Previous Next

568. Maximum Vacation days

April 29, 2017 | 13.9K views

★★★★★
Average Rating: 5 (9 votes)

LeetCode wants to give one of its best employees the option to travel among **N** cities to collect algorithm problems. But all work and no play makes Jack a dull boy, you could take vacations in some particular cities and weeks. Your job is to schedule the traveling to maximize the number of vacation days you could take, but there are certain rules and restrictions you need to follow.

Rules and restrictions:

1. You can only travel among **N** cities, represented by indexes from 0 to N-1. Initially, you are in the city indexed 0 on **Monday**.
2. The cities are connected by flights. The flights are represented as a **N*N** matrix (not necessary symmetrical), called **flights** representing the airline status from the city i to the city j. If there is no flight from the city i to the city j, **flights[i][j] = 0**; Otherwise, **flights[i][j] = 1**. Also, **flights[i][i] = 0** for all i.
3. You totally have **K** weeks (**each week has 7 days**) to travel. You can only take flights at most once **per day** and can only take flights on each week's **Monday** morning. Since flight time is so short, we don't consider the impact of flight time.
4. For each city, you can only have restricted vacation days in different weeks, given an **N*K** matrix called **days** representing this relationship. For the value of **days[i][j]**, it represents the maximum days you could take vacation in the city **i** in the week **j**.

You're given the **flights** matrix and **days** matrix, and you need to output the maximum vacation days you could take during **K** weeks.

Example 1:

```
Input:flights = [[0,1,1],[1,0,1],[1,1,0]], days = [[1,3,1],[6,0,3],[3,3,3]]
Output: 12
Explanation:
Ans = 6 + 3 + 3 = 12.

One of the best strategies is:
1st week : fly from city 0 to city 1 on Monday, and play 6 days and work 1 day.
(Although you start at city 0, we could also fly to and start at other cities since it
2nd week : fly from city 1 to city 2 on Monday, and play 3 days and work 4 days.
3rd week : stay at city 2, and play 3 days and work 4 days.
```

Example 2:

```
Input:flights = [[0,0,0],[0,0,0],[0,0,0]], days = [[1,1,1],[7,7,7],[7,7,7]]
Output: 3
Explanation:
Ans = 1 + 1 + 1 = 3.

Since there is no flights enable you to move to another city, you have to stay at city
For each week, you only have one day to play and six days to work.
So the maximum number of vacation days is 3.
```

Example 3:

```
Input:flights = [[0,1,1],[1,0,1],[1,1,0]], days = [[7,0,0],[0,7,0],[0,0,7]]
Output: 21
Explanation:
Ans = 7 + 7 + 7 = 21

One of the best strategies is:
1st week : stay at city 0, and play 7 days.
2nd week : fly from city 0 to city 1 on Monday, and play 7 days.
3rd week : fly from city 1 to city 2 on Monday, and play 7 days.
```

Note:

1. **N and K** are positive integers, which are in the range of [1, 100].
2. In the matrix **flights**, all the values are integers in the range of [0, 1].
3. In the matrix **days**, all the values are integers in the range [0, 7].
4. You could stay at a city beyond the number of vacation days, but you should **work** on the extra days, which won't be counted as vacation days.
5. If you fly from the city A to the city B and take the vacation on that day, the deduction towards vacation days will count towards the vacation days of city B in that week.
6. We don't consider the impact of flight hours towards the calculation of vacation days.

Solution

Approach #1 Using Depth First Search [Time Limit Exceeded]

Algorithm

In the brute force approach, we make use of a recursive function dfs, which returns the number of vacations which can be taken startring from cur_city as the current city and $weekno$ as the starting week.

In every function call, we traverse over all the cities(represented by i) and find out all the cities which are connected to the current city, cur_city. Such a city is represented by a 1 at the corresponding $flights[cur\_city][i]$ position. Now, for the current city, we can either travel to the city which is connected to it or we can stay in the same city. Let's say the city to which we change our location from the current city be represented by j. Thus, after changing the city, we need to find the number of vacations which we can take from the new city as the current city and the incremented week as the new starting week. This count of vacations can be represented as: $days[j][weekno] + dfs(flights, days, j, weekno + 1)$.

Thus, for the current city, we obtain a number of vacations by choosing different cities as the next cities. Out of all of these vacations coming from different cities, we can find out the maximum number of vacations that need to be returned for every dfs function call.

