174. Dungeon Game

Hard Array Dynamic Programming Matrix

Leetcode Link

Problem Description

The problem presents a scenario where a knight must rescue a princess trapped in the bottom-right corner of a dungeon represented by a 2D grid. The grid has dimensions $m \times n$, with each cell of the grid representing a room that may contain demons, magic orbs, or be empty. The knight begins at the top-left room and can only move rightward or downward.

The knight has an initial health value, which must always remain above 0 for him to survive. If the knight passes through a room with demons, his health decreases by the number depicted in that room; if the room contains magic orbs, his health increases accordingly. The objective is to calculate the minimum health the knight needs to start with to ensure he can reach the princess without dying.

Intuition

The key to solving this problem lies in dynamic programming, particularly in understanding that the optimal decision at each room depends on the decisions made in subsequent rooms. Rather than navigating from the start to the end, we work backwards from the princess's room to the knight's starting point. This way, we can always make sure the knight has just enough health to reach the next step.

We initialize a 2D array (`dp`) that will store the minimum health required to reach the princess from any given cell. The knight's goal is to reach the cell containing the princess with at least 1 health point.

Starting from the princess's cell, we backtrack and calculate the minimum health needed for each previous room. The minimum health required to enter any room is the maximum of 1 (the knight can't have less than 1 health) and the difference between the health required to enter the subsequent room and the value of the current room (which may be positive, negative, or zero).

This computation proceeds until we reach the starting cell, at which point `dp[0][0]` gives us the minimum health that the knight must have to rescue the princess successfully.

Solution Approach

We approach the solution by implementing dynamic programming to solve the problem in a bottom-up manner. Let's go through the implementation using the supplied Python code.

1. **Initialization of the DP table**: We create a 2D array `dp` of size `(m+1)x(n+1)` filled with `inf` (infinity). This represents the minimum health needed to reach the princess from any given cell. We fill it with `inf` to represent that we haven't calculated the health for any room yet. We use `m+1` and `n+1` for ease of implementation so that we can handle the boundary without checking for edge cases in each iteration.
2. **Boundary Conditions**: Set `dp[m - 1][n]` and `dp[m - 1][n]` to 1 because the knight needs a minimum of 1 health point to survive. These are the cells adjacent to the princess's cell. As the knight can only move rightward or downward, these cells represent the only two ways to reach the bottom-right corner without being in it.
3. **Main Loop**: We iterate over the dungeon grid starting from the cell `(m - 1, n - 1)` which is right above and to the left of the princess's cell, moving backward to the start at `(0, 0)`. For each cell `(i, j)`, we calculate the minimum health the knight needs to proceed to either of the next cells `(i + 1, j)` (downwards) or `(i, j + 1)` (rightward).

   Here's the core logic: we use `max(1, min(dp[i + 1][j], dp[i][j + 1]) - dungeon[i][j])`, which means:

   - We first determine the less risky path by finding the minimum of the health needed to move to the next cells (`min(dp[i + 1][j], dp[i][j + 1])`).
   - We then subtract the current room's value from this health. If the current room has a demon (negative value), this effectively increases the health needed to handle the demon. If the room has an orb (positive value), this decreases the required health.
   - However, even if the room's positive value exceeds the health from the next step, the knight cannot have less than 1 health (`max(1, ...)`).
4. **Final Answer**: After filling the DP table, the value in the top-left cell `dp[0][0]` is our answer as it represents the minimum health the knight needs to start with in order to rescue the princess.

Using this approach, we avoid recalculating the minimum health requirement for each cell multiple times, leading to an efficient algorithm with a time complexity of `O(mn)` where `m` and `n` are the dimensions of the dungeon grid.

Example Walkthrough

Let's say the dungeon grid is represented by the following 2D grid, where `m = 2` and `n = 3`:

```
[[-2, -3,  3],
 [-5, -10, 1]]
```

Here's the step-by-step breakdown of the dynamic programming approach:

1. **Initialization of DP Table**: We initialize `dp` with size `(2+1)x(3+1)` and fill it with `inf` except for the boundaries `dp[2][2]` and `dp[2][3]` or `dp3][2]`, which are set to 1.

```
dp = [ [inf, inf, inf, inf],
       [inf, inf, inf, 1  ],
       [inf, inf, 1,   inf]]
```

