1575. Count All Possible Routes

### Memoization Array **Dynamic Programming** Hard

In this problem, we're given an array locations which contains distinct positive integers representing the position of different cities. The positions are essentially distances between some reference point and each city, but those distances are abstract in the problem; they don't need to represent real-world units.

**Leetcode Link** 

Additionally, we have three integers: start, finish, and fuel. These represent the index of the starting city in the locations array, the index of the destination city, and the initial amount of fuel we have, respectively.

• Traveling from city i to city j consumes |locations[i] - locations[j] | units of fuel (|x| denotes the absolute value of x). You

The rules of the game are:

**Problem Description** 

cannot make the trip if you don't have enough fuel to reach city j. You can visit any city more than once, and the fuel left can NOT be negative at any point.

the destination if you are already there).

You can travel from any city i to any different city j.

- The goal is to determine how many different routes you can take from the start city to the finish city, given the constraints of fuel usage. Importantly, the number of routes can be very large, so we return the result modulo 10^9 + 7.
- Intuition

To solve this problem, we can use depth-first search (DFS) combined with memoization to avoid redundant calculations. Here is the thought process we use to arrive at the solution:

1. The core of the solution is the function dfs(i, k) which calculates the number of possible routes from city i with k fuel remaining to reach the finish city.

2. Initially, if the current city i is the same as the destination finish, we record one possible route (as no fuel is needed to reach

- 3. If we don't have enough fuel to directly reach the destination from our current city, there's no need to explore further from this state; the function returns 0.
- 4. For each city i, we iterate over all other cities j, and recursively call our dfs function to find out how many routes we can take from city j to finish, with the fuel reduced by the amount it would take to travel from i to j.
- fuel), so we do not repeat computations for those states. This dramatically improves performance. 6. Since we need to return the number of routes modulo 10^9 + 7, we take the modulo after every addition to keep the numbers

5. We use memoization to store already calculated states (representing unique combinations of the current city and the remaining

within an integer range and comply with the problem constraints.

By summarizing this intuition, we get our dynamic top-down approach (DFS with memoization). It's important to note that the

up. However, that approach is more complex and requires a deeper understanding of DP to implement correctly.

- number of possible routes could be enormous, hence the need for modulo operation to avoid integer overflow. For an alternative and more space-efficient approach, one could also use dynamic programming (DP) to solve this problem bottom-
- The solution to this problem leverages recursion and memoization to implement a depth-first search that efficiently counts all the possible routes from the starting city to the destination, considering the given fuel limit. Here's how the implementation unfolds, step by step:

1. Memoization: To prevent the same calculations from being executed multiple times, we employ memoization. This is done by

decorating our recursive function with @cache (available in Python 3.9+). The cache decorator automatically memorizes the

If the remaining fuel k is less than the absolute difference in location between city i and the destination city finish (i.e., k <</li>

abs(locations[i] - locations[finish])), the function returns 0, indicating that the destination cannot be reached from

return values of the function calls based on their input arguments. 2. Recursive Function dfs(i, k): This function takes the current city index i and the remaining fuel k as arguments. It returns the

count of all possible routes from city i to the destination that can be achieved with k or less amount of fuel.

### If city i is the destination city (i.e., i == finish), the function initializes the route count ans with 1 since you're already at the destination.

this state.

4. Recursive Exploration:

5. Returning the Result:

from city i to city j.

the given amount of fuel.

number of cities and m is the amount of fuel.

Suppose we have the following input:

start: 0 (the starting city is at position 4)

finish: 2 (the destination city is at position 1)

index 0 (which is at location 4) with 4 fuel.

i is not equal to finish, so we do not add 1 to our route count.

Check base case: Not at destination, and have enough fuel.

Check base case: Not at destination, and have enough fuel.

dfs(2, 1) from here adds 1 to our routes.

To city index 0 (already visited, not destination): dfs(0, 2).

Recursive exploration from city index 1:

Recursive exploration from city index 0:

(the exact paths are  $0 \rightarrow 2$  and  $0 \rightarrow 1 \rightarrow 2$ ).

■ To city index 1: dfs(1, 1).

locations array: [4, 3, 1]

Using the solution approach:

fuel: 4 units

3. Base Cases:

**Solution Approach** 

∘ We recursively call dfs(j, k - abs(locations[i] - locations[j])) to find out the number of routes from city j to the destination with the updated remaining fuel. • The result for each city j is added to ans modulo mod (which is 10\*\*9 + 7) to ensure we maintain the result within the bounds of an integer and comply with the problem constraints about the modulo.

We iterate through all other possible cities (represented by index j), and if j != i, we calculate the fuel required to travel

6. Solution Execution: • The entry point for our calculation is dfs(start, fuel), this call initiates the depth-first search from the starting city with

We don't consider dynamic programming (DP) approach here, but it's important to note that it could also be utilized in a situation

where memory consumption needs to be optimized. The DP approach would iteratively build up a table f[i][k] representing the

Both DFS with memoization and the DP approach use time complexity  $0(n^2 * m)$  and space complexity 0(n \* m) where n is the

number of paths from city i with k remaining fuel to finish. The final answer would then be read from f[start][fuel].