Java Copy

```java

public class Solution {
    public int maxVacationDays(int[][] flights, int[][] days) {
        return dfs(flights, days, 0, 0);
    }
    public int dfs(int[][] flights, int[][] days, int cur_city, int weekno) {
        if (weekno == days[0].length)
            return 0;
        int maxvac = 0;
        for (int i = 0; i < flights.length; i++) {
            if (flights[cur_city][i] == 1 || i == cur_city) {
                int vac = days[i][weekno] + dfs(flights, days, i, weekno + 1);
                maxvac = Math.max(maxvac, vac);
            }
        }
        return maxvac;
    }
}

```

Complexity Analysis

- Time complexity : $O(n^k)$. Depth of Recursion tree will be k and each node contains n branches in the worst case. Here n represents the number of cities and k is the total number of weeks.
- Space complexity : $O(k)$. The depth of the recursion tree is k.

Approach #2 Using DFS with memoization [Accepted]:

Algorithm

In the last approach, we make a number of redundant function calls, since the same function call of the form `dfs(flights, days, cur_city, weekno)` can be made multiple number of times with the same cur_city and $weekno$. These redundant calls can be pruned off if we make use of memoization.

In order to remove these redundant function calls, we make use of a 2-D memoization array $memo$. In this array, $memo[i][j]$ is used to store the number of vacactions that can be taken using the i^{th} city as the current city and the j^{th} week as the starting week. This result is equivalent to that obtained using the function call: `dfs(flights, days, i, j)`. Thus, if the $memo$ entry corresponding to the current function call already contains a valid value, we can directly obtain the result from this array instead of going deeper into recursion.

