1575. Count All Possible Routes

Hard | Memoization | Array | Dynamic Programming — Leetcode Link

Problem Description

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.

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.

The rules of the game are:

- You can travel from any city `i` to any different city `j`.
- Traveling from city `i` to city `j` consumes `|locations[i] - locations[j]|` units of fuel (`|x|` denotes the absolute value of `x`). You 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 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 the destination if you are already there).

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`.

5. We use memoization to store already calculated states (representing unique combinations of the current city and the remaining 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 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 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-up. However, that approach is more complex and requires a deeper understanding of DP to implement correctly.

Solution Approach

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 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.

3. **Base Cases**:
   - If the remaining fuel `k` is less than the absolute difference in location between city `i` and the destination city `finish` (i.e., `k < abs(locations[i] - locations[finish])`), the function returns `0`, indicating that the destination cannot be reached from this state.
   - 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.
4. **Recursive Exploration**:
   - We iterate through all other possible cities (represented by index `j`), and if `j != i`, we calculate the fuel required to travel from city `i` to city `j`.
   - 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.
5. **Returning the Result**:
   - The function finally returns `ans`, which is the total number of routes from city `i` with fuel `k` to the destination.
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 the given amount of fuel.

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

Both DFS with memoization and the DP approach use time complexity `O(n^2 * m)` and space complexity `O(n * m)` where `n` is the number of cities and `m` is the amount of fuel.

Example Walkthrough

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

Suppose we have the following input:

- `locations` array: [4, 3, 1]
- `start`: 0 (the starting city is at position 4)
- `finish`: 2 (the destination city is at position 1)
- `fuel`: 4 units

Using the solution approach:

1. **Initialization**: We start by initializing our memoization cache (not shown in this example). Then we begin our algorithm at city index 0 (which is at location 4) with 4 fuel.

2. **Recursive Function `dfs(i, k)`**: Our first call will be `dfs(0, 4)`.

3. **Base Cases**: We check our base cases:
   - `i` is not equal to `finish`, so we do not add 1 to our route count.
   - We have enough fuel (more than the distance from our current city to `finish`), so we do not return 0.
4. **Recursive Exploration**:
   - 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)`.
     - 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)`.
5. Exploring `dfs(1, 3)`:
   - Check base case: Not at destination, and have enough fuel.
   - Recursive exploration from city index 1:
     - To city index 0 (already visited, not destination): `dfs(0, 2)`.
     - To city index 2 (destination): direct fuel cost is `|3 - 1| = 2`. Since we have 3 fuel, we can reach the destination, so `dfs(2, 1)` from here adds 1 to our routes.
6. Exploring `dfs(2, 1)`:
   - Check base case: We're at the destination, so this recursive path adds 1 to our route count.
7. Exploring `dfs(0, 2)`:
   - Check base case: Not at destination, and have enough fuel.
   - Recursive exploration from city index 0:
     - To city index 1: `dfs(1, 1)`.
     - 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.
8. Exploring `dfs(1, 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.
9. **Adding up the routes**:
   - 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 exact paths are 0→2 and 0→1→2).
10. **Returning the Result**:
    - 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 from dfs(2, 1)) + (1 from dfs(1, 3) -> dfs(2, 1))`. Thus, we conclude `dfs(0, 4)` = 2.

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 the total count.

Python Solution

```python
from typing import List
from functools import cache

class Solution:
    def countRoutes(
        self, locations: List[int], start: int, finish: int, fuel: int
    ) -> int:
        # Use memoization to store results of subproblems
        @cache
        def dfs(current_location_index: int, remaining_fuel: int) -> int:
            # If remaining fuel is not enough to reach the destination, return 0
            if remaining_fuel < abs(locations[current_location_index] - locations[finish]):
                return 0

            # Initialize answer with 1 if at the finish location, otherwise with 0
            routes_count = 1 if current_location_index == finish else 0

            # Explore routes from the current location to every other location
            for next_location_index, location in enumerate(locations):
                if next_location_index != current_location_index:
                    # Calculate the fuel cost for the next move
                    next_move_cost = abs(locations[current_location_index] - location)

                    # 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)
                    routes_count %= mod  # Ensure the result is within the modulo limit

            # Return the total number of routes from the current location
            return routes_count

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

        # Start DFS traversal from the start location with the initial amount of fuel
        return dfs(start, fuel)

# Example of usage
# sol = Solution()
# routes_count = sol.countRoutes([2, 3, 6, 8, 4], 1, 3, 5)
# print(routes_count)  # Output will depend on the inputs provided

