2271. Maximum White Tiles Covered by a Carpet

Medium Greedy Array Binary Search Prefix Sum Sorting Leetcode Link

Problem Description

In this problem, we are provided with a list of intervals, where each interval represents a set of consecutive white tiles on the floor. Each interval is given as a pair, with the first number representing the starting position of the sequence of white tiles, and the second number representing the ending position of this sequence.

Additionally, we have been given the length of a carpet (`carpetLen`) that we can use to cover a continuous stretch of white tiles. Our goal is to strategically place this carpet such that it covers the maximum number of white tiles possible.

To summarize, we must find the optimal position for our carpet to maximize the number of covered white tiles and return this maximum number.

Intuition

The intuition behind the solution is to use a sliding window algorithm. A sliding window is a subset of data that moves through a set of elements, which is useful when you have an array or list and you want to consider a subarray/sublist of a specific length (in our case, the length is defined by `carpetLen`).

1. First, we sort the intervals based on their starting positions. This allows us to process the tiles sequentially and determine the carpets' positions more efficiently.
2. We initialize two pointers, `i` and `j`, to traverse the intervals. The pointer `i` denotes the beginning of the window while `j` is used to find the extent to which the carpet can cover tiles (tiles fully covered by the carpet without exceeding its length).
3. We iterate over the tiles with pointer `i`. For every new position of `i`, we proceed with pointer `j` to extend the window's reach - this includes adding on the number of tiles that are within the range that the carpet could potentially cover.
4. Since the carpet can partially cover an interval, the check `if j < n and li + carpetLen > tiles[j][0]` is used to add the remaining portion of white tiles not completely covered by the interval at `j`, up to the length of the carpet.
5. We store the maximum number of covered tiles in a variable `ans`, updating it with the maximum between its current value and the number of covered tiles (including partial coverage on the last interval if necessary).
6. As we move our window (incrementing the pointer `i`), we subtract the number of tiles that are no longer included in the carpet's coverage from our current sum `s` and continue to find the optimal number of covered tiles until all intervals have been considered.

This process allows us to effectively find the maximum white tiles that the carpet can cover by shifting our sliding window across the sorted tile intervals and accounting for both fully and partially covered intervals.

Solution Approach

The solution provided employs a sliding window technique, coupled with sorting and careful addition and subtraction of the tiles within each range. Let's break down the implementation step-by-step:

1. **Sorting Intervals**: The code first sorts the `tiles` list by the starting position of each interval. This sequential order is crucial for the sliding window technique to work efficiently because it ensures that we're always looking at contiguous sections of floor tiles.

   ```
   tiles.sort()
   ```

2. **Initializing Variables**: The function initializes `n` as the length of the `tiles` list, `s` as the current count of white tiles covered by the carpet (initially set to 0), `ans` as the answer to be returned (also starting at 0), and `j` as the second pointer that will stretch the window to its limits.

   ```
   n = len(tiles)
   s = ans = j = 0
   ```

3. **Sliding Window**: The primary loop iterates over the sorted intervals using the index and the interval itself (`(i, (li, ri))`):

   - It uses a `while` loop to extend pointer `j` as far as possible without the carpet length (`carpetLen`) being exceeded, accumulating the count of white tiles that any position of the carpet could cover in `s`.

     ```
     while j < n and tiles[j][1] - li + 1 <= carpetLen:
         s += tiles[j][1] - tiles[j][0] + 1
         j += 1
     ```

   - For partially covered ranges (where the carpet covers only a part of the j-th range), an if condition checks whether the carpet would partially cover the next set of tiles and updates the answer accordingly.

     ```
     if j < n and li + carpetLen > tiles[j][0]:
         ans = max(ans, s + li + carpetLen - tiles[j][0])
     else:
         ans = max(ans, s)
     ```

   - After considering the current position of `i`, the code needs to subtract from `s` the number of tiles in the range of `i` that won't be covered once `i` moves to `i + 1`.

     ```
     s -= ri - li + 1
     ```

4. **Returning the Result**: After the loop terminates, the highest number of covered white tiles recorded in `ans` during the traversal is returned as the solution.

   ```
   return ans
   ```

This approach aggregates the number of tiles that can be covered by the carpet while tracking partially covered ranges correctly. The sliding window advances across the floor, effectively examining each possible placement of the carpet to find the maximum coverage. The sorting step ensures that the sliding window does not move back and forth, which would be inefficient, and instead incrementally adjusts by moving both `i` and `j` forward through the tiles. This yields an optimal solution in terms of execution time and memory usage.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose we have a list of intervals given by white tiles on the floor and a carpet length of 4 tiles:

```
Intervals: [(1, 2), (3, 5), (7, 10)]
CarpetLen: 4
```

The intervals represent sequences of consecutive white tiles. Our carpet can cover 4 consecutive tiles, and we aim to cover as many white tiles as we can.