2. **Boundary Conditions**: The bottom-right corner `(m - 1, n - 1)` is the princess's room, hence `dp[1][2]` will be the maximum of 1 and the inverse of that room's demon value `3` minus `1` (minimum health needed to survive). Since `3` minus `1` is `2`, and we need a maximum of `1`, it stays as `1`:

```
dp = [ [inf, inf, inf, inf],
       [inf, inf, 1,   1  ],
       [inf, inf, 1,   inf]]
```

3. **Main Loop**:
   - We first calculate `dp[1][1]`, which is the maximum of `1` and `min(dp[1 + 1][1], dp[1][1 + 1] - dungeon[1][1]`. The minimum of `dp[2][1]` (inf) and `dp[1][2]` (1) is `1`. The dungeon value at `[1][1]` is `-10`. Thus `1 (-10 + 10 = 0)` becomes the value of `dp[1][1]`.
   - Then we calculate `dp[1][0]` which is the maximum of `1` and `min(dp[1 + 1][0], dp[1][0 + 1] - dungeon[1][0]`. The minimum of `dp[2][0]` (inf) and `dp[1][1]` (1) is `1`. With a dungeon value of `-5`, we get `max(1, 1 - (-5)) = 6`.

```
dp = [ [inf, inf, inf, inf],
       [6,   1,   1,   1  ],
       [inf, inf, 1,   inf]]
```

- For the top row, we repeat the same for `dp[0][2]` and `dp[0][1]`. Finally, `dp[0][0]` is computed, the maximum of `1` and `min(dp[0 + 1][0], dp[0][0 + 1]) - dungeon[0][0]`. The minimum of `dp[1][0]` (6) and `dp[0][1]` (inf) is `6`. Subtracting the dungeon value `-2`, we need `max(1, 6 - (-2)) = 8`.

```
dp = [ [8,   inf, inf, inf],
       [6,   1,   1,   1  ],
       [inf, inf, 1,   inf]]
```

Now, the DP table shows `dp[0][0] = 8`, which means the knight needs at least `8` health points to ensure he can reach the princess without dying.

This simple example illustrates the backward evaluation from the end goal, using dynamic programming to efficiently compute the health needed at each step. The final answer as per the `dp` table shows that the knight needs a minimum starting health of `8`.

Python Solution

```python
from typing import List
from math import inf

class Solution:
    def calculate_minimum_hp(self, dungeon: List[List[int]]) -> int:
        # Get the dimensions of the dungeon matrix.
        num_rows, num_columns = len(dungeon), len(dungeon[0])

        # Initialize dp (Dynamic Programming) matrix with infinity.
        # An extra row and column are added to handle the edge cases easily.
        dp = [[inf] * (num_columns + 1) for _ in range(num_rows + 1)]

        # The hero needs at least 1 HP to reach the princess.
        # Initialize the cell to the princess's right and the one below her to 1.
        dp[num_rows][num_columns - 1] = dp[num_rows - 1][num_columns] = 1

        # Start from the bottom right corner and move to the top left corner.
        for i in range(num_rows - 1, -1, -1):    # Iterate over rows in reverse
            for j in range(num_columns - 1, -1, -1):  # Iterate over columns in reverse

                # Find the minimum HP needed to go to the next cell.
                # Cannot have less than 1 HP, hence the max with 1.
                min_hp_on_exit = min(dp[i + 1][j], dp[i][j + 1]) - dungeon[i][j]
                dp[i][j] = max(1, min_hp_on_exit)

        # Return the HP needed at the start (0,0) to guarantee reaching the princess.
        return dp[0][0]

# Example usage:
# sol = Solution()
# print(sol.calculate_minimum_hp([[-2, -3, 3], [-5, -10, 1], [10, 30, -5]]))
```

Java Solution

```java
class Solution {

    // Function to calculate the minimum initial health needed to reach the bottom-right corner
    public int calculateMinimumHP(int[][] dungeon) {
        // Dimensions of the dungeon
        int rows = dungeon.length;
        int cols = dungeon[0].length;

        // Dynamic programming table where each cell represents the minimum health needed
        int[][] minHealth = new int[rows + 1][cols + 1];

        // Initialize the dp table with high values except the border cells right and below the dungeon
        for (int[] row : minHealth) {
            Arrays.fill(row, Integer.MAX_VALUE);
        }

        // Initialization for the border cells where the hero can reach the princess with 1 health point
        minHealth[rows][cols - 1] = minHealth[rows - 1][cols] = 1;

        // Start from the bottom-right corner of the dungeon and move leftward and upward
        for (int i = rows - 1; i >= 0; --i) {
            for (int j = cols - 1; j >= 0; --j) {
                // The health at the current cell is the minimum health needed from the next step minus the current cell's effect
                // It should be at least 1 for the hero to be alive
                int healthNeeded = Math.min(minHealth[i + 1][j], minHealth[i][j + 1]) - dungeon[i][j];
                minHealth[i][j] = Math.max(1, healthNeeded);
            }
        }

        // The result is the minimum health needed at the starting cell
        return minHealth[0][0];
    }
}
```