```

Java Solution

```java
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'.
    public int countRoutes(int[] locations, int start, int finish, int fuel) {
        locationCount = locations.length;
        this.locations = locations;
        this.finishLocationIndex = finish;
        memoizationCache = new Integer[locationCount][fuel + 1];
        return dfs(start, fuel);
    }

    // Helper function using Depth First Search (DFS) to compute the number of routes.
    private int dfs(int currentIndex, int remainingFuel) {
        // If the remaining fuel is not enough to go from currentIndex to finish, return 0.
        if (remainingFuel < Math.abs(locations[currentIndex] - locations[finishLocationIndex])) {
            return 0;
        }

        // Return cached result if already computed.
        if (memoizationCache[currentIndex][remainingFuel] != null) {
            return memoizationCache[currentIndex][remainingFuel];
        }

        // If the current location is the finish location, start with a count of 1, otherwise 0.
        int routeCount = currentIndex == finishLocationIndex ? 1 : 0;

        // Try all other locations using the remaining fuel and accumulate the routes count.
        for (int nextIndex = 0; nextIndex < locationCount; ++nextIndex) {
            if (nextIndex != currentIndex) {
                // Calculate remaining fuel after move and recursively call dfs.
                int fuelCost = Math.abs(locations[currentIndex] - locations[nextIndex]);
                routeCount = (routeCount + dfs(nextIndex, remainingFuel - fuelCost)) % MOD;
            }
        }

        // Store computed result in cache (memoization) and return the result.
        return memoizationCache[currentIndex][remainingFuel] = routeCount;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <cstring>
#include <functional>
using namespace std;

class Solution {
public:
    int countRoutes(vector<int>& locations, int start, int finish, int fuel) {
        int numLocations = locations.size();

        // dp[i][k] will store the number of ways to end up at location 'i' with 'k' fuel left.
        int dp[numLocations][fuel + 1];
        memset(dp, -1, sizeof(dp)); // Initialize all dp values with -1.

        // Modulo for the answer to prevent integer overflow issues.
        const int modulo = 1e9 + 7;

        // 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.
        function<int(int, int)> dfs = [&](int i, int k) -> int {
            // Base case: If not enough fuel to reach the destination, return 0.
            if (k < abs(locations[i] - locations[finish])) {
                return 0;
            }

            // If we have already computed this state, return its value.
            if (dp[i][k] != -1) {
                return dp[i][k];
            }

            // Count the current position if it's the destination.
            int count = i == finish;

            // Try to move to every other position and accumulate the count.
            for (int j = 0; j < numLocations; ++j) {
                if (j != i) {
                    // Recursively call dfs for the new location with decremented fuel.
                    count = (count + dfs(j, k - abs(locations[i] - locations[j]))) % modulo;
                }
            }

            // Save and return the computed number of ways for this state.
            return dp[i][k] = count;
        };

        // Calculate the number of routes starting from the start location with given fuel.
        return dfs(start, fuel);
    }
};

```

Typescript Solution

```typescript
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;

    // Depth-First Search function to count routes with remaining fuel 'remainingFuel' from 'currentIndex'.
    const dfs = (currentIndex: number, remainingFuel: number): number => {
        // If not enough fuel to reach the 'finish', return 0 as it's not a valid route.
        if (remainingFuel < Math.abs(locations[currentIndex] - locations[finish])) {
            return 0;
        }
        // If the answer is memoized, return it to avoid re-computation.
        if (memo[currentIndex][remainingFuel] !== -1) {
            return memo[currentIndex][remainingFuel];
        }
        // Initialize answer with 1 if 'currentIndex' is the 'finish', since it's a valid route.
        let routeCount = currentIndex === finish ? 1 : 0;
        // Explore neighbours except for the current one.
        for (let nextIndex = 0; nextIndex < locationCount; ++nextIndex) {
            if (nextIndex !== currentIndex) {
                const requiredFuel = Math.abs(locations[currentIndex] - locations[nextIndex]);
                // Accumulate answer with the count of routes from the 'nextIndex' with the reduced fuel.
                routeCount = (routeCount + dfs(nextIndex, remainingFuel - requiredFuel)) % modulo;
            }
        }
        // Store the computed result in the memo array and return it.
        return (memo[currentIndex][remainingFuel] = routeCount);
    };

    // Start the DFS from the 'start' location with the initial amount of fuel.
    return dfs(start, fuel);
}

```

Time and Space Complexity

The time complexity of the code is `O(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.