• The function finally returns ans, which is the total number of routes from city i with fuel k to the destination.

Example Walkthrough Let's go through an example to illustrate the solution approach using the depth-first search (DFS) with memoization.

2. Recursive Function dfs(i, k): Our first call will be dfs(0, 4). 3. Base Cases: We check our base cases:

We have enough fuel (more than the distance from our current city to finish), so we do not return 0.

1. Initialization: We start by initializing our memoization cache (not shown in this example). Then we begin our algorithm at city

## From city index 0, we calculate the fuel required to get to other cities: ■ To city index 1 (locations[1] == 3): fuel cost is |4 - 3| = 1. We then recurse with dfs(1, 4 - 1), which is dfs(1, 3).

5. Exploring dfs(1, 3):

7. Exploring dfs(0, 2):

8. Exploring dfs(1, 1):

9. Adding up the routes:

10. Returning the Result:

the total count.

Python Solution

class Solution:

) -> int:

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

41

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

41

42

43

45

44 }

37 # Example of usage

38 # sol = Solution()

from typing import List

def countRoutes(

@cache

return routes\_count

39 # routes\_count = sol.countRoutes([2, 3, 6, 8, 4], 1, 3, 5)

locationCount = locations.length;

this.finishLocationIndex = finish;

this.locations = locations;

return dfs(start, fuel);

return 0;

40 # print(routes\_count) # Output will depend on the inputs provided

return dfs(start, fuel)

from functools import cache

conclude dfs(0, 4) = 2.

4. Recursive Exploration:

6. Exploring dfs(2, 1): Check base case: We're at the destination, so this recursive path adds 1 to our route count.

■ To city index 2 (destination): direct fuel cost is |3 - 1| = 2. Since we have 3 fuel, we can reach the destination, so

■ To city index 2 (destination): direct fuel cost is 3, but we only have 2 fuel, so this path does not add to our route count.

• From the original dfs(0, 4), we got routes from exploring both dfs(1, 3) and dfs(2, 1), which in total gives us 2 routes

• The total number of possible routes from our starting city with the given fuel to reach the destination is 2 when using DFS

with memoization. The answer is calculated as  $(1 \text{ from } dfs(2, 1)) + (1 \text{ from } dfs(1, 3) \rightarrow dfs(2, 1))$ . Thus, we

■ To city index 2 (locations[2] == 1): fuel cost is |4 - 1| = 3. We then recurse with dfs(2, 4 - 3), which is dfs(2, 1).

- Check base case: Not at destination, and we don't have enough fuel left to go anywhere else, so this does not add to our route count.
- This example demonstrates how we recursively explore different paths, using memoization to avoid redundant calculations, and how we handle the returns when we reach the destination or when we lack the fuel to continue. Each recursive call returns the number of ways to get from the current city to the destination with the given amount of remaining fuel. Finally, we sum all possible routes to get

self, locations: List[int], start: int, finish: int, fuel: int

def dfs(current\_location\_index: int, remaining\_fuel: int) -> int:

for next\_location\_index, location in enumerate(locations):

# Calculate the fuel cost for the next move

# Return the total number of routes from the current location

public int countRoutes(int[] locations, int start, int finish, int fuel) {

// Helper function using Depth First Search (DFS) to compute the number of routes.

// If the remaining fuel is not enough to go from currentIndex to finish, return 0.

if (remainingFuel < Math.abs(locations[currentIndex] - locations[finishLocationIndex])) {</pre>

// If the current location is the finish location, start with a count of 1, otherwise 0.

// Try all other locations using the remaining fuel and accumulate the routes count.

memoizationCache = new Integer[locationCount][fuel + 1];

if (memoizationCache[currentIndex][remainingFuel] != null) {

return memoizationCache[currentIndex][remainingFuel];

int routeCount = currentIndex == finishLocationIndex ? 1 : 0;

for (int nextIndex = 0; nextIndex < locationCount; ++nextIndex) {</pre>

// Store computed result in cache (memoization) and return the result.

int countRoutes(vector<int>& locations, int start, int finish, int fuel) {

memset(dp, -1, sizeof(dp)); // Initialize all dp values with -1.

// Modulo for the answer to prevent integer overflow issues.

// dp[i][k] will store the number of ways to end up at location 'i' with 'k' fuel left.

return memoizationCache[currentIndex][remainingFuel] = routeCount;

private int dfs(int currentIndex, int remainingFuel) {

// Return cached result if already computed.

if (nextIndex != currentIndex) {

int numLocations = locations.size();

int dp[numLocations][fuel + 1];

# Start DFS traversal from the start location with the initial amount of fuel

mod = 10\*\*9 + 7 # Define the modulo value (10^9 + 7)

if next\_location\_index != current\_location\_index:

# If remaining fuel is not enough to reach the destination, return 0

# Explore routes from the current location to every other location

next\_move\_cost = abs(locations[current\_location\_index] - location)

routes\_count %= mod # Ensure the result is within the modulo limit

# Recursively call dfs for the next location with updated remaining fuel

# Add the result to the routes count while handling the modulo operation

routes\_count += dfs(next\_location\_index, remaining\_fuel - next\_move\_cost)

# Use memoization to store results of subproblems

if remaining\_fuel < abs(locations[current\_location\_index] - locations[finish]):</pre> 12 13 return 0 14 # Initialize answer with 1 if at the finish location, otherwise with 0 16 routes\_count = 1 if current\_location\_index == finish else 0 17

Java Solution class Solution { private int[] locations; private int finishLocationIndex; private int locationCount; private Integer[][] memoizationCache; private final int MOD = (int) 1e9 + 7;// This function counts the number of ways to go from 'start' to 'finish' with given 'fuel'.

