568. Maximum Vacation Days

Hard Array Dynamic Programming Matrix

Leetcode Link

Problem Description

LeetCode offers a challenge that simulates planning a travel itinerary to maximize vacation days while adhering to certain constraints. You can visit `n` different cities, each identified by indices from `0` to `n-1`, and have `k` weeks to spend your vacation days. The goal is to choose a travel schedule that will allow you to maximize your total vacation days.

The rules you must keep in mind are:

1. You start in city `0`.
2. Travel is possible between cities based on a flights matrix. A `1` in `flights[i][j]` means you can fly from city `i` to city `j`, while a `0` means no flight is available.
3. You have `k` weeks to travel, and you can only fly on Monday mornings.
4. Each city has a limit on how many vacation days you can spend there each week, described by a matrix `days`.
5. You can work on days that are not counted towards your vacation.
6. Travelling from city `A` to city `B` uses up a vacation day of city `B`.
7. The time spent on flights is not factored into the vacation days.

The task is to take these rules into account and determine the maximum number of vacation days you can enjoy.

Intuition

To solve this problem, we need an approach that can consider all possible travel routes and select the one that yields the maximum vacation days. A dynamic programming solution is well-suited for this problem since it allows us to make decisions at each stage while keeping track of all previously computed states for future reference.

The essential idea behind the solution is to use a two-dimensional array `f` where `f[k][j]` represents the maximum vacation days you can have by the end of week `k` if you are in city `j`. Initialization is critical: `f[0][0]` is set to `0` because the starting point is city `0` with no vacation days used yet.

The dynamic programming process then iterates over each week. For every city `j`, it updates the maximum vacation days by trying to maintain the same city or by flying from some city `i` where a flight is available (`flights[i][j] == 1`). After taking the maximum from these options, the vacation days available in city `j` for that week (`days[j][k - 1]`) are added.

This approach ensures that at any point in the iteration, `f[k][j]` contains the optimum number of vacation days up to that week if you are in city `j`. After iterating over all weeks and cities, the solution will be the maximum value in the last row of the array `f`, which represents the maximum vacation days for the last week at any city.

The elegance of dynamic programming lies in its ability to break down complex decisions into simpler, overlapping subproblems, storing those results, and reusing them in an optimal manner to construct an answer for the global problem.

Solution Approach

The provided solution is an implementation of dynamic programming. To understand the code, let's break it down into its core components:

1. **Dynamic Array Initialization**:

   ```
   f = [[-inf] * n for _ in range(K + 1)]
   f[0][0] = 0
   ```

   In this block, we're initializing an array `f` that holds the maximum vacation days. `-inf` is used to indicate that a state is initially unreachable. The first city at the first week is set to 0 since we start with no vacation days utilized prior to the first week.

2. **Nested Loops for State Transition**:

   ```
   for k in range(1, K + 1):
       for j in range(n):
           f[k][j] = f[k - 1][j]
           for i in range(n):
               if flights[i][j]:
                   f[k][j] = max(f[k][j], f[k - 1][i])
           f[k][j] += days[j][k - 1]
   ```

   - The outermost loop iterates over weeks (`k`), as each iteration corresponds to the state of each week.
   - The second loop iterates over every city (`j`), as we want to calculate the maximum vacation days if we end up in city `j` at the end of week `k`.
   - Inside the second loop, there's an inner loop over every city (`i`) that checks if a flight is available from city `i` to city `j`. If a flight is available, the maximum vacation days for city `j` are updated by taking the maximum of the current value or the value for city `i` from the previous week.
   - After deciding whether to stay in the same city or fly from another city, we add the vacation days for city `j` in the corresponding week (`days[j][k - 1]`).
3. **Final Result**:

   ```
   return max(f[-1][j] for j in range(n))
   ```

   - After updating the array `f` for all the weeks, the final result is the maximum value in the last week (which is at index `K`) for all cities. This represents the scenario where, irrespective of the city you are in, you want the maximum vacation days accumulated by the last week of your travel schedule.

This solution effectively utilizes a bottom-up dynamic programming approach, filling out a table of sub-solutions (vacation days for each city at the end of each week), which are then combined to form the solution to the overarching problem (maximum vacation days after `k` weeks).

The approach ensures we consider every potential week-to-week transition while adhering to the constraints of available flights and maximum vacation days in each city. By completing the matrix, the algorithm canvasses all strategic paths without redundant recalculations, which characterizes dynamic programming's optimization.

Example Walkthrough

Let's consider a scenario with 3 cities (`n = 3`) and 2 weeks (`K = 2`) to spend on vacation. The available flights matrix and days matrix are as follows:

```
flights = [
    [0, 1, 1],
    [1, 0, 1],
    [1, 1, 0]
]

days = [
    [1, 3],
    [6, 0],
     [3, 4]
 ]
```

The flights matrix indicates that from any city, one can fly to either of the other two cities. The days matrix shows the vacation days one can spend in each city during each of the two weeks.