Following the steps outlined in the solution:

1. **Sorting Intervals**: We sort the intervals by their starting positions, although our example list is already sorted.
2. **Initializing Variables**: We set:
   - `n = 3` (length of the intervals list)
   - `s = 0` (current sum of covered tiles)
   - `ans = 0` (answer to return, the maximum number of tiles that can be covered)
   - `j = 0` (second pointer to extend the window)
3. **Sliding Window**: We iterate through each interval.
   - For `i = 0`: interval (1, 2). We see if the carpet can cover interval `(1, 2)` entirely and reach into the next one.
     - Carpet can cover `(1, 2)` entirely, `s = 2` now.
     - Then we check if it can cover `(3, 5)` without exceeding the length. It can cover until `4` (since the carpet is 4 tiles long), so `s += 2`, resulting in `s = 4`.
     - Carpet cannot entirely cover the next interval `(7, 10)` without exceeding its length, so `ans = max(ans, s) = 4`.
   - Before moving to the next element, subtract the number of tiles in range `(1, 2)` as it won't be covered in the next iteration. So, `s -= 2`.
   - For `i = 1`: interval (3, 5). We see if the carpet can cover interval `(3, 5)` entirely and reach into the next one.
     - Carpet can cover `(3, 5)` entirely, `s = 3` now.
     - It can partially cover `(7, 10)`, up to `7`, so we add `2` to `s` for the partial coverage, resulting in `s = 5`.
     - `ans = max(ans, s) = 5` for this iteration.
   - We don't perform a subtraction step here, as there are no more intervals for the carpet to slide over.
4. **Returning the Result**: The loop has terminated, and the highest number of covered white tiles recorded is `ans = 5`, which is the solution. We return `ans`.

```
return ans # returns 5 as the maximum number of covered tiles
```

This example demonstrates that by sorting the intervals and using a sliding window approach, we can efficiently determine that the carpet can cover a maximum of 5 white tiles when placed starting from tile 3 to tile 6.

Python Solution

```python
class Solution:
    def maximumWhiteTiles(self, tiles: List[List[int]], carpet_length: int) -> int:
        # Sort the tile intervals by the starting point of each tile.
        tiles.sort()

        # Initialize useful variables.
        num_tiles = len(tiles)
        covered = max_covered = pointer = 0

        # Iterate through the sorted tiles.
        for index, (start, end) in enumerate(tiles):
            # Extending the carpet to cover as many tiles as possible.
            while pointer < num_tiles and tiles[pointer][1] - start + 1 <= carpet_length:
                # Add the number of tiles covered by the current part of the carpet.
                covered += tiles[pointer][1] - tiles[pointer][0] + 1
                # Move the pointer to see if more tiles can be covered with the remaining carpet length.
                pointer += 1

            # Checking whether extending the carpet partially covers a tile.
            if pointer < num_tiles and start + carpet_length > tiles[pointer][0]:
                # Update the max_covered by adding partial coverage of the next tile.
                max_covered = max(max_covered, covered + start + carpet_length - tiles[pointer][0])
            else:
                # Else the carpet lies within a contiguous stretch of white tiles.
                max_covered = max(max_covered, covered)

            # Subtract the number of tiles covered by the current start tile
            # as we move the start to the next tile in the subsequent iteration.
            covered -= end - start + 1

        # Return the maximum number of white tiles that can be covered by the carpet.
        return max_covered

