2201. Count Artifacts That Can Be Extracted

Medium Array Hash Table Simulation Leetcode Link

Problem Description

In the given LeetCode problem, we have a grid of `n x n` size where some artifacts are buried. The grid is such that each cell can be represented by coordinates `(r, c)`, where `r` is the row number and `c` is the column number, with both being 0-indexed. Artifacts are buried in rectangular areas within this grid. Each artifact is defined by two coordinates - the top-left cell `(r1, c1)` and the bottom-right cell `(r2, c2)`. This forms a rectangle in the grid, including the cells on the border of these coordinates.

As part of the excavation process, we dig certain cells in the grid, specified by array `dig`. Each element in `dig` is a pair of coordinates `(r, c)` indicating a cell we have dug up. An artifact is considered to be extracted if all the cells covering it are excavated. The goal is to find out how many artifacts can be completely extracted given the cells we have dug.

Intuition

To solve this problem, the first step is to keep track of all the dug cells in an efficient manner. Since we need to check whether a cell is dug or not, we can use a set data structure (`s`). This set will allow us to quickly determine the presence or absence of each dug cell by using the coordinates as a unique key for each cell.

Once we have this set of dug cells, the next step is to go through each artifact described in the `artifacts` array. For each artifact, we iterate over every cell that it covers, which are the cells bounded by `(r1, c1)` and `(r2, c2)` inclusive. We check if all these cells are present in our set of dug cells.

The `check` function accomplishes this iteration and checking. If at least one cell belonging to an artifact is not present in the dug cells set, it means that the artifact cannot be fully extracted, and thus, we return `False`. If all of the cells are present, we return `True` indicating the artifact can be extracted.

Finally, we use a generator expression to sum up the number of `True` values returned by the `check` function for each artifact, which translates to the number of artifacts that can be extracted.

Solution Approach

The solution follows a simple approach mainly utilizing set data structures and iteration.

1. First, we convert the `dig` list into a set `s` which includes tuples of coordinates `(i, j)` for each cell we dig. The set data structure is chosen for its efficient `O(1)` average time complexity for lookup operations, which is crucial since we need to check if each cell of an artifact has been dug.
2. A helper function `check` is defined which takes an `artifact` as an argument, represented by a list `[r1, c1, r2, c2]`. The function checks if all cells that belong to the artifact have been dug. It does this by iterating over all cells in the rectangular area that the artifact covers—from `r1` to `r2` and `c1` to `c2`—and checking if each cell `(x, y)` is in the set `s`. If any cell is not present, `check` returns `False` because this means the artifact is incomplete and cannot be extracted.
3. If all cells of the artifact are found in the set, `check` returns `True`.
4. The main function `digArtifacts` applies the `check` function to each artifact in the `artifacts` array and sums up the `True` values (logically equivalent to `1`) to obtain the total number of complete artifacts that can be extracted. This uses Python's `sum` function with a generator expression, which is a compact and efficient way to sum a sequence of boolean values.

Here is the relevant section of the code demonstrating the process:

```python
s = {(i, j) for i, j in dig}  # Convert the dig list to a set for efficient lookups
def check(artifact):
    r1, c1, r2, c2 = artifact  # Unpack the coordinates of the artifact
    for x in range(r1, r2 + 1):  # Iterate over the rows
        for y in range(c1, c2 + 1):  # Iterate over the columns
            if (x, y) not in s:  # Check if the cell is not dug
                return False
    return True  # All cells are dug and artifact can be extracted

# Sum up the instances where the artifact can be extracted
return sum(check(v) for v in artifacts)
```

Each step of the approach is designed to cut down on unnecessary computations, which is why we use a set for digs and the `check` function only iterates over cells belonging to the artifacts and not the entire grid. This solution has a time complexity that scales with the number of digs and the size of the artifacts, as opposed to the entire grid size, making it efficient even for larger grid sizes.

Example Walkthrough

Let's consider the following example to illustrate the solution approach:

Suppose we have a grid of size `3 x 3` and there are two artifacts buried:

- Artifact A with coordinates `[0, 0, 1, 1]`, which spans from the top-left corner to the cell in the second row and second column.
- Artifact B with coordinates `[1, 1, 2, 2]`, which spans from the cell in the second row and second column to the bottom-right corner.