Initialization

- First, we initialize the dynamic array `f` with `-infinity` to indicate unknown/unreachable states, except for the starting point which is city `0` at week `0` with `0` vacation days spent.

Iteration Process

Week 1:

- When we are at the start of week 1 (`k = 1`), we look at each city `j`.
  - For city `0`, since we start there, we don't need to consider flights. We can take advantage of the vacation days in city `0`, which is `1`. So `f[1][0] = 1`.
  - For city `1`, there is a flight from city `0` to city `1`. We can either stay in city `0` and accumulate `1` day or fly to city `1` and accumulate `6` days. So we choose to fly and `f[1][1] = 6`.
  - For city `2`, there's also a flight from city `0` to city `2`. Again, we can either stay in city `0` with `1` day or fly to city `2` and accumulate `3` days. So we fly and `f[1][2] = 3`.

Week 2:

- Moving to week 2 (`k = 2`), we now examine the options from the perspective of each city `j`.
  - In city `0`, we can either come from city `0` with `1` day (accumulated from last week) or come from city `2` with `3` days as there are flights available from both. We choose the latter for a total of `3 + 3 = 6` days because `days[0][1] = 3`.
  - In city `1`, the flight from city `0` to city `1` doesn't give us any vacation days for this week as `days[1][1] = 0`. So we stay in city `1` and keep the `6` days accumulated from last week.
  - In city `2`, we can fly from either city `1` or city `0`, with last week's days of `6` and `3`, respectively. Since city `2` has `4` vacation days available this week (`days[2][1]`), we choose to come from city `1` for a total of `6 + 4 = 10` days in city `2`.

Final Decision and Result

- After completing the iteration process, we find the maximum value in the `f` array for the last week. This would be `max(f[2])` which equates to `max([6, 6, 10]) = 10`. The maximum vacation days one can spend is `10` by traveling from city `0` to city `1` in the first week and then to city `2` in the second week.

This example illustrates each step of the dynamic programming approach and shows how the solution can be traced and optimized week by week, given the flight availability and the vacation days constraints.

Python Solution

```python
from typing import List

class Solution:
    def maxVacationDays(self, flights: List[List[int]], days: List[List[int]]) -> int:
        # Number of cities
        num_cities = len(flights)
        # Number of weeks
        num_weeks = len(days[0])
        # Initialize the dp array with -infinity to signify unvisited states
        # plus one row for initial state (week 0)
        dp = [[float('-inf')] * num_cities for _ in range(num_weeks + 1)]
        # Starting city has 0 vacation days initially
        dp[0][0] = 0

        # Loop through each week
        for week in range(1, num_weeks + 1):
            # Loop through each city for the current week
            for current_city in range(num_cities):
                # First, assume staying in the same city as the previous week
                dp[week][current_city] = dp[week - 1][current_city]

                # Then, check for other cities we can fly to
                for previous_city in range(num_cities):
                    # If there is a flight from a previous city to the current city
                    if flights[previous_city][current_city]:
                        # Take the max of staying or flying from the previous city
                        dp[week][current_city] = max(dp[week][current_city], dp[week - 1][previous_city])

                # Add the vacation days for the current city and week
                dp[week][current_city] += days[current_city][week - 1]

        # Find the max vacation days from the last week across all cities
        return max(dp[num_weeks][city] for city in range(num_cities))