```

Java Solution

```java
import java.util.Arrays;

class Solution {
    // Method to calculate the maximum number of white tiles that can be covered by the carpet.
    public int maximumWhiteTiles(int[][] tiles, int carpetLen) {
        // Sort the tiles array based on the left edge of each tile.
        Arrays.sort(tiles, (a, b) -> a[0] - b[0]);

        // Initialize the total number of tiles and the answer (maximum tiles covered).
        int numberOfTiles = tiles.length;
        int totalCovered = 0, maxTilesCovered = 0;

        // Use two pointers to navigate through the tiles array.
        for (int leftIndex = 0, rightIndex = 0; leftIndex < numberOfTiles; ++leftIndex) {
            // Slide the right pointer to the right while the carpet can cover the new tile entirely.
            while (rightIndex < numberOfTiles && tiles[rightIndex][1] - tiles[leftIndex][0] + 1 <= carpetLen) {
                totalCovered += tiles[rightIndex][1] - tiles[rightIndex][0] + 1;
                ++rightIndex;
            }

            // Check if the carpet partially covers the next tile.
            if (rightIndex < numberOfTiles && tiles[leftIndex][0] + carpetLen > tiles[rightIndex][0]) {
                maxTilesCovered = Math.max(maxTilesCovered, totalCovered + tiles[leftIndex][0] + carpetLen - tiles[rightIndex][0]);
            } else {
                // Carpet does not reach the next tile, just consider current total coverage.
                maxTilesCovered = Math.max(maxTilesCovered, totalCovered);
            }

            // Move the left pointer to the right and decrease the total coverage accordingly.
            totalCovered -= (tiles[leftIndex][1] - tiles[leftIndex][0] + 1);
        }

        // Return the maximum number of tiles that the carpet can cover.
        return maxTilesCovered;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    int maximumWhiteTiles(vector<vector<int>>& tiles, int carpetLength) {
        // Sort the tiles based on their starting position.
        sort(tiles.begin(), tiles.end());

        int covered = 0, maxCovered = 0, numTiles = tiles.size();

        // Two pointers approach: `left` tracks the start of the carpet, `right` the end.
        for (int left = 0, right = 0; left < numTiles; ++left) {
            // Extend the right boundary of the carpet as far as possible within carpet length.
            while (right < numTiles && tiles[right][1] - tiles[left][0] + 1 <= carpetLength) {
                covered += tiles[right][1] - tiles[right][0] + 1;
                ++right;
            }

            // If the carpet can't extend to cover the whole next tile, include partial coverage.
            if (right < numTiles && tiles[left][0] + carpetLength > tiles[right][0]) {
                maxCovered = std::max(maxCovered, covered + tiles[left][0] + carpetLength - tiles[right][0]);
            } else {
                // Otherwise, compare the current covered area to the max covered so far.
                maxCovered = std::max(maxCovered, covered);
            }

            // Move the left boundary of the carpet, reducing the covered area accordingly.
            covered -= (tiles[left][1] - tiles[left][0] + 1);
        }

        // Return the maximum white tiles space that can be covered by the carpet.
        return maxCovered;
    }
};

```

Typescript Solution

```typescript
function maximumWhiteTiles(tiles: number[][], carpetLength: number): number {
    // Sort the tiles based on their starting position.
    tiles.sort((a, b) => a[0] - b[0]);

    let covered = 0;
    let maxCovered = 0;
    let numTiles = tiles.length;

    // Two pointers approach: `left` tracks the start of the carpet, `right` the end.
    for (let left = 0, right = 0; left < numTiles; ++left) {
        // Extend the right boundary of the carpet as far as possible within carpet length.
        while (right < numTiles && tiles[right][1] - tiles[left][0] + 1 <= carpetLength) {
            covered += tiles[right][1] - tiles[right][0] + 1;
            ++right;
        }

        // If the carpet can't extend to cover the whole next tile, include partial coverage.
        if (right < numTiles && tiles[left][0] + carpetLength > tiles[right][0]) {
            maxCovered = Math.max(maxCovered, covered + tiles[left][0] + carpetLength - tiles[right][0]);
        } else {
            // Otherwise, compare the current covered area to the max covered so far.
            maxCovered = Math.max(maxCovered, covered);
        }

        // Move the left boundary of the carpet, reducing the covered area accordingly.
        covered -= (tiles[left][1] - tiles[left][0] + 1);
    }

    // Return the maximum white tiles space that can be covered by the carpet.
    return maxCovered;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is primarily determined by the sorting of the `tiles` array and the two-pointer technique used to iterate through the tiles.

1. Sorting the tiles requires `O(n log n)` time, where `n` is the number of tiles.
2. The two-pointer technique involves a while-loop that iterates through the tiles. In the worst case, both `i` and `j` pointers can touch each tile once, resulting in `O(n)` time complexity.

Since these processes are sequential, the overall time complexity of the algorithm is `O(n log n + n)`, which simplifies to `O(n log n)` because the sorting term is dominant.

Space Complexity

The space complexity is `O(1)` or constant space, as the sorting is done in-place (assuming the sort function does in-place sorting) and only a fixed number of integer variables are used to keep track of the pointers, the sum of the tiles within the carpet, and the current maximum.

The auxiliary space for variables like `s`, `ans`, `i`, and `j` is negligible and does not grow with the input size. Thus, the space complexity does not rely on the number of tiles and remains constant.