Java Copy

```java
public class Solution {
    public int maxVacationDays(int[][] flights, int[][] days) {
        int[][] memo = new int[flights.length][days[0].length];
        for (int[] l: memo)
            Arrays.fill(l, Integer.MIN_VALUE);
        return dfs(flights, days, 0, 0, memo);
    }
    public int dfs(int[][] flights, int[][] days, int cur_city, int weekno, int[][] memo) {
        if (weekno == days[0].length)
            return 0;
        if (memo[cur_city][weekno] != Integer.MIN_VALUE)
            return memo[cur_city][weekno];
        int maxvac = 0;
        for (int i = 0; i < flights.length; i++) {
            if (flights[cur_city][i] == 1 || i == cur_city) {
                int vac = days[i][weekno] + dfs(flights, days, i, weekno + 1, memo);
                maxvac = Math.max(maxvac, vac);
            }
        }
        memo[cur_city][weekno] = maxvac;
        return maxvac;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^2k)$. $memo$ array of size $n*k$ is filled and each cell filling takes O(n) time .
- Space complexity : $O(n*k)$. $memo$ array of size $n*k$ is used. Here n represents the number of cities and k is the total number of weeks.

Approach #3 Using 2-D Dynamic Programming [Accepted]:

Algorithm

The idea behind this approach is as follows. The maximum number of vacations that can be taken given we start from the i^{th} city in the j^{th} week is not dependent on the the vacations that can be taken in the earlier weeks. It only depends on the number of vacations that can be taken in the upcoming weeks and also on the connections between the various cities($flights$).

Therefore, we can make use of a 2-D dp, in which $dp[i][k]$ represents the maximum number of vacations which can be taken starting from the i^{th} city in the k^{th} week. This dp is filled in the backward manner(in terms of the week number).

While filling up the entry for $dp[i][k]$, we need to consider the following cases:

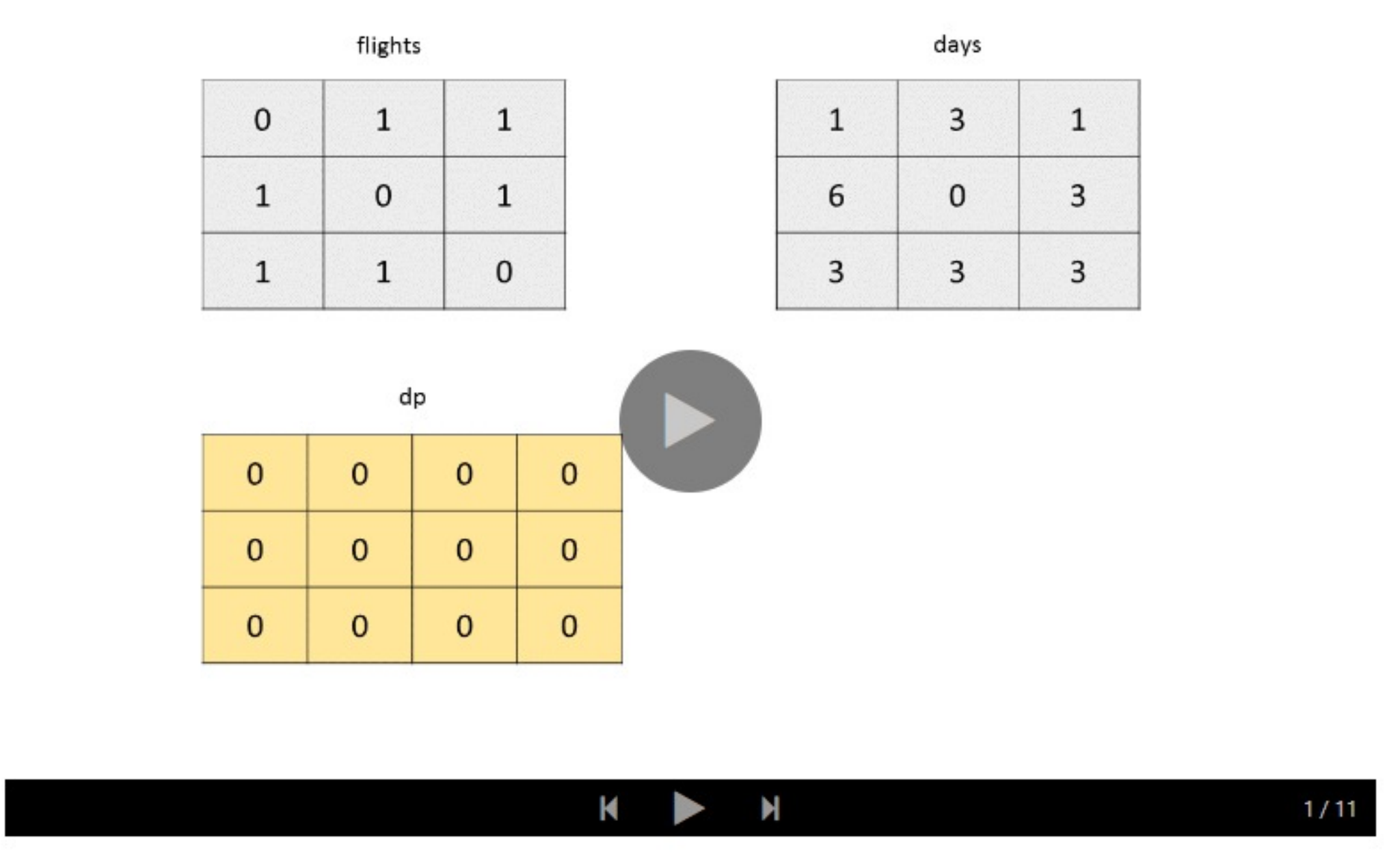
1. We start from the i^{th} city in the k^{th} week and stay in the same city for the $(k+1)^{th}$ week. Thus, the factor to be considered for updating the $dp[i][k]$ entry will be given by: $days[i][k] + dp[i, k+1]$.
2. We start from the i^{th} city in the k^{th} week and move to the j^{th} city in the $(k+1)^{th}$ week. But, for changing the city in this manner, we need to be able to move from the i^{th} city to the j^{th} city i.e. $flights[i][j]$ should be 1 for such i and j.

But, while changing the city from i^{th} city in the k^{th} week, we can move to any j^{th} city such that a connection exists between the i^{th} city and the j^{th} city i.e. $flights[i][j] = 1$. But, in order to maximize the number of vacations that can be taken starting from the i^{th} city in the k^{th} week, we need to choose the destination city that leads to maximum no. of vacations. Thus, the factor to be considered here, is given by: $\max days[j][k] + days[j, k+1]$, for all i, j, k satisfying $flights[i][j] = 1, 0 \leq i, j \leq n, where n$$ refers to the number of cities.

At the end, we need to find the maximum out of these two factors to update the $dp[i][k]$ value.

In order to fill the dp values, we start by filling the entries for the last week and proceed backwards. At last, the value of $dp[0][0]$ gives the required result.

The following animation illustrates the process of filling the dp array.



flights

0	1	1
1	0	1
1	1	0

days

1	3	1
6	0	3
3	3	3

dp

0	0	0	0
0	0	0	0
0	0	0	0

Below code is inspired by @hackerhuang

Java Copy

