<https://leetcode.com/problems/minimum-number-of-refueling-stops/description/>

A car travels from a starting position to a destination which is target miles east of the starting position.

There are gas stations along the way. The gas stations are represented as an array stations where stations[i] = [positioni, fueli] indicates that the ith gas station is positioni miles east of the starting position and has fueli liters of gas.

The car starts with an infinite tank of gas, which initially has startFuel liters of fuel in it. It uses one liter of gas per one mile that it drives. When the car reaches a gas station, it may stop and refuel, transferring all the gas from the station into the car.

Return the minimum number of refueling stops the car must make in order to reach its destination. If it cannot reach the destination, return -1.

Note that if the car reaches a gas station with 0 fuel left, the car can still refuel there. If the car reaches the destination with 0 fuel left, it is still considered to have arrived.

**Example 1:**

**Input:** target = 1, startFuel = 1, stations = []

**Output:** 0

**Explanation:** We can reach the target without refueling.

**Example 2:**

**Input:** target = 100, startFuel = 1, stations = [[10,100]]

**Output:** -1

**Explanation:** We can not reach the target (or even the first gas station).

**Example 3:**

**Input:** target = 100, startFuel = 10, stations = [[10,60],[20,30],[30,30],[60,40]]

**Output:** 2

**Explanation:** We start with 10 liters of fuel.We drive to position 10, expending 10 liters of fuel.

We refuel from 0 liters to 60 liters of gas.

Then, we drive from position 10 to position 60 (expending 50 liters of fuel),

and refuel from 10 liters to 50 liters of gas. We then drive to and reach the target.

We made 2 refueling stops along the way, so we return 2.

**Constraints:**

1. 1 <= target, startFuel <= 10^9
2. 0 <= stations.length <= 500
3. 1 <= positioni < positioni+1 < target
4. 1 <= fueli < 10^9

**Attempt 1: 2024-08-17**

**Solution 1: Native DFS (30 min, TLE)**

**Style 1: For loop style (114/198)**

**Three parameters version (curFuel, index and remainDistance (or distanceBeforeCurSection))**

class Solution {

public int minRefuelStops(int target, int startFuel, int[][] stations) {

// Set maximum stops as (stations.length + 1) not Integer.MAX\_VALUE

// to avoid stackoverflow, plus one because need at least larger

// than potential actual maximum stops which equals to total stations

int stops = stations.length + 1;

stops = helper(startFuel, target, 0, 0, stations);

return stops == stations.length + 1 ? -1 : stops;

}

private int helper(int curFuel, int remainDistance, int index, int distanceBeforeCurSection, int[][] stations) {

// Base case: Since current fuel more than remain distance, no more stops need

if(curFuel >= remainDistance) {

return 0;

}

int minStops = stations.length + 1;

for(int i = index; i < stations.length; i++) {

// If current fuel not able to reach current station, return max stops

int curSectionDistance = stations[i][0] - distanceBeforeCurSection;

if(curFuel >= curSectionDistance) {

minStops = Math.min(minStops, 1 + helper(curFuel - curSectionDistance + stations[i][1],

remainDistance - curSectionDistance, i + 1, stations[i][0], stations));

}

}

return minStops;

}

}

**Style 2: Standard 0-1 knapsack style (pick + not pick) (114/198)**

**Three parameters version (curFuel, index and remainDistance (or distanceBeforeCurSection))**

class Solution {

    public int minRefuelStops(int target, int startFuel, int[][] stations) {

        // Set maximum stops as (stations.length + 1) not Integer.MAX\_VALUE

        // to avoid stackoverflow, plus one because need at least larger

        // than potential actual maximum stops which equals to total stations

        int stops = stations.length + 1;

        stops = helper(startFuel, target, 0, 0, stations);

        return stops == stations.length + 1 ? -1 : stops;

    }

    private int helper(int curFuel, int remainDistance, int index, int distanceBeforeCurSection, int[][] stations) {

        // Base case 1: Since current fuel more than remain distance, no more stops need

        if(curFuel >= remainDistance) {

            return 0;

        }

        // Base case 2: If reach the last station but not get enough fuel to finish remain distance

        if(index == stations.length) {

            return stations.length + 1;

        }

        // Base case 3: If current fuel not able to reach current station, return max stops

        int curSectionDistance = stations[index][0] - distanceBeforeCurSection;

        if(curFuel < curSectionDistance) {

            return stations.length + 1;

        }

        // Option 1: Skip the current station

        int skip = helper(curFuel, remainDistance, index + 1, distanceBeforeCurSection, stations);

        // Option 2: Refuel at the current station

        int refuel = 1 + helper(curFuel - curSectionDistance + stations[index][1], remainDistance - curSectionDistance, index + 1, stations[index][0], stations);

        return Math.min(skip, refuel);

    }

}

**Refer to**

<https://leetcode.com/problems/minimum-number-of-refueling-stops/solutions/2452236/c-recursion-memoization-max-heap/>

**Three parameters version**

// Recursion - TLE (Time - 2^n, Space - 0(n))

int helper(int target, int fuel, vector<vector<int>>& s, int i, int prev\_stop, vector<vector<vector<int>>> &dp){

if(target <= fuel) return 0;

if(i == s.size()) return INT\_MAX-1;

int distance = s[i][0] - prev\_stop;

if(fuel < distance) return INT\_MAX-1;

int take\_fuel = 1 + helper(target - distance, fuel + s[i][1] - distance, s, i+1, s[i][0], dp);

int not\_take\_fuel = helper(target, fuel, s, i+1, prev\_stop, dp);

return min(take\_fuel, not\_take\_fuel);

}

int minRefuelStops(int target, int startFuel, vector<vector<int>>& stations) {

int ans = helper(target, startFuel, stations, 0, 0, dp);

if(ans == INT\_MAX-1) return -1;

return ans;

}

**Wrong Solution which not consider fuel consumption between current station to next station below:**

**Refer to chatGPT**

**There is a wrong solution which does not consider the fuel consumption between the current station and the next candidate station, which makes it logically different from correct solution. This is an important distinction, and it highlights a key difference in how the wrong and correct solution handle the problem.**

public class Solution {

public int minRefuelStops(int target, int startFuel, int[][] stations) {

int n = stations.length;

int result = helper(target, startFuel, stations, 0, n);

return result == Integer.MAX\_VALUE ? -1 : result;

