calculated minimum number of clips needed
    return answer;
}

```

Time and Space Complexity

Time Complexity

The provided Python code has a for loop that iterates through the `clips` list, which has a length that we can refer to as `n`. Inside this loop, we have constant time operations (`if` comparison and `max` function), meaning that this part of the algorithm has a time complexity of `O(n)`.

Following that, there is another for loop which iterates through the `last` list. Since `last` has a length equal to the `time` parameter, this loop iterates `time` times. Once again, we perform constant time operations within the loop (`max` function, `if` comparisons, and simple assignments).

Therefore, the second loop has a time complexity of `O(time)`. Since these two loops are not nested but executed in sequence, the overall time complexity of the code combines both complexities, resulting in `O(n + time)`.

Space Complexity

For space complexity, the code allocates an array `last` of length equal to `time`, which requires `O(time)` space. The rest of the variables used in the code (`ans`, `mx`, and `pre`) require constant space, `O(1)`.

Therefore, the overall space complexity of the function is `O(time)` because this is the largest space requirement that does not change with different inputs of `clips`.