The dig array contains the cells that have been dug: `[(0, 0), (0, 1), (1, 1), (2, 2)]`.

To determine how many artifacts can be completely extracted, we will follow the solution approach:

1. Convert the dig array into a set for efficient lookups.

   s = {(0, 0), (0, 1), (1, 1), (2, 2)}

2. Use the `check` function to verify whether all the cells of an artifact have been dug.
3. For Artifact A, the `check` function will look at coordinates:
   - (0, 0), which is in the set `s`
   - (0, 1), which is in the set `s`
   - (1, 0), which is *not* in the set `s`
   - (1, 1), which is in the set `s`

   Since the cell (1, 0) hasn't been dug, Artifact A cannot be fully extracted; hence, `check` returns `False`.
4. For Artifact B, the `check` function will verify the following cells:
   - (1, 1), which is in the set `s`
   - (1, 2), which is *not* in the set `s`
   - (2, 1), which is *not* in the set `s`
   - (2, 2), which is in the set `s`

   Since the cells (1, 2) and (2, 1) haven't been dug, Artifact B cannot be fully extracted; thus, `check` returns `False`.
5. Now, we sum up the `True` values returned by the `check` function for each artifact. Since both artifacts returned `False`, the sum is `0`, meaning no artifacts can be completely extracted based on the cells dug.

By using this solution approach, we are focusing the computation only on the cells that need to be checked, avoiding unnecessary work on other grid cells that do not contain artifacts. This example illustrates how combining set data structures with a focused checking function works efficiently to solve the problem.

Python Solution

```python
from typing import List

class Solution:
    def digArtifacts(self, n: int, artifacts: List[List[int]], digs: List[List[int]]) -> int:
        # Helper function to check if all parts of the artifact have been dug up.
        def is_artifact_fully_dug(artifact):
            # Unpack the corners of the rectangular artifact.
            top_left_row, top_left_col, bottom_right_row, bottom_right_col = artifact

            # Iterate through all positions of the artifact.
            for row in range(top_left_row, bottom_right_row + 1):
                for col in range(top_left_col, bottom_right_col + 1):
                    # Check if the current position has not been dug.
                    if (row, col) not in dug_positions_set:
                        return False  # If any position is undug, artifact is incomplete.
            return True  # All positions dug, artifact is complete.

        # Create a set of dug positions for faster lookup.
        dug_positions_set = {(row, col) for row, col in digs}

        # Count the number of fully dug artifacts using the helper function.
        fully_dug_artifacts_count = sum(is_artifact_fully_dug(artifact) for artifact in artifacts)

        return fully_dug_artifacts_count

# Example usage:
# sol = Solution()
# result = sol.digArtifacts(n=4, artifacts=[[1,0,2,0],[0,2,1,3]], digs=[[0,3],[1,0],[2,1]])
# print(result)  # Expected output: 1, since only the second artifact is fully dug up.