}

private int helper(int target, int fuel, int[][] stations, int index, int n) {

if (fuel >= target) {

return 0;

}

if (index == n || fuel < stations[index][0]) {

return Integer.MAX\_VALUE;

}

int skip = helper(target, fuel, stations, index + 1, n);

int refuel = helper(target, fuel + stations[index][1], stations, index + 1, n);

if (refuel != Integer.MAX\_VALUE) {

refuel++;

}

return Math.min(skip, refuel);

}

}

**Key Differences:**

1. Fuel Consumption Between Stations:
   1. **Correct Solution: Explicitly accounts for the fuel consumed when moving from one station to the next by calculating the distance between them and adjusting the fuel accordingly.**
   2. **Wrong Solution: Assumes that the car can directly reach the next station without explicitly deducting the fuel consumed to get there. This is a significant omission, as it overlooks the critical step of subtracting the distance from the current fuel level.**

**Adjusting the Wrong Solution to Match the Logic:**

To make the second solution logically equivalent to your first solution, we need to incorporate the fuel consumption when moving from the current station to the next. Here's how we can do it:

**Corrected Solution:**

public class Solution {

public int minRefuelStops(int target, int startFuel, int[][] stations) {

int n = stations.length;

int result = helper(target, startFuel, stations, 0, 0, n);

return result == Integer.MAX\_VALUE ? -1 : result;

}

private int helper(int target, int fuel, int[][] stations, int index, int prevDistance, int n) {

if (fuel >= target - prevDistance) {

return 0;

}

if (index == n) {

return Integer.MAX\_VALUE;

}

int distanceToNextStation = stations[index][0] - prevDistance;

if (fuel < distanceToNextStation) {

return Integer.MAX\_VALUE;

}

// Option 1: Skip the current station

int skip = helper(target, fuel, stations, index + 1, prevDistance, n);

// Option 2: Refuel at the current station

int refuel = helper(target, fuel - distanceToNextStation + stations[index][1], stations, index + 1, stations[index][0], n);

if (refuel != Integer.MAX\_VALUE) {

refuel++;

}

return Math.min(skip, refuel);

}

}

**Explanation of Changes:**

1. Fuel Adjustment Between Stations:
   1. The code now calculates the distanceToNextStation as stations[index][0] - prevDistance.
   2. The available fuel is checked against this distance to ensure the car can reach the next station.
   3. If the car chooses to refuel, the fuel is adjusted by subtracting the distance and adding the fuel from the station: fuel - distanceToNextStation + stations[index][1].
2. Tracking Previous Distance:
   1. The prevDistance parameter is used to keep track of the distance covered so far. This is crucial for calculating the distance to the next station.
3. Handling Base Cases:
   1. The base cases are updated to account for whether the fuel is sufficient to reach the target after adjusting for the distance to the next station.

**Conclusion:**

**This corrected solution now properly accounts for the fuel consumption between stations, making it logically equivalent to your original solution. The crucial difference is that it now includes the necessary calculations to ensure that the car can actually reach the next station, rather than assuming it can reach it directly.**

**Solution 2: DFS + Memoization (30 min, MLE)**

**Style 1: For loop style + Memoization (109/198)**

**Three parameters version (curFuel, index and remainDistance (or distanceBeforeCurSection))**

class Solution {

    public int minRefuelStops(int target, int startFuel, int[][] stations) {

        Map<String, Integer> memo = new HashMap<>();

        // Set maximum stops as (stations.length + 1) not Integer.MAX\_VALUE

        // to avoid stackoverflow, plus one because need at least larger

        // than potential actual maximum stops which equals to total stations

        int stops = stations.length + 1;

        stops = helper(startFuel, target, 0, 0, stations, memo);

        return stops == stations.length + 1 ? -1 : stops;

    }

    private int helper(int curFuel, int remainDistance, int index, int distanceBeforeCurSection, int[][] stations, Map<String, Integer> memo) {

        // Base case 1: Since current fuel more than remain distance, no more stops need

        if(curFuel >= remainDistance) {

            return 0;

        }

        String key = curFuel + "\_" + index + "\_" + remainDistance;

        if(memo.containsKey(key)) {

            return memo.get(key);

        }

        int minStops = stations.length + 1;

        for(int i = index; i < stations.length; i++) {

            // If current fuel not able to reach current station, return max stops

            int curSectionDistance = stations[i][0] - distanceBeforeCurSection;

            if(curFuel >= curSectionDistance) {

                minStops = Math.min(minStops, 1 + helper(curFuel - curSectionDistance + stations[i][1],

                    remainDistance - curSectionDistance, i + 1, stations[i][0], stations, memo));

            }

        }

        memo.put(key, minStops);

        return minStops;

    }

}

**Style 2: Standard 0-1 knapsack style (pick + not pick) + Integer[ ][ ][ ] Memoization**

**Memory Limit Exceeded when input as [1000000000, 1000000000], 14/198 test case passed**

**For all unique identification status requires three dimensions (curFuel, index, remainDistance (or distanceBeforeCurSection))**

class Solution {

    public int minRefuelStops(int target, int startFuel, int[][] stations) {

        // Set maximum stops as (stations.length + 1) not Integer.MAX\_VALUE

        // to avoid stackoverflow, plus one because need at least larger

        // than potential actual maximum stops which equals to total stations

        int stops = stations.length + 1;

        Integer[][][] memo = new Integer[stops][target + 1][target + 1];

        stops = helper(startFuel, target, 0, 0, stations, memo);

        return stops == stations.length + 1 ? -1 : stops;

    }

    private int helper(int curFuel, int remainDistance, int index, int distanceBeforeCurSection, int[][] stations, Integer[][][] memo) {

        // Base case 1: Since current fuel more than remain distance, no more stops need

        if(curFuel >= remainDistance) {

            return 0;

        }

        // Base case 2: If reach the last station but not get enough fuel to finish remain distance

        if(index == stations.length) {

            return stations.length + 1;

        }

        // Base case 3: If current fuel not able to reach current station, return max stops

        int curSectionDistance = stations[index][0] - distanceBeforeCurSection;

        if(curFuel < curSectionDistance) {

            return stations.length + 1;

