# Maximum Vacation Days

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 i
t is Monday.)
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 cit
y 0 for the whole 3 weeks.
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 1

Solution 1. DFS. The idea is just try each `possible` city for every week and keep tracking the `max` vacation days. Time complexity O(N^K). Of course it will TLE....

```
public class Solution {
    int max = 0, N = 0, K = 0;

    public int maxVacationDays(int[][] flights, int[][] days) {
        N = flights.length;
        K = days[0].length;
        dfs(flights, days, 0, 0, 0);

        return max;
    }

    private void dfs(int[][] f, int[][] d, int curr, int week, int sum) {
        if (week == K) {
            max = Math.max(max, sum);
            return;
        }

        for (int dest = 0; dest < N; dest++) {
            if (curr == dest || f[curr][dest] == 1) {
                dfs(f, d, dest, week + 1, sum + d[dest][week]);
            }
        }
    }
}
```

Solution 2. DP. `dp[i][j]` stands for the max vacation days we can get in week `i` staying in city `j`. It's obvious that `dp[i][j] = max(dp[i - 1][k] + days[j][i]) (k = 0...N - 1, if we can go from city k to city j)`. Also because values of week `i` only depends on week `i - 1`, so we can compress two dimensional `dp` array to one dimension. Time complexity O(K * N * N) as we can easily figure out from the 3 level of loops.

```java
public class Solution {
    public int maxVacationDays(int[][] flights, int[][] days) {
        int N = flights.length;
        int K = days[0].length;
        int[] dp = new int[N];
        Arrays.fill(dp, Integer.MIN_VALUE);
        dp[0] = 0;

        for (int i = 0; i < K; i++) {
            int[] temp = new int[N];
            Arrays.fill(temp, Integer.MIN_VALUE);
            for (int j = 0; j < N; j++) {
                for(int k = 0; k < N; k++) {
                    if (j == k || flights[k][j] == 1) {
                        temp[j] = Math.max(temp[j], dp[k] + days[j][i]);
                    }
                }
            }
            dp = temp;
        }

        int max = 0;
        for (int v : dp) {
            max = Math.max(max, v);
        }

        return max;
    }
}
```

written by shawngao original link here

## Solution 2

Let's maintain `best[i]`, the most vacation days you can have ending in city `i` on week t. At the end, we simply want max(best), the best answer for any ending city.

For every flight `i -> j` (including staying in the same city, when `i == j`), we have a candidate answer best[j] = best[i] + days[j][t], and we want the best answer of those.

When the graph is sparse, we can precompute `flights_available[i] = [j for j, adj in enumerate(flights[i]) if adj or i == j]` instead to save some time, but this is not required.

```
def maxVacationDays(self, flights, days):
    NINF = float('-inf')
    N, K = len(days), len(days[0])
    best = [NINF] * N
    best[0] = 0

    for t in xrange(K):
        cur = [NINF] * N
        for i in xrange(N):
            for j, adj in enumerate(flights[i]):
                if adj or i == j:
                    cur[j] = max(cur[j], best[i] + days[j][t])
        best = cur
    return max(best)
```

written by awice original link here

## Solution 3

```
class Solution {
public:
    int maxVacationDays(vector<vector<int>>& flights, vector<vector<int>>& days) {
        int res = 0;
        int N = flights.size();
        int K = days[0].size();
        vector<int> v(N,0), v_prev(N,0);
        unordered_set<int> r,r_prev;
        r.insert(0);
        r_prev = r;
        for(int i = 0; i < N; ++i)
           flights[i][i] = 1;
        for(int w = 0; w < K; ++w)
        {
            for(auto it = r_prev.begin(); it != r_prev.end(); ++it)
            {
                for(int i = 0; i < N; ++i)
                {
                    if(flights[*it][i])
                    {
                        v[i] = max(v[i],v_prev[*it]+days[i][w]);
                        res = max(res,v[i]);
                        r.insert(i);
                    }
                }
            }
            v_prev = v;
            r_prev = r;
        }
        return res;

    }
};
```

written by i9r0k original link here