```java
public class Solution {
    public int maxVacationDays(int[][] flights, int[][] days) {
        if (days.length == 0 || flights.length == 0) return 0;
        int[][] dp = new int[days.length][days[0].length + 1];
        for (int week = days[0].length - 1; week >= 0; week--) {
            for (int cur_city = 0; cur_city < days.length; cur_city++) {
                dp[cur_city][week] = days[cur_city][week] + dp[cur_city][week + 1];
                for (int dest_city = 0; dest_city < days.length; dest_city++) {
                    if (flights[cur_city][dest_city] == 1) {
                        dp[cur_city][week] = Math.max(days[dest_city][week] + dp[dest_city][week + 1], dp[cur_city][week]);
                    }
                }
            }
        }
        return dp[0][0];
    }
```

Complexity Analysis

- Time complexity : $O(n^2k)$. dp array of size $n*k$ is filled and each cell filling takes O(n) time. Here n represents the number of cities and k is the total number of weeks.
- Space complexity : $O(n*k)$. dp array of size $n*k$ is used.

Approach #4 Using 1-D Dynamic Programming [Accepted]:

Algorithm

As can be observed in the previous approach, in order to update the dp entries for i^{th} week, we only need the values corresponding to $(i+1)^{th}$ week along with the $days$ and $flights$ array. Thus, instead of using a 2-D dp array, we can omit the dimension corresponding to the weeks and make use of a 1-D dp array.

Now, $dp[i]$ is used to store the number of vacations that provided that we start from the i^{th} city in the current week. The procedure remains the same as that of the previous approach, except that we make the updations in the same dp row again and again. In order to store the dp values corresponding to the current week temporarily, we make use of a $temp$ array so that the original dp entries corresponding to $week+1$ aren't altered.

Java Copy

```java
public class Solution {
    public int maxVacationDays(int[][] flights, int[][] days) {
        if (days.length == 0 || flights.length == 0) return 0;
        int[] dp = new int[days.length];
        for (int week = days[0].length - 1; week >= 0; week--) {
            int[] temp = new int[days.length];
            for (int cur_city = 0; cur_city < days.length; cur_city++) {
                temp[cur_city] = days[cur_city][week] + dp[cur_city];
                for (int dest_city = 0; dest_city < days.length; dest_city++) {
                    if (flights[cur_city][dest_city] == 1) {
                        temp[cur_city] = Math.max(days[dest_city][week] + dp[dest_city], temp[cur_city]);
                    }
                }
            }
            dp = temp;
        }

        return dp[0];
    }
}
```

Complexity Analysis

- Time complexity : $O(n^2k)$. dp array of size $n*k$ is filled and each cell filling takes O(n) time. Here n represents the number of cities and k is the total number of weeks.
- Space complexity : $O(k)$. dp array of size nk is used.

Analysis written by: @vinod23

Rate this article: ★★★★★

Previous Next

Comments: 10 Sort By

Type comment here... (Markdown is supported)

Preview Post

tairan ★6 November 9, 2017 1:18 PM

The space complexity for approach #4 should be O(n), days.length is actually the number of cities.

6 Share Reply

SHOW 2 REPLIES

Hellofafar ★7 November 11, 2017 12:52 PM

In approach 3,
dp[cur_city][week] = Math.max(days[dest_city][week] + dp[dest_city][week + 1], dp[cur_city][week]);
should be
dp[cur_city][week] = Math.max(days[cur_city][week] + dp[dest_city][week + 1], dp[cur_city][week]);

3 Share Reply

SHOW 1 REPLY

jeremyasm ★175 June 17, 2019 9:02 PM

Click 'View in Article' in the top right corner of the Solution Section, then you see the complete 4 approches and code. At first, I thougt it was incomplete ..

1 Share Reply

jeremyasm ★175 June 17, 2019 8:57 PM

Java code for solution 3

```
    public int maxVacationDays(int[][] flights, int[][] days) {
        int n = flights.length;
        int k = days[0].length;
```

Read More

1 Share Reply

rw363 ★5 January 13, 2019 2:24 PM

can anyone explain the time complexity of approach#2. How does it come up with O(n^2k)? Thanks!

0 Share Reply

SHOW 1 REPLY

gigiwaiwai ★2 August 25, 2018 7:04 AM

"maxdays[j][k]+days[j,k+1]" should be "maxdays[j][k]+dp[j,k+1]"

0 Share Reply

Rainbow14 ★0 April 23, 2018 3:20 AM

With DP approach, i'm curious on how do we make sure we travel all N cities?

0 Share Reply

SHOW 2 REPLIES

lijingyabeyond ★19 April 5, 2020 9:30 AM

Example 2 is the case now with coronavirus stops all flights.

0 Share Reply

zhilich ★36 February 13, 2020 6:43 PM

I think that all DFS solutions are actually DP as well. True DFS would go forward vs backwards and examine all possible paths and only do Math.Max at the deepest level when all cities in the path are known.

0 Share Reply

galster ★243 November 5, 2018 7:28 AM

Both solution (1) & (2) are incorrect;
They don't account for the fact that on the first Monday we can start our dfs from city 0 or any city connected by flight to it.

-3 Share Reply

SHOW 1 REPLY