#### // Calculate remaining fuel after move and recursively call dfs. 35 int fuelCost = Math.abs(locations[currentIndex] - locations[nextIndex]); 36 37 routeCount = (routeCount + dfs(nextIndex, remainingFuel - fuelCost)) % MOD; 38 39 40

C++ Solution

1 #include <vector>

6 class Solution {

public:

9

10

11

12

13

14

15

2 #include <cstring>

#include <functional>

using namespace std;

```
16
             const int modulo = 1e9 + 7;
 17
 18
             // Define memoized depth-first search function using lambda and recursion.
             // The function calculates the number of routes to end up at index 'i' with 'k' fuel remaining.
 19
 20
             function<int(int, int)> dfs = [&](int i, int k) -> int {
 21
                 // Base case: If not enough fuel to reach the destination, return 0.
 22
                 if (k < abs(locations[i] - locations[finish])) {</pre>
 23
                     return 0;
 24
 25
 26
                 // If we have already computed this state, return its value.
                 if (dp[i][k] != -1) {
 27
 28
                     return dp[i][k];
 29
 30
                 // Count the current position if it's the destination.
 31
 32
                 int count = i == finish;
 33
 34
                 // Try to move to every other position and accumulate the count.
 35
                 for (int j = 0; j < numLocations; ++j) {</pre>
 36
                     if (j != i) {
                         // Recursively call dfs for the new location with decremented fuel.
 37
                         count = (count + dfs(j, k - abs(locations[i] - locations[j]))) % modulo;
 38
 39
 40
 41
 42
                 // Save and return the computed number of ways for this state.
                 return dp[i][k] = count;
 43
 44
             };
 45
 46
             // Calculate the number of routes starting from the start location with given fuel.
 47
             return dfs(start, fuel);
 48
 49
    };
 50
Typescript Solution
   function countRoutes(locations: number[], start: number, finish: number, fuel: number): number {
       const locationCount = locations.length;
       // Initialize a memoization array to store intermediate results,
       // with all initial values set to -1 indicating uncomputed states.
       const memo = Array.from({ length: locationCount }, () => Array(fuel + 1).fill(-1));
       // Define a constant for modulo operation to avoid integer overflow.
       const modulo = 1e9 + 7;
8
       // Depth-First Search function to count routes with remaining fuel 'remainingFuel' from 'currentIndex'.
9
       const dfs = (currentIndex: number, remainingFuel: number): number => {
10
           // If not enough fuel to reach the 'finish', return 0 as it's not a valid route.
11
12
           if (remainingFuel < Math.abs(locations[currentIndex] - locations[finish])) {</pre>
13
               return 0;
14
           // If the answer is memoized, return it to avoid re-computation.
15
           if (memo[currentIndex][remainingFuel] !== -1) {
16
```

# 23 if (nextIndex !== currentIndex) {

return dfs(start, fuel);

Time and Space Complexity

24

25

26

27

28

29

30

31

32

33

34

36

35 }

**}**;

return memo[currentIndex][remainingFuel]; 17 18 19 // Initialize answer with 1 if 'currentIndex' is the 'finish', since it's a valid route. 20 let routeCount = currentIndex === finish ? 1 : 0; 21 // Explore neighbours except for the current one. for (let nextIndex = 0; nextIndex < locationCount; ++nextIndex) {</pre>

// Accumulate answer with the count of routes from the 'nextIndex' with the reduced fuel.

const requiredFuel = Math.abs(locations[currentIndex] - locations[nextIndex]);

// Store the computed result in the memo array and return it.

// Start the DFS from the 'start' location with the initial amount of fuel.

return (memo[currentIndex][remainingFuel] = routeCount);

routeCount = (routeCount + dfs(nextIndex, remainingFuel - requiredFuel)) % modulo;

The time complexity of the code is  $0(n^2 * m)$  where n is the number of locations and m is the amount of fuel provided. This is because, for every recursive call to dfs, there's a potential for iterating over all n locations (except the current one), and this can happen at most m times due to the iterative decrement of fuel at each state. Multiplying these together, we end up with n^2 \* m considering every state (i, k) for each location i and fuel k.

The space complexity of the code is O(n \* m) due to the memoization used in the dfs function. This memoization stores the results of each unique call to dfs(i, k). Since i can take n different values and k can take up to m different values, the cache could potentially store n \* m distinct states.