        }

        if(memo[index][curFuel][remainDistance] != null) {

            return memo[index][curFuel][remainDistance];

        }

        // Option 1: Skip the current station

        int skip = helper(curFuel, remainDistance, index + 1, distanceBeforeCurSection, stations, memo);

        // Option 2: Refuel at the current station

        int refuel = 1 + helper(curFuel - curSectionDistance + stations[index][1], remainDistance - curSectionDistance, index + 1, stations[index][0], stations, memo);

        return memo[index][curFuel][remainDistance] = Math.min(skip, refuel);

    }

}

**Refer to**

<https://leetcode.com/problems/minimum-number-of-refueling-stops/solutions/2452236/c-recursion-memoization-max-heap/>

**Three parameters version**

// Memoization - Out of memory (Time - O(n^2), Space - O( target \* target \* n)

int helper(int target, int fuel, vector<vector<int>>& s, int i, int prev\_stop, vector<vector<vector<int>>> &dp){

if(target <= fuel) return 0;

if(i == s.size()) return INT\_MAX-1;

int distance = s[i][0] - prev\_stop;

if(fuel < distance ) return INT\_MAX-1;

if(dp[target][fuel][i] != -1) return dp[target][fuel][i];

int take\_fuel = 1 + helper(target - distance, fuel + s[i][1] - distance, s, i+1, s[i][0], dp);

int not\_take\_fuel = helper(target, fuel, s, i+1, prev\_stop, dp);

return dp[target][fuel][i] = min(take\_fuel, not\_take\_fuel);

}

int minRefuelStops(int target, int startFuel, vector<vector<int>>& stations) {

int n = stations.size();

vector<vector<vector<int>>> dp(target+1, vector<vector<int>>(target+1, vector<int>(n+1, -1)));

int ans = helper(target, startFuel, stations, 0, 0, dp);

if(ans == INT\_MAX-1) return -1;

return ans;

}

**Style 3: Standard 0-1 knapsack style (pick + not pick) + Map<String, Integer> Memoization (103/198)**

**Three parameters version (curFuel, index and remainDistance (or distanceBeforeCurSection))**

class Solution {

    public int minRefuelStops(int target, int startFuel, int[][] stations) {

        Map<String, Integer> memo = new HashMap<>();

        // Set maximum stops as (stations.length + 1) not Integer.MAX\_VALUE

        // to avoid stackoverflow, plus one because need at least larger

        // than potential actual maximum stops which equals to total stations

        int stops = stations.length + 1;

        stops = helper(startFuel, target, 0, 0, stations, memo);

        return stops == stations.length + 1 ? -1 : stops;

    }

    private int helper(int curFuel, int remainDistance, int index, int distanceBeforeCurSection, int[][] stations, Map<String, Integer> memo) {

        // Base case 1: Since current fuel more than remain distance, no more stops need

        if(curFuel >= remainDistance) {

            return 0;

        }

        // Base case 2: If reach the last station but not get enough fuel to finish remain distance

        if(index == stations.length) {

            return stations.length + 1;

        }

        // Base case 3: If current fuel not able to reach current station, return max stops

        int curSectionDistance = stations[index][0] - distanceBeforeCurSection;

        if(curFuel < curSectionDistance) {

            return stations.length + 1;

        }

        // The key combination must include all three parameters (curFuel, index, remainDistance)

        // to guarantee its unique mapping, and if replace 'remainDistance' with 'distanceBeforeCurSection'

// it also works, since 'remainDistance' dynamically derived from 'distanceBeforeCurSection'

        String key = curFuel + "\_" + index + "\_" + remainDistance;

        if(memo.containsKey(key)) {

            return memo.get(key);

        }

        // Option 1: Skip the current station

        int skip = helper(curFuel, remainDistance, index + 1, distanceBeforeCurSection, stations, memo);

        // Option 2: Refuel at the current station

        int refuel = 1 + helper(curFuel - curSectionDistance + stations[index][1], remainDistance - curSectionDistance, index + 1, stations[index][0], stations, memo);

        int result = Math.min(skip, refuel);

        memo.put(key, result);

        return result;

    }

}

**Wrong Map<String, Integer> memoization solution, it errors out since only take two parameters (curFuel and index) as key, since the basement solution (Style 2: Standard 0-1 knapsack style (pick + not pick)) has three parameters (curFuel, index and remainDistance (or distanceBeforeCurSection))**

**when input as below it error out (97/198)**

**Input: target =1000, startFuel =299, stations =[[13,100],[25,117],[122,82],[189,123],[268,56],[382,214],[408,280],[421,272],[589,110],[899,4]]**

**Wrong Output: 3**

**Expected: 4**

class Solution {

    public int minRefuelStops(int target, int startFuel, int[][] stations) {

        Map<String, Integer> memo = new HashMap<>();

        // Set maximum stops as (stations.length + 1) not Integer.MAX\_VALUE

        // to avoid stackoverflow, plus one because need at least larger

        // than potential actual maximum stops which equals to total stations

        int stops = stations.length + 1;

        stops = helper(startFuel, target, 0, 0, stations, memo);

        return stops == stations.length + 1 ? -1 : stops;

    }

    private int helper(int curFuel, int remainDistance, int index, int distanceBeforeCurSection, int[][] stations, Map<String, Integer> memo) {

        // Base case 1: Since current fuel more than remain distance, no more stops need

        if(curFuel >= remainDistance) {

            return 0;

        }

        // Base case 2: If reach the last station but not get enough fuel to finish remain distance

        if(index == stations.length) {

            return stations.length + 1;

        }

        // Base case 3: If current fuel not able to reach current station, return max stops

        int curSectionDistance = stations[index][0] - distanceBeforeCurSection;

        if(curFuel < curSectionDistance) {

            return stations.length + 1;

        }

        String key = curFuel + "\_" + index;

        if(memo.containsKey(key)) {

            return memo.get(key);

        }

        // Option 1: Skip the current station

        int skip = helper(curFuel, remainDistance, index + 1, distanceBeforeCurSection, stations, memo);

        // Option 2: Refuel at the current station

        int refuel = 1 + helper(curFuel - curSectionDistance + stations[index][1], remainDistance - curSectionDistance, index + 1, stations[index][0], stations, memo);

        int result = Math.min(skip, refuel);

        memo.put(key, result);

        return result;