C++ Solution

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

using std::vector;
using std::max;
using std::min;
using std::numeric;

class Solution {
public:
    // Function to calculate the minimum health needed to reach the princess (at bottom-right of the dungeon)
    // starting with positive health when moving only rightward or downward.
    int calculateMinimumHP(vector<vector<int>>& dungeon) {
        int m = dungeon.size();     // Number of rows
        int n = dungeon[0].size();  // Number of columns
        int dp[m + 1][n + 1];       // Create DP table of size (m+1)x(n+1)

        // Initialize the dp array with a very large value, as we are looking for minimum health required.
        memset(dp, 0x3f, sizeof dp);

        // Set the health needed at the dungeon's exit. We need at least 1 health at the end.
        dp[m][n - 1] = 1; // If we are at the last cell of the last row
        dp[n - 1][m] = 1; // If we are at the last cell of the last column

        // Loop through the dungeon starting from the bottom right corner, moving to the upper left corner
        for (int i = m - 1; i >= 0; --i) {  // Loop for rows
            for (int j = n - 1; j >= 0; --j) { // Loop for columns
                // The minimum health needed at the start of this cell is 1 or the health we need for the next cell
                // minus the current cell value, whichever is larger. We use 'Math.max' to ensure we don't have non-positive health.
                // We also want the smaller of the two possible paths (rightward or downward) to reach the next cell hence we use `Math
                dp[i][j] = Math.max(1, Math.min(dp[i + 1][j], dp[i][j + 1]) - dungeon[i][j]);
            }
        }

        return dp[0][0];  // The minimum health needed at the start is at the top-left corner of the DP table.
    }
};
```

Typescript Solution

```typescript
// type definition for the 2D dungeon matrix
type Dungeon = number[][];

// Function to calculate the minimum health needed to reach the princess (at bottom-right of the dungeon)
// starting with positive health when moving only rightward or downward.
function calculateMinimumHP(dungeon: Dungeon): number {
    const m: number = dungeon.length;     // Number of rows
    const n: number = dungeon[0].length;  // Number of columns
    let dp: number[][] = Array.from(Array(m + 1), () => new Array(n + 1).fill(0x3f3f3f3f)); // Create DP table filled with large va

    // Set the health needed at the dungeon's exit. We need at least 1 health at the end.
    dp[m][n - 1] = 1; // If at the last cell of the last row
    dp[n - 1][m] = 1; // If at the last cell of the last column

    // Loop through the dungeon starting from the bottom right corner, moving to the upper left corner
    for (let i = m - 1; i >= 0; --i) {  // Loop for rows
        for (let j = n - 1; j >= 0; --j) { // Loop for columns
            // The minimum health needed at the start of this cell is either 1 or the health we need for the next cell
            // minus the current cell value, whichever is larger. We use 'Math.max' to ensure we don't have non-positive health.
            // We also want the smaller of the two possible paths (rightward or downward) to reach the next cell hence we use `Math
            dp[i][j] = Math.max(1, Math.min(dp[i + 1][j], dp[i][j + 1]) - dungeon[i][j]);
        }
    }

    return dp[0][0];  // The minimum health needed at the start is at the top-left corner of the DP table.
};
```

Time and Space Complexity

The given Python code implements a dynamic programming algorithm to calculate the minimum initial health required for a character to navigate a dungeon represented as a 2D grid where some cells contain positive values (health potions) and others contain negative values (traps).

Time Complexity

The time complexity of the algorithm is `O(mn)`, where `m` is the number of rows and `n` is the number of columns in the `dungeon` grid. This is because the algorithm consists of a nested loop structure that iterates over each cell of the `dungeon` grid exactly once.

Space Complexity

The space complexity is also `O(mn)` due to the auxiliary 2D list `dp` that has the same dimensions as the `dungeon` grid plus one extra row and column to handle the boundary cases. This `dp` list is used to store the minimum health required at each cell to reach the bottom right corner of the grid.