```

Java Solution

```java
class Solution {

    public int digArtifacts(int n, int[][] artifacts, int[][] dig) {
        // Create a set to store the positions that have been dug.
        Set<Integer> dugPositions = new HashSet<>();
        // Convert the 2D dig positions into a 1D format and add to the set.
        for (int[] position : dig) {
            dugPositions.add(position[0] * n + position[1]);
        }
        int numFullyDugArtifacts = 0; // Counter for fully excavated artifacts.

        // Check each artifact to see if all its parts have been dug up.
        for (int[] artifact : artifacts) {
            if (isArtifactFullyDug(artifact, dugPositions, n)) {
                numFullyDugArtifacts++; // Increment the counter if an artifact is fully dug.
            }
        }

        return numFullyDugArtifacts;
    }

    // Helper method to check if all the cells of an artifact have been dug.
    private boolean isArtifactFullyDug(int[] artifact, Set<Integer> dugPositions, int n) {
        int topRow = artifact[0], leftColumn = artifact[1], bottomRow = artifact[2], rightColumn = artifact[3];
        // Iterate over the cells that the artifact spans.
        for (int i = topRow; i <= bottomRow; ++i) {
            for (int j = leftColumn; j <= rightColumn; ++j) {
                // If a cell has not been dug, the artifact is not fully excavated.
                if (!dugPositions.contains(i * n + j)) {
                    return false;
                }
            }
        }
        // If all cells have been dug, the artifact is fully excavated.
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to compute the number of fully dug artifacts
    int digArtifacts(int gridSize, vector<vector<int>>& artifacts, vector<vector<int>>& digs) {
        // Create a hash set to store the dig positions for constant time lookup
        unordered_set<int> dugPositions;
        for (auto& dig : digs) {
            // Convert 2D coordinates to a single integer using the grid size
            dugPositions.insert(dig[0] * gridSize + dig[1]);
        }

        // Variable to store the count of fully dug artifacts
        int fullyDugArtifacts = 0;

        // Check each artifact
        for (auto& artifact : artifacts) {
            // Increment the fully dug artifacts count if the artifact is fully dug
            if (isArtifactFullyDug(artifact, dugPositions, gridSize)) {
                ++fullyDugArtifacts;
            }
        }

        // Return the total count of fully dug artifacts
        return fullyDugArtifacts;
    }

private:
    // Helper function to check if all parts of the artifact have been dug up
    bool isArtifactFullyDug(vector<int>& artifact, unordered_set<int>& dugPositions, int gridSize) {
        // Artifact's top-left and bottom-right coordinates
        int rowStart = artifact[0];
        int colStart = artifact[1];
        int rowEnd = artifact[2];
        int colEnd = artifact[3];

        // Iterate over the grid cells covered by the artifact
        for (int i = rowStart; i <= rowEnd; ++i) {
            for (int j = colStart; j <= colEnd; ++j) {
                // Convert 2D coordinates to a single integer using the grid size
                int position = i * gridSize + j;

                // If any part of the artifact is not dug, return false
                if (!dugPositions.count(position)) {
                    return false;
                }
            }
        }

        // If all parts are dug, the artifact is fully dug, return true
        return true;
    }
};

```

Typescript Solution

```typescript
function digArtifacts(n: number, artifacts: number[][], dig: number[][]): number {
    // Create a 2D array to track visited cells with initial value `false`
    let visitedCells = Array.from({ length: n }, () => new Array(n).fill(false));

    // Mark the cells that have been dug as visited
    for (let [row, col] of dig) {
        visitedCells[row][col] = true;
    }

    // Initialize a counter for completely uncovered artifacts
    let uncoveredArtifactsCount = 0;

    // Check each artifact to see if it is fully uncovered
    for (let [startRow, startCol, endRow, endCol] of artifacts) {
        let isUncovered = true;

        // Iterate over the cells covered by the artifact
        for (let i = startRow; i <= endRow && isUncovered; i++) {
            for (let j = startCol; j <= endCol && isUncovered; j++) {
                // If any cell of the artifact is not visited, mark the artifact as not uncovered
                if (!visitedCells[i][j]) {
                    isUncovered = false;
                }
            }
        }

        // If the artifact is completely uncovered, increment the count
        if (isUncovered) {
            uncoveredArtifactsCount++;
        }
    }

    // Return the total count of completely uncovered artifacts
    return uncoveredArtifactsCount;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed in the following steps:

1. Conversion of `dig` list into a set `s`: This operation iterates over each element in the `dig` list once. Therefore, the time complexity for this part is `O(D)`, where `D` is the length of the `dig` list.
2. The `check` function: This function is called for each artifact. Inside the function, there's a nested loop that iterates over the artifact's area which is at most `(r2 - r1 + 1) * (c2 - c1 + 1)` for each artifact. In the worst case, this could be equal to the size of the grid (`n * n`), if the artifact covers the entire grid.
3. The last line sums up the results of the `check` function for each artifact. Since `check` is invoked once per artifact, the time complexity for calling `check` overall is `O(A * n^2)`, where `A` is the number of artifacts, since we consider the worst case where an artifact spans the entire grid.

Combining these complexities, we have an overall time complexity of `O(D + A * n^2)`.

Space Complexity

The space complexity can be analyzed by looking at the additional space used by the program components:

1. The set `s` that stores the dug positions: In the worst case, it will store all `D` positions where digs happened, yielding a space complexity of `O(D)`.
2. The space used for the `check` function's call stack is negligible as it does not involve recursive calls or additional data structures with size depending on the inputs. However, due to the iterating for each cell in the range of an artifact, at worst case the space required to store the temporary variables `x` and `y` in the stack is constant, `O(1)`.

Overall, since there are no other significant data structures being used, the space complexity is `O(D)`.