    }

}

**The issue with your solution arises from the way you handle the remaining distance after refueling or skipping a station. Specifically, the memoization key (key = curFuel + "\_" + index) does not consider all factors that affect the state, such as the remaining distance. This can lead to incorrect results when certain states are revisited with different conditions.**

**Problem:**

**The key you are using for memoization (curFuel + "\_" + index) is not sufficient to uniquely identify the state, as it doesn't include the remainDistance or distanceBeforeCurSection. Therefore, when a state is revisited, the algorithm might incorrectly assume that it has already computed the optimal solution for that state, even though the actual conditions (such as remaining distance) might be different.**

**Solution:**

**To fix this, you should modify the memoization key to include all relevant state variables that can affect the outcome. Specifically, you need to include remainDistance or distanceBeforeCurSection in the memoization key.**

**Key Changes:**

1. Memoization Key:
   1. The memoization key is now constructed using all relevant state variables: curFuel, index, and remainDistance. This ensures that different states are uniquely identified in the memoization map.
2. Memoization Map:
   1. The memo map now stores results based on this comprehensive key, ensuring that overlapping subproblems are correctly identified.

**Explanation:**

By including remainDistance or distanceBeforeCurSection in the memoization key, we ensure that different scenarios (e.g., with different remaining distances) are treated as distinct states. This prevents incorrect reuse of previously computed results when the conditions have changed.

**Test:**

After applying this change, the code should return the correct output for the given input:

1. Input: target = 1000, startFuel = 299, stations = [[13,100],[25,117],[122,82],[189,123],[268,56],[382,214],[408,280],[421,272],[589,110],[899,4]]
2. Expected Output: 4

**Solution 3: 2D DP (60 min)**

class Solution {

    public int minRefuelStops(int target, int startFuel, int[][] stations) {

        // Base case: test out by input 1,1,[]

        if(startFuel >= target) {

            return 0;

        }

        int n = stations.length;

        // dp[i][j] stores the maximum distance reachable with j refuels using the first i stations.

        int[][] dp = new int[n + 1][n + 1];

        // Initialize dp[0][0] with startFuel (max distance with 0 refuels and 0 stations)

// Note: If we use for(int i = 0; i <= n; i++) {dp[i][0] = startFuel;} then inner

// for loop we can start j from 1 instead of 0 and no need j > 0 in condition

// 'if(j > 0 && dp[i - 1][j - 1] >= stations[i - 1][0])'

        dp[0][0] = startFuel;

        // Max potential stops will be only n, initialize n + 1 equal to Integer.MAX\_VALUE

        int minStops = n + 1;

        for(int i = 1; i <= n; i++) {

            for(int j = 0; j <= i; j++) {

                // Not take ith station to refuel, which means in previous (i - 1)

                // stations need to pick up j stations to refuel, so how far (i - 1)

                // stations can reach can compare with former i stations to find max

                dp[i][j] = dp[i - 1][j];

                // Take ith station to refuel, which means in previous (i - 1) stations

                // need to pick up (j - 1) stations to refuel, the pre-requistion is

                // to make sure we able to reach ith station {stations[i - 1] is the ith

                // station represent in array, and stations[i - 1][0] indicates that

                // ith station miles east of the starting position} based on previous

                // status (how far it can reach) stored in dp[i - 1][j - 1]

                // Note: j > 0 test out by input 100,1,[[10,100]] ArrayIndexOutOfBoundsException

                if(j > 0 && dp[i - 1][j - 1] >= stations[i - 1][0]) {

                    dp[i][j] = Math.max(dp[i][j], dp[i - 1][j - 1] + stations[i - 1][1]);

                }

                if(dp[i][j] >= target) {

                    minStops = Math.min(minStops, j);

                }

            }

        }

        return minStops == n + 1 ? -1 : minStops;

    }

}

**Refer to chatGPT**

Converting the DFS solution to a 2D dynamic programming (DP) approach involves creating a DP table where each entry represents the minimum number of stops required to reach a certain distance with a given number of refuels.

**Steps to Convert DFS to 2D DP:**

1. Create a 2D DP table dp[i][j] where:
   1. i represents the index of the station.
   2. j represents the number of refuels.
   3. dp[i][j] stores the maximum distance that can be reached by refueling j times by the time you reach station i.
2. Initialization:
   1. Start with dp[0][0] = startFuel since with zero refuels, the farthest you can reach is your initial fuel.
3. Iterate Over Stations:
   1. For each station, consider both options: refueling at the station or skipping it. Update the DP table accordingly.
4. Check for Minimum Refuels:
   1. After processing all stations, find the minimum number of refuels needed to reach or exceed the target.

**2D DP Solution:**

class Solution {

public int minRefuelStops(int target, int startFuel, int[][] stations) {

int n = stations.length;

// dp[i][j] represents the max distance we can reach with j refuels using the first i stations.

long[][] dp = new long[n + 1][n + 1];

// Initialize dp[0][0] with startFuel (max distance with 0 refuels and 0 stations)

dp[0][0] = startFuel;

// Iterate over each station

for (int i = 1; i <= n; i++) {

int stationPos = stations[i - 1][0];

int stationFuel = stations[i - 1][1];

for (int j = 0; j <= i; j++) {

// Without refueling at station i-1

dp[i][j] = dp[i - 1][j];

// With refueling at station i-1, only if we could have reached station i-1 with j-1 refuels

if (j > 0 && dp[i - 1][j - 1] >= stationPos) {

dp[i][j] = Math.max(dp[i][j], dp[i - 1][j - 1] + stationFuel);

}

}

}

// Find the minimum number of refuels needed to reach or exceed the target

for (int j = 0; j <= n; j++) {

if (dp[n][j] >= target) {

return j;

}

}

return -1; // If it's not possible to reach the target

}

}

**Explanation:**

1. DP Array Setup:
   1. We use a 2D DP array dp[i][j] where i ranges from 0 to n (number of stations) and j ranges from 0 to n (possible refuels).
   2. dp[i][j] stores the maximum distance that can be reached with j refuels using the first i stations.
2. Initialization:
   1. dp[0][0] = startFuel: With zero refuels and zero stations, the maximum distance reachable is the initial fuel.
3. State Transition:
   1. For each station i, you decide whether to refuel or not:
      1. Without refueling: You carry forward the maximum distance from the previous station (dp[i][j] = dp[i - 1][j]).
      2. With refueling: If you could reach station i-1 with j-1 refuels (dp[i - 1][j - 1] >= stationPos), then consider refueling and update dp[i][j].
4. Result:
   1. After filling the DP table, check for the smallest j where dp[n][j] >= target, which gives the minimum refuels needed.
   2. If no such j exists, return -1.

**Comparison with the DFS Version:**

1. DFS Solution:
   1. The DFS approach recursively explores each possible sequence of refuels, which can be inefficient and may lead to overlapping subproblems.
2. 2D DP Solution:
   1. The 2D DP approach systematically builds up the solution from smaller subproblems, ensuring each refueling decision is considered optimally without redundant calculations.

This 2D DP solution is more structured and avoids the exponential time complexity that might occur with a DFS approach, making it more efficient for larger inputs.

**Solution 4: 1D DP (30 min)**

class Solution {

    public int minRefuelStops(int target, int startFuel, int[][] stations) {

        // Base case: test out by input 1,1,[]

        if(startFuel >= target) {

            return 0;

        }

        int n = stations.length;

        // dp[j] represents the max distance we can reach with j refuels.

        int[] dp = new int[n + 1];

        // Initialize dp[0] with startFuel (max distance with 0 refuels)

        dp[0] = startFuel;

        int minStops = n + 1;

        for(int i = 1; i <= n; i++) {

            for(int j = i; j >= 0; j--) {

                // Update dp array from the back to avoid overwriting

                // Since dp[i][j] relates to dp[i - 1][j] and dp[i - 1][j - 1],

                // if updating the compressed 1D DP array left -> right,

                // dp[j - 1] is updated before dp[j] with row i's dp[i][j - 1]

                // value, which replaced the target value dp[i - 1][j - 1]

                // if updating the compressed 1D DP array right -> left,

                // dp[j - 1] hasn't been udpated when calculating dp[j],

                // which remains the target value dp[i - 1][j - 1]

                // Note: Actually add 'j > 0' with 'j >= 0' in for loop condition

                // just for match its 2D DP solution pattern, we can directly

                // remove 'j > 0' and update 'j >= 0' to 'j >= 1' in for loop

                // condition instead to simplify

                if(j > 0 && dp[j - 1] >= stations[i - 1][0]) {

                    dp[j] = Math.max(dp[j], dp[j - 1] + stations[i - 1][1]);

                }

                if(dp[j] >= target) {

                    minStops = Math.min(minStops, j);

                }

            }

        }

        return minStops == n + 1 ? -1 : minStops;

    }

}

**Refer to chatGPT**

**To convert the 2D dynamic programming (DP) solution to a 1D DP solution, the idea is to recognize that each row of the DP table only depends on the previous row. Therefore, we can optimize space by using a single array instead of a 2D table.**

**Key Concepts:**

1. 2D DP: dp[i][j] stores the maximum distance reachable with j refuels using the first i stations.
2. 1D DP: We reduce the 2D array to a 1D array where dp[j] represents the maximum distance reachable with j refuels after processing all stations.

**Transition from 2D to 1D:**

1. **Instead of updating a new row in the DP table, we update the same row (1D array) from the back to avoid overwriting values that are still needed for the current iteration.**

**Steps:**

1. Initialization:
   1. Start with dp[0] = startFuel, which means with 0 refuels, the maximum distance reachable is the initial fuel.
2. Iterate Over Stations:
   1. For each station, iterate backward over the number of refuels (j) to ensure that each station's contribution is considered only once.
3. Check for Minimum Refuels:
   1. After processing all stations, find the minimum number of refuels needed to reach or exceed the target.

**1D DP Solution:**

class Solution {

public int minRefuelStops(int target, int startFuel, int[][] stations) {

int n = stations.length;

// dp[j] represents the max distance we can reach with j refuels.

long[] dp = new long[n + 1];

// Initialize dp[0] with startFuel (max distance with 0 refuels)

dp[0] = startFuel;

// Iterate over each station

for (int i = 0; i < n; i++) {

int stationPos = stations[i][0];

int stationFuel = stations[i][1];

// Update dp array from the back to avoid overwriting

for (int j = i; j >= 0; j--) {

// If we can reach this station with j refuels

if (dp[j] >= stationPos) {

// Consider refueling at this station

dp[j + 1] = Math.max(dp[j + 1], dp[j] + stationFuel);

}

}

}

// Find the minimum number of refuels needed to reach or exceed the target

for (int j = 0; j <= n; j++) {

if (dp[j] >= target) {

return j;

}

}

return -1; // If it's not possible to reach the target

}

}

**Explanation:**

1. DP Array Setup:
   1. We use a 1D DP array dp[j] where j ranges from 0 to n (number of possible refuels).
   2. dp[j] stores the maximum distance reachable with j refuels after considering all stations.
2. Initialization:
   1. dp[0] = startFuel: With zero refuels, the maximum distance reachable is the initial fuel.
3. State Transition:
   1. For each station i, iterate backward over the number of refuels (j):
      1. If you can reach the current station with j refuels (dp[j] >= stationPos), consider refueling and update dp[j + 1].
4. Result:
   1. After filling the DP array, check for the smallest j where dp[j] >= target, which gives the minimum refuels needed.
   2. If no such j exists, return -1.

**Benefits of 1D DP:**

1. Space Efficiency: The 1D DP solution significantly reduces space complexity from O(n^2) to O(n) while maintaining the same time complexity.
2. Performance: The 1D approach is often faster because of reduced memory access and better cache usage.

This conversion optimizes the solution and is particularly useful when dealing with large inputs.

**Below are old attempts but including valuable details and thoughts:**

Refer to

<https://leetcode.com/problems/minimum-number-of-refueling-stops/discuss/613853/Python-5-solutions-gradually-optimizing-from-Naive-DFS-to-O(n)-space-DP>

**Solution 1: Native DFS with for loops subset implementation (TLE)**

**Python version**

class Solution(object):

def minRefuelStops(self, target, startFuel, stations):

"""

:type target: int

:type startFuel: int

:type stations: List[List[int]]

:rtype: int

"""

# 1) Naive DFS

self.full\_target = target

def dfs(curFuel, start, target):

if curFuel >= target:

return 0

rst = sys.maxsize

for i in xrange(start, len(stations)):

dis, fuel = stations[i][0] - (self.full\_target - target), stations[i][1]

if curFuel - dis >= 0:

rst = min(rst, dfs(curFuel - dis + fuel, i + 1, target - dis) + 1)

return rst

stops = dfs(startFuel, 0, target)

return stops if stops != sys.maxsize else -1

**Java version**

class Solution {

public int minRefuelStops(int target, int startFuel, int[][] stations) {

// Set maximum stops as (stations.length + 1) not Integer.MAX\_VALUE

// to avoid stackoverflow, plus one because need at least larger

// than potential actual maximum stops which equals to total stations

int stops = stations.length + 1;

stops = helper(startFuel, 0, target, target, stations);

return stops != stations.length + 1 ? stops : -1;

}

public int helper(int curFuel, int start, int remain, int original\_target, int[][] stations) {

// Base case: since current fuel more than remain distance, no more stops need

if(curFuel >= remain) {

return 0;

}

int min\_stops = stations.length + 1;

for(int i = start; i < stations.length; i++) {

int passed\_distance = original\_target - remain;

int distance\_to\_ith\_station = stations[i][0] - passed\_distance;

int fuel = stations[i][1];

if(curFuel - distance\_to\_ith\_station >= 0) {

min\_stops = Math.min(min\_stops, helper(curFuel - distance\_to\_ith\_station + fuel, i + 1, remain - distance\_to\_ith\_station, original\_target, stations) + 1);

}

}

return min\_stops;

}

}

**Solution 2: DFS with for loops subset implementation + memorization (Memory Limit Exceeded)**

**Python version**

# 2) DFS with for loops subset implementation + memorization

self.full\_target = target

mem = dict()

def dfs(curFuel, start, target):

if curFuel >= target:

return 0

if (curFuel, start, target) in mem:

return mem[(curFuel, start, target)]

rst = sys.maxsize

for i in xrange(start, len(stations)):

dis, fuel = stations[i][0] - (self.full\_target - target), stations[i][1]

if curFuel - dis >= 0:

rst = min(rst, dfs(curFuel - dis + fuel, i + 1, target - dis) + 1)

mem[(curFuel, start, target)] = rst

return mem[(curFuel, start, target)]

stops = dfs(startFuel, 0, target)

return stops if stops != sys.maxsize else -1

**Refer to**

<https://leetcode.com/problems/minimum-number-of-refueling-stops/discuss/149958/Does-anyone-have-a-Top-Down-DP-version-of-this>

**I got the recursive solution, but I am getting memory limit exceeded whenever I try to memoize.**

class Solution {

int N;

int [][] stations;

Map<Integer, Integer> map = new HashMap<>();

int [][] memo;

public int minRefuelStops(int target, int startFuel, int[][] stations) {

this.N = stations.length;

this.stations = stations;

for(int i = 0; i < stations.length; ++i){

int [] s = stations[i];

map.put(s[0], i);

}

map.put(0, -1);

memo = new int[stations.length + 1][10000000 + 1];

for(int [] arr : memo) Arrays.fill(arr, -1);

int sol = helper(0, startFuel, target);

return (sol >= N + 1) ? -1 : sol;

}

public int helper(int station, int fuel, int target){

if(memo[station][fuel] != -1) return memo[station][fuel];

if(station + fuel >= target) return 0;

if(fuel <= 0) return N + 1;

int cur = N + 1;

for(int i = map.get(station) + 1; i < stations.length; ++i){

int [] current = stations[i];

if(fuel - (current[0] - station) < 0) continue;

cur = Math.min(cur, 1 + helper(current[0], fuel - (current[0] - station) + current[1], target));

}

memo[station][fuel] = cur;

return cur;

}

}

**This does not work I am getting memory and time limit exceeded. Does anyone have an idea how to fix this?**

**Java version**

**Style 1: memo[i][j] means minimum stops needed for i liters fuel at jth station**

**Memory Limit Exceeded when input as [1, 1], 0 test case passed**

class Solution {

public int minRefuelStops(int target, int startFuel, int[][] stations) {

// Need to memoize tuple of (current fuel, position of station)

// since these two values keep changing during you recursive calls

// 1 <= target, startFuel <= 10^9 --> need (target + 1)

// 1 <= fueli < 10^9 --> need (10^9)

// memo[i][j] means minimum stops needed for i liters fuel at jth station

Integer[][] memo = new Integer[1000000000][target + 1];

int stops = stations.length + 1;

stops = helper(startFuel, 0, target, target, stations, memo);

return stops != stations.length + 1 ? stops : -1;

}

public int helper(int curFuel, int start, int remain, int original\_target, int[][] stations, Integer[][] memo) {

if(curFuel >= remain) {

return 0;

}

if(memo[curFuel][start] != null) {

return memo[curFuel][start];

}

int min\_stops = stations.length + 1;

for(int i = start; i < stations.length; i++) {

int passed\_distance = original\_target - remain;

int distance\_to\_ith\_station = stations[i][0] - passed\_distance;

int fuel = stations[i][1];

if(curFuel - distance\_to\_ith\_station >= 0) {

min\_stops = Math.min(min\_stops, helper(curFuel - distance\_to\_ith\_station + fuel, i + 1, remain - distance\_to\_ith\_station, original\_target, stations, memo) + 1);

}

}

memo[curFuel][start] = min\_stops;

return memo[curFuel][start];

}

}

**Style 2: memo[i][j] means minimum stops needed for ith station with j miles remain**

**Memory Limit Exceeded when input as [1000000000, 1000000000], 14/198 test case passed**

class Solution {

public int minRefuelStops(int target, int startFuel, int[][] stations) {

// Need to memoize tuple of (position of station, remain distance to target)

// since these two values keep changing during you recursive calls

// 1 <= target, startFuel <= 10^9 --> need (target + 1)

// remain is same as target range

// memo[i][j] means minimum stops needed for ith station with j miles remain

Integer[][] memo = new Integer[target + 1][target + 1];

int stops = stations.length + 1;

stops = helper(startFuel, 0, target, target, stations, memo);

return stops != stations.length + 1 ? stops : -1;

}

public int helper(int curFuel, int start, int remain, int original\_target, int[][] stations, Integer[][] memo) {

if(curFuel >= remain) {

return 0;

}

if(memo[start][remain] != null) {

return memo[start][remain];

}

int min\_stops = stations.length + 1;

for(int i = start; i < stations.length; i++) {

int passed\_distance = original\_target - remain;

int distance\_to\_ith\_station = stations[i][0] - passed\_distance;

int fuel = stations[i][1];

if(curFuel - distance\_to\_ith\_station >= 0) {

min\_stops = Math.min(min\_stops, helper(curFuel - distance\_to\_ith\_station + fuel, i + 1, remain - distance\_to\_ith\_station, original\_target, stations, memo) + 1);

}

}

memo[start][remain] = min\_stops;

return memo[start][remain];

}

}

**Solution 3: DFS taken/not taken subset implementation / 0-1 Knapsack (TLE, 114 / 198 test cases passed)**

**Python version**

# 3) DFS taken/not taken subset implementation + memorization

self.full\_target = target

mem = dict()

def dfs(curFuel, start, target):

if curFuel >= target:

return 0

if start == len(stations):

return sys.maxsize

if (curFuel, start, target) in mem:

return mem[(curFuel, start, target)]

dis, fuel = stations[start][0] - (self.full\_target - target), stations[start][1]

taken, not\_taken = sys.maxsize, sys.maxsize

if curFuel - dis >= 0:

taken = dfs(curFuel - dis + fuel, start + 1, target - dis) + 1

not\_taken = dfs(curFuel - dis, start + 1, target - dis)

mem[(curFuel, start, target)] = min(taken, not\_taken)

return mem[(curFuel, start, target)]

stops = dfs(startFuel, 0, target)

return stops if stops != sys.maxsize else -1

**Java version**

class Solution {

public int minRefuelStops(int target, int startFuel, int[][] stations) {

int stops = stations.length + 1;

stops = helper(startFuel, 0, target, target, stations);

return stops != stations.length + 1 ? stops : -1;

}

public int helper(int curFuel, int start, int remain, int original\_target, int[][] stations) {

if(curFuel >= remain) {

return 0;

}

if(start == stations.length) {

return stations.length + 1;

}

int passed\_distance = original\_target - remain;

int distance\_to\_ith\_station = stations[start][0] - passed\_distance;

int fuel = stations[start][1];

// The minimum stops needed if take the 'start' indexed gas station to refuel

int taken = stations.length + 1;

// The minimum stops needed if not take the 'start' indexed gas station to refuel

int not\_taken = stations.length + 1;

if(curFuel - distance\_to\_ith\_station >= 0) {

taken = helper(curFuel - distance\_to\_ith\_station + fuel, start + 1, remain - distance\_to\_ith\_station, original\_target, stations) + 1;

not\_taken = helper(curFuel - distance\_to\_ith\_station, start + 1, remain - distance\_to\_ith\_station, original\_target, stations);

}

return Math.min(taken, not\_taken);

}

}

**Solution 4: DFS taken/not taken subset implementation + memorization / 0-1 Knapsack / Top Down DP Memoization (TLE 103/198)**

**Python version**

# 3) DFS taken/not taken subset implementation + memorization

self.full\_target = target

mem = dict()

def dfs(curFuel, start, target):

if curFuel >= target:

return 0

if start == len(stations):

return sys.maxsize

if (curFuel, start, target) in mem:

return mem[(curFuel, start, target)]

dis, fuel = stations[start][0] - (self.full\_target - target), stations[start][1]

taken, not\_taken = sys.maxsize, sys.maxsize

if curFuel - dis >= 0:

taken = dfs(curFuel - dis + fuel, start + 1, target - dis) + 1

not\_taken = dfs(curFuel - dis, start + 1, target - dis)

mem[(curFuel, start, target)] = min(taken, not\_taken)

return mem[(curFuel, start, target)]

stops = dfs(startFuel, 0, target)

return stops if stops != sys.maxsize else -1

**Java version**

class Solution {

public int minRefuelStops(int target, int startFuel, int[][] stations) {

// Use Map and key set up as 'curFuel + "\_" + start' is because traditional

// way to create memo is hard to setup dimension size, especially for current

// fuel, the fuel as given condition is 1 <= fueli < 10^9, which means dimension

// size at least 10^9, which is easy to get Memory Limit Exceeded

// So store two recursively changing keys 'curFuel' and 'start' as a String

// combination is most practical way, but even this way encounter TLE for 103/198

Map<String, Integer> memo = new HashMap<String, Integer>();

int stops = stations.length + 1;

stops = helper(startFuel, 0, target, target, stations, memo);

return stops != stations.length + 1 ? stops : -1;

}

public int helper(int curFuel, int start, int remain, int original\_target, int[][] stations, Map<String, Integer> memo) {

if(curFuel >= remain) {

return 0;

}

if(start == stations.length) {

return stations.length + 1;

}

String key = curFuel + "\_" + start;

if(memo.containsKey(key)) {

return memo.get(key);

}

int passed\_distance = original\_target - remain;

int distance\_to\_ith\_station = stations[start][0] - passed\_distance;

int fuel = stations[start][1];

int taken = stations.length + 1;

int not\_taken = stations.length + 1;

if(curFuel - distance\_to\_ith\_station >= 0) {

taken = helper(curFuel - distance\_to\_ith\_station + fuel, start + 1, remain - distance\_to\_ith\_station, original\_target, stations, memo) + 1;

not\_taken = helper(curFuel - distance\_to\_ith\_station, start + 1, remain - distance\_to\_ith\_station, original\_target, stations, memo);

}

int result = Math.min(taken, not\_taken);

memo.put(key, result);

return result;

}

}

**How to define what dp[i][j] represents ? Why we choose it store maximum distance instead of minimum number of stops ?**

**Refer to**

<https://leetcode.com/problems/minimum-number-of-refueling-stops/discuss/613853/Python-5-solutions-gradually-optimizing-from-Naive-DFS-to-O(n)-space-DP/645219>

Q:

dp[i][j] - "minimum number of refueling stops needed from first i stations to reach distance j"

Could you share code based on this approach? I was trying but got stuck.

A:

"For knapsack problems, when the value amount is too large or there is a limitation on value to reach,

consider to build a dp[i][j] with former i bags, pick j of them instead of the regular way dp[i][j]

with former i bags, value j can be constructed or not."

We can build the 2D DP in multiple ways

dp[i][j] - "minimum number of refueling stops needed from first i stations to reach distance j"

dp[i][j] - "maximum distance which can be reached by refueling 'i' times using some stations from [0,j]" (we can swap i with j)

We can build the DP solution using both interpretations.

The first one is worse since its O(number\_of\_stations \* target).

The solution built using 1 gives TLE. Hence, it is better to use solution 2.

**Solution 5: Bottom Up DP: reversed 0-1 knapsack (2D-DP)**

**Python version**

# 4) DP: reversed 0-1 knapsack.

# Original 0-1 knapsack: maximum value given # of bags limitation: dp[i][j] = bool -- former i bags, j VALUE can be constructed or not

# Reversed 0-1 knapsack: minimum # of bags used to reach a given value dp[i][j] = value -- former i bags, j # OF BAGS being picked, what is the maximum value

if startFuel >= target:

return 0

n = len(stations)

# dp[i][j]: in former i stations, pick j stations to fuel, how far it can mostly reach

dp = [[0] \* (n + 1) for \_ in xrange(n + 1)]

for i in range(n + 1):

dp[i][0] = startFuel

rst = sys.maxsize

for i in range(1, n + 1):

# for j in range(i, 0, -1): ... both works, as long as the i - 1 row has finished, updating i row from left to right/right to left doesn't matter

for j in range(1, i + 1): # j <= i because in former i stations, at most i stations can be picked

dp[i][j] = max(dp[i][j], dp[i - 1][j])

if dp[i - 1][j - 1] >= stations[i - 1][0]:

dp[i][j] = max(dp[i][j], dp[i - 1][j - 1] + stations[i - 1][1])

if dp[i][j] >= target:

rst = min(rst, j)

return rst if rst != sys.maxsize else -1

**Java version**

class Solution {

public int minRefuelStops(int target, int startFuel, int[][] stations) {

// Base case: test out by input 1,1,[]

if(startFuel >= target) {

return 0;

}

int n = stations.length;

// dp[i][j]: in former i stations, pick j stations to fuel,

// how far it can mostly reach

int[][] dp = new int[n + 1][n + 1];

// Initialize no matter given how many former i stations, pick up 0

// station to fuel, the farest position able to reach equal to startFuel

for(int i = 0; i <= n; i++) {

dp[i][0] = startFuel;

}

// Maximum potential stops will be only n, initalize n + 1 the effect same

// as Integer.MAX\_VALUE

int stops = n + 1;

for(int i = 1; i <= n; i++) {

// For j in range(i, 0, -1): ... both works, as long as the i - 1 row has

// finished, updating i row from left to right/right to left doesn't matter

// j <= i because in former i stations, at most i stations can be picked

for(int j = 1; j <= i; j++) {

// Not take ith station to refuel, which means in previous (i - 1)

// stations need to pick up j stations to refuel, so how far (i - 1)

// stations can reach can compare with former i stations to find max

dp[i][j] = Math.max(dp[i][j], dp[i - 1][j]);

// Take ith station to refuel, which means in previous (i - 1) stations

// need to pick up (j - 1) stations to refuel, the pre-requistion is

// to make sure we able to reach ith station {stations[i - 1] is the ith

// station represent in array, and stations[i - 1][0] indicates that

// ith station miles east of the starting position} based on previous

// status (how far it can reach) stored in dp[i - 1][j - 1]

if(dp[i - 1][j - 1] >= stations[i - 1][0]) {

dp[i][j] = Math.max(dp[i][j], dp[i - 1][j - 1] + stations[i - 1][1]);

}

if(dp[i][j] >= target) {

stops = Math.min(stops, j);

}

}

}

return stops != (n + 1) ? stops : -1;

}

}

**Solution 6: Bottom Up DP: reversed 0-1 knapsack (1D-DP)**

**Python version**

# 5) DP, space optimized

--> In DP theory, for Bottom Up DP, the previous status should not be updated anymore and should be fixed value after it initialized,

otherwise later status will have no foundation / basement to calculate, because if previous status change, and all later status

which calculated from fixed formula and previous status will come out as all need to update, which means unstable

if startFuel >= target:

return 0

n = len(stations)

# dp[j]: in former i stations, pick j stations to fuel, how far it can mostly reach

dp = [startFuel] + [0] \* n

rst = sys.maxsize

for i in range(1, n + 1):

# since dp[i][j] relates to dp[i - 1][j] and dp[i - 1][j - 1],

# if updating the compressed 1-d dp array left -> right, dp[j - 1] is updated before dp[j] with row i's dp[i][j - 1] value, which replaced the target value dp[i - 1][j - 1]

# if updating the compressed 1-d dp array right -> left, dp[j - 1] hasn't been udpated when calculating dp[j], which remains the target value dp[i - 1][j - 1]

for j in range(i, 0, -1):

if dp[j - 1] >= stations[i - 1][0]:

dp[j] = max(dp[j], dp[j - 1] + stations[i - 1][1])

if dp[j] >= target:

rst = min(rst, j)

return rst if rst != sys.maxsize else -1

**Java version**

class Solution {

public int minRefuelStops(int target, int startFuel, int[][] stations) {

if(startFuel >= target) {

return 0;

}

int n = stations.length;

// dp[j]: in former i stations, pick j stations to fuel, how far it can mostly reach

int[] dp = new int[n + 1];

dp[0] = startFuel;

int stops = n + 1;

for(int i = 1; i <= n; i++) {

for(int j = i; j >= 1; j--) {

// Since dp[i][j] relates to dp[i - 1][j] and dp[i - 1][j - 1],

// if updating the compressed 1-d dp array left -> right,

// dp[j - 1] is updated before dp[j] with row i's dp[i][j - 1]

// value, which replaced the target value dp[i - 1][j - 1]

// if updating the compressed 1-d dp array right -> left,

// dp[j - 1] hasn't been udpated when calculating dp[j],

// which remains the target value dp[i - 1][j - 1]

if(dp[j - 1] >= stations[i - 1][0]) {

dp[j] = Math.max(dp[j], dp[j - 1] + stations[i - 1][1]);

}

if(dp[j] >= target) {

stops = Math.min(stops, j);

}

}

}

return stops != (n + 1) ? stops : -1;

}

}