```

Java Solution

```java
class Solution {
    public int maxVacationDays(int[][] flights, int[][] days) {
        int numCities = flights.length; // Number of cities
        int numWeeks = days[0].length; // Number of weeks
        final int INF = Integer.MIN_VALUE; // Representation of negative infinity
        int[][] dp = new int[numWeeks + 1][numCities]; // DP table for storing max vacation days

        // Initialize DP table with negative infinity indicating not reachable
        for (int[] week : dp) {
            Arrays.fill(week, INF);
        }

        // Base case: start at city 0 at week 0
        dp[0][0] = 0;

        // Fill the DP table week by week
        for (int week = 1; week <= numWeeks; ++week) {
            for (int currentCity = 0; currentCity < numCities; ++currentCity) {
                // Max vacation days in the current city without flying
                dp[week][currentCity] = dp[week - 1][currentCity];

                // Check for flights from other cities to the current city and update max days
                for (int prevCity = 0; prevCity < numCities; ++prevCity) {
                    if (flights[prevCity][currentCity] == 1) {
                        dp[week][currentCity] = Math.max(dp[week][currentCity], dp[week - 1][prevCity]);
                    }
                }

                // If the city is reachable, add vacation days for the current week
                if (dp[week][currentCity] != INF) {
                    dp[week][currentCity] += days[currentCity][week - 1];
                }
            }
        }

        // Find the maximum vacation days from all reachable cities at the last week
        int maxVacationDays = 0;
        for (int city = 0; city < numCities; ++city) {
            maxVacationDays = Math.max(maxVacationDays, dp[numWeeks][city]);
        }

        return maxVacationDays;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int maxVacationDays(vector<vector<int>>& flights, vector<vector<int>>& days) {
        // n represents the number of cities
        int numCities = flights.size();
        // k represents the number of weeks
        int numWeeks = days[0].size();
        // dp array to store the maximum vacation days up to week k in city j
        int dp[numWeeks + 1][numCities];
        // initializing the dp array with minimum possible value
        memset(dp, -0x3f, sizeof(dp));
        // starting at city 0 with 0 vacation days
        dp[0][0] = 0;

        // Iterate over each week
        for (int week = 1; week <= numWeeks; ++week) {
            // Iterate over each city for the current week
            for (int city = 0; city < numCities; ++city) {
                // Set the dp value for the current city and week to the vacation days of the same city in the previous week
                dp[week][city] = dp[week - 1][city];
                // Check all possible cities where we could have come from to the current city
                for (int prevCity = 0; prevCity < numCities; ++prevCity) {
                    if (flights[prevCity][city] == 1) {
                        // Update the dp value for the current city and week with the maximum between the current dp value and
                        // vacation days of the previous city in the previous week
                        dp[week][city] = max(dp[week][city], dp[week - 1][prevCity]);
                    }
                }
                // Add the vacation days of the current city for the current week
                dp[week][city] += days[city][week - 1];
            }
        }

        // Initialize answer for the maximum vacation days
        int maxVacation = 0;
        // Iterate over each city at the last week to find the maximum vacation days
        for (int city = 0; city < numCities; ++city) {
            maxVacation = max(maxVacation, dp[numWeeks][city]);
        }

        return maxVacation;
    }
};

```

Typescript Solution

```typescript
// Define the maximumDays function to calculate max vacation days
function maximumDays(flights: number[][], days: number[][]): number {
    // n represents the number of cities
    let numCities: number = flights.length;
    // k represents the number of weeks
    let numWeeks: number = days[0].length;

    // dp array to store the maximum vacation days up to week k in city j
    let dp: number[][] = new Array(numWeeks + 1).fill(0).map(() => new Array(numCities).fill(Number.MIN_SAFE_INTEGER));

    // starting at city 0 with 0 vacation days
    dp[0][0] = 0;

    // Iterate over each week
    for (let week = 1; week <= numWeeks; ++week) {
        // Iterate over each city for the current week
        for (let city = 0; city < numCities; ++city) {
            // Initially set the dp value for the current city and week
            // to the vacation days of the same city in the previous week
            dp[week][city] = dp[week - 1][city];

            for (let prevCity = 0; prevCity < numCities; ++prevCity) {
                // Check if there is a flight from prevCity to city
                if (flights[prevCity][city] === 1) {
                    // Update the dp value for the current city and week with the maximum between the current dp value and
                    // the vacation days of the previous city in the previous week
                    dp[week][city] = Math.max(dp[week][city], dp[week - 1][prevCity]);
                }
            }
            // Add the vacation days of the current city for the current week
            dp[week][city] += days[city][week - 1];
        }
    }

    // Initialize the answer for the maximum vacation days
    let maxVacation: number = 0;
    // Iterate over each city at the last week to find the maximum vacation days
    for (let city = 0; city < numCities; ++city) {
        maxVacation = Math.max(maxVacation, dp[numWeeks][city]);
    }

    return maxVacation;
}

// Usage:
// const flights = [[...], [...], ...];
// const days = [[...], [...], ...];
// console.log(maximumDays(flights, days));

```

Time and Space Complexity

The code provided aims to solve a problem where it calculates the maximum vacation days one can take given certain flight connectivity and days one can spend in each city per week.

Time Complexity

The time complexity of the algorithm is primarily determined by the nested loops:

1. The outer loop runs for `K` iterations, where `K` is the number of weeks.
2. The first inner loop runs for `n` iterations, where `n` is the number of cities.
3. The second inner loop (nested inside the first one) also runs for `n` iterations, again going through all cities.

Each of these loop iterations involves constant time operations or operations that take `O(1)` time.

Thus, the total time complexity is given by `O(K * n^2)`.

Space Complexity

The space complexity is determined by the storage used for the `f` array, which is a 2D array with dimensions `(K + 1) x n`. Here, `f[k][j]` represents the maximum vacation days one can achieve till week `k` in city `j`.

Thus, the space complexity of the algorithm is `O(K * n)`.