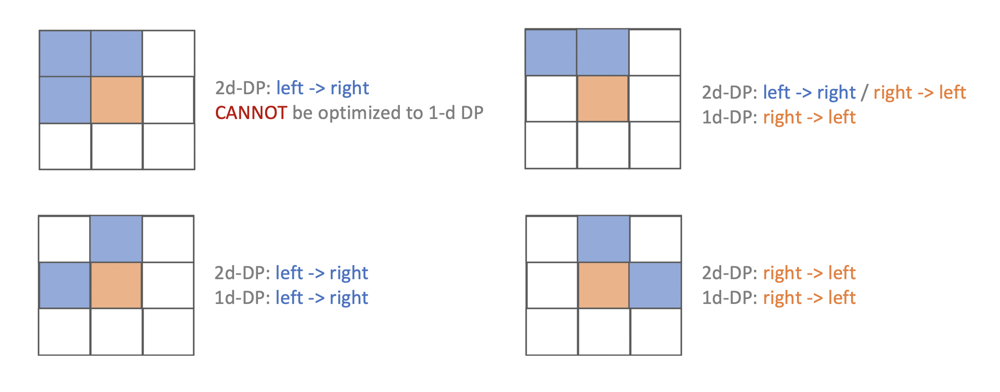
<https://leetcode.com/problems/minimum-number-of-refueling-stops/discuss/613853/Python-5-solutions-gradually-optimizing-from-Naive-DFS-to-O(n)-space-DP>

Notes:

* For a problem with no clear DP idea, always start with **DFS/BFS** solution
* A DFS solution with return values instead of a self.rst global value can be **easier to be rewrotten with memorization**
* 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.
  + **Original** 0-1 knapsack: maximum value given # of bags limitation: dp[i][j] = bool, in 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, in former i bags, j **# OF BAGS** being picked, what is the maximum value.
* In a space-uncompressed dp solution, inner loop's left to right / right to left **updating direction doesn't matter when dp[i][j] not related to dp[i][j +/-1]** , but the compressed 1-d dp's **updating direction matters a lot**, because last row's results might be replaced by current row's ones. Check [here](https://leetcode.com/problems/ones-and-zeroes/discuss/609509/Python-Three-recursive-and-two-DP-solutions) for anthoer exapmle where updating direction needs to be modified because of space optimization.  
  

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

## # 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

## # 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

## # 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

## # 5) DP, space optimized

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

To add some clarification about  
"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

1. dp[i][j] - "minimum number of refueling stops needed from first i stations to reach distance j"
2. 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.

Java Version for above 5 solutions:

## Java solution 1: Native DFS with for loops subset implementation (TLE)

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;

}

}

## Java solution 2: DFS with for loops subset implementation + memorization (Memory Limit Exceeded)

### 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];

}

}

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

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);

}

}

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

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;

}

}

## Java solution 5: Bottom Up DP: reversed 0-1 knapsack (2D-DP)

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;

}

}

## Java solution 6: Bottom Up DP: reversed 0-1 knapsack (1D-DP)

### **